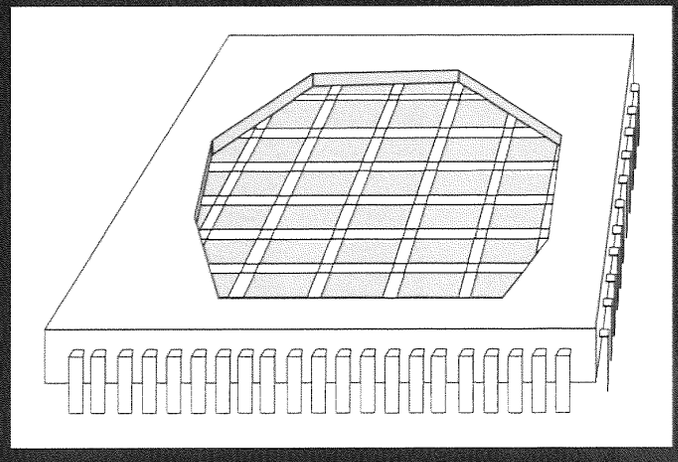
Networks on Chip

Axel Jantsch and Hannu Tenhunen (Eds.)



Kluwer Academic Publishers

Qualcomm Exhibit 1006

Networks on Chip

edited by

Axel Jantsch
Royal Institute of Technology, Stockholm

and

Hannu Tenhunen
Royal Institute of Technology, Stockholm



KLUWER ACADEMIC PUBLISHERS
BOSTON / DORDRECHT / LONDON

Qualcomm Exhibit 1006

A C.I.P. Catalogue record for this book is available from the Library of Congress.

ISBN 1-4020-7392-5

Published by Kluwer Academic Publishers,
P.O. Box 17, 3300 AA Dordrecht, The Netherlands.

Sold and distributed in North, Central and South America
by Kluwer Academic Publishers,
101 Philip Drive, Norwell, MA 02061, U.S.A.

In all other countries, sold and distributed
by Kluwer Academic Publishers,
P.O. Box 322, 3300 AH Dordrecht, The Netherlands.

Printed on acid-free paper

All Rights Reserved
© 2003 Kluwer Academic Publishers, Boston
No part of this work may be reproduced, stored in a retrieval system, or transmitted
in any form or by any means, electronic, mechanical, photocopying, microfilming, recording
or otherwise, without written permission from the Publisher, with the exception
of any material supplied specifically for the purpose of being entered
and executed on a computer system, for exclusive use by the purchaser of the work.

Printed in the Netherlands.

Contents

Preface vii

Part I System Design and Methodology

1
Will Networks on Chip Close the Productivity Gap? 3
Axel Jantsch and *Hannu Tenhunen*

2
A Design Methodology for NoC-based Systems 19
Juha-Pekka Soininen and *Hannu Heusala*

3
Mapping Concurrent Applications onto Architectural Platforms 39
Andrew Mihal and *Kurt Keutzer*

4
Guaranteeing The Quality of Services in Networks on Chip 61
Kees Goossens, John Dielissen, Jef van Meerbergen, Peter Poplavko, Andrei Rădulescu, Edwin Rijpkema, Erwin Waterlander and *Paul Wielage*

Part II Hardware and Basic Infrastructure

5
On Packet Switched Networks for On-chip Communication 85
Shashi Kumar

6
Energy-reliability Trade-off for NoCs 107
Davide Bertozzi, Luca Benini and *Giovanni De Micheli*

7
Testing Strategies for Networks on Chip 131
Raimund Ubar and *Jaan Raik*

8
Clocking Strategies for Networks on Chip 153
Johnny Öberg

9
A Parallel Computer as a NoC Region 173
Martti Forsell

10
An IP-Based On-Chip Packet-Switched Network 193
Ilkka Saastamoinen, David Sigüenza-Tortosa and *Jari Nurmi*

Part III Software and Application Interfaces

11
Beyond the von Neumann Machine: Communication as the Driving Design Paradigm for MP-SoC from Software to Hardware 217
Eric Verhulst

12
NoC Application Programming Interfaces 239
Zhonghai Lu and *Raimo Haukilahti*

13
Multi-level Software Validation for NoC 261
Sungjoo Yoo, Gabriela Nicolescu, Iuliana Bacivarov, Wassim Youssef, Aimen Bouchhima and *Ahmed A. Jerraya*

14
Software for Multiprocessor Networks on Chip 281
Miltos Grammatikakis, Marcello Coppola and *Fabrizio Sensini*

Preface

During the 1990s more and more processor cores and large reusable components have been integrated on a single silicon die, which has become known under the label *System on Chip (SoC)*. Main difficulties of this era were, and still are, the standardization of the component interfaces and the validation of the entire system with respect to its physical and functional properties. Buses and point to point connections were the main means to connect the components. Buses are attractive because they provide high performance interconnections while they can still be shared by several communication partners. Hence they can be used very cost efficiently.

As silicon technology advances further, several problems related to buses have appeared. Buses can efficiently connect 3-10 communication partners but they do not scale to higher numbers. Even worse, they behave very unpredictably as seen from an individual component, because many other components also use them. A second problem comes from the physics of deep submicron technology. Long, global wires and buses become undesirable due to their low and unpredictable performance, high power consumption and noise phenomenon. A third problem comes from the application perspective. Designing and verifying the inter-task communication in a system is a hard problem per se. Getting it to work and dimensioning communication resources correctly is even harder for large bus based communication networks due to the unpredictability of the communication performance. Moreover, every system has a different communication structure, making reuse difficult.

As a consequence, around 1999 several research groups have started to investigate systematic approaches to the design of the communication part of SoCs. It soon turned out that the problem has to be addressed at all levels from the physical to the architectural to the operating system and application level. Hence, the term *Network on Chip (NoC)* is today used mostly in a very broad meaning, encompassing the hardware communication infra-structure, the middleware and operating system communication services and a design methodology and tools to map applications onto a NoC. All this together can be called a *NoC platform.* The breadth of the topic is also highlighted by the scope of this book which ranges from physical issues to embedded software. Quite natural

Qualcomm Exhibit 1006

for a young and quickly evolving research area, the terminology is not yet uniformly used and different authors use the terms differently, as it will become apparent when reading this book.

It should not come as a surprise that the **first part** deals with design and methodology issues. The infamous design productivity gap is one of the strongest, if not the single most important, driving force towards design, architecture and implementation structures. Only when we succeed to restrict the design space in a sensible way will we be able to exploit technology potential. Several chapters of the first part emphasize predictability of the design process and the guarantee of high level features by all implementations.

The **second part** is concerned with the hardware infrastructure. The network topology, power management, fault tolerance, testing and clocking, among other topics, are all key issues that must be solved satisfactorily to make NoCs feasible.

Software and the application perspective is in the center of **part three**. Not surprisingly, communication services and the role of the operating system in future NoC systems are central in the chapters of this part.

Although this book touches upon most of the important NoC issues, many are only superficially dealt with and some key issues are not addressed. NoC is a young and emerging area and we still have to learn to asses the quality of a particular solution, be it for the topology, switch design, communication or operating system services, with respect to an application or application area. We expect for the near future that NoC specific cost and performance will be developed that may be application sensitive. An interesting question is for instance, how to express the communication performance of a NoC. Raw bandwidth may not be adequate for applications with a highly variable traffic pattern consisting of a mixture of real-time control messages and high throughput video streams. In addition, transient faults may occasionally destroy transmitted information, which may make a fault management and retransmission scheme necessary. But again, this will depend on the sensitivity of the application data. Hence, efficiency of a particular NoC system will be more and more expressed in relation to an application or an application area. Existing benchmarks can be used to address this question but new and modified benchmarks will also be required.

As it is indicated frequently in this book, NoC could lead to a fundamental paradigm shift with respect to the way we develop platforms, we design systems and we model applications. At least it will result in a scalable platform architecture for the billion transistor chip era. We expect in any case the NoC area to flourish and prosper and take unexpected and innovative turns and directions and we hope that this book contributes to this exciting research theme.

AXEL JANTSCH AND HANNU TENHUNEN

I

SYSTEM DESIGN AND METHODOLOGY

Qualcomm Exhibit 1006

Chapter 1

WILL NETWORKS ON CHIP CLOSE THE PRODUCTIVITY GAP?

Axel Jantsch and Hannu Tenhunen
Royal Institute of Technology, Stockholm
axel@imit.kth.se, hannu@imit.kth.se

Abstract We introduce two properties of the design process called the *arbitrary composability* and the *linear effort properties*. We argue that a design paradigm, which has these two properties is scalable and has the potential to keep up with the pace of technology advances. Then we discuss some of the trends that will enforce significant changes on current design methodologies and techniques. Finally, we argue that the emerging Network-on-Chip (NoC) paradigm promises to address these trends and challenges and has all prerequisites to provide the arbitrary composability and the linear effort properties. Consequently we conclude that NoC is a likely basis for future System-on-Chip platforms and methodologies.

Keywords: Networks on chip, Productivity gap, System on chip design methodology

1. Introduction

To boost design productivity it is crucial that the effort to add new parts to a given design does not depend on the size of the existing design but only on the size of the new parts. In other words, the design effort must be a linear function of the size of the new parts. If this is the case, large parts and blocks of previous designs can be reused and the design effort can be invested into the new parts. This is also a necessary prerequisite to provide a solid methodology, architecture, and thus a platform, that are sustainable over several technology generations.

The central thesis of this chapter is that a Network-on-Chip (NoC) has the potential to provide such a sustainable platform and, if successful, will incur such a significant change on the system-on-chip architecture

A. Jantsch and H. Tenhunen (eds.), Networks on Chip, 3–18.
© 2003 *Kluwer Academic Publishers. Printed in the Netherlands.*

Qualcomm Exhibit 1006

and design process that it can be called a paradigm change. On the other hand, if it fails to do so, NoC will be just one of several architectures and platforms available to embedded system designers.

> **Arbitrary composability property:** *Given a set of components and a set of combinator operators which allow to connect and integrate the components into larger component assemblages. Components and combinators together are arbitrarily composable if a given component assemblage* **A** *can be extended with any component by using any of the combinators without changing the relevant behavior of* **A**.

Please note, that this and the following property are meant as engineering heuristics, not as mathametical properties. As such they are ideals and can be achieved at higher or lower degrees.

Note further, that this property is defined with respect to what is considered to be a *relevant behavior*. Thus depending on the given objectives and definition of behavior, the same components and combinators may or may not have the arbitrary composability property.

For instance, the standard logic gates `NAND`, `NOR`, `INV`, etc. have this property with respect to their logic level I/O behavior because adding new gates to a netlist of gates will not change the behavior of the original netlist, unless old connections are broken. A given network of gates can be used in any context and will exhibit identical behavior whatever the surrounding netlist may be. New gate netlists can be added to existing ones, using the outputs and results produced by any other part of the circuit without changing the older parts. This is the foundation of our ability to build designs with millions of gates and to reuse large blocks in arbitrary environments.

It should be noted that this nice property of gates is in part due to the implementation process which allows the scaling of transistor sizes, insertion of buffers, and sensible placement and routing by automatic tools.

It is enlightening to see the effects when the arbitrary composition property is violated. Two of the most severe problems in today's designs stem from violations of this property. Timing closure, i.e. the problem to get the timing of the circuit implementation right, is difficult because small changes or addition to the gate netlist may change the timing of the entire system by adding to the critical path or due to an unexpected effect of placement and routing on the timing of seemingly unrelated circuit parts. The system verification problem is so hard because at the system level behaviors are not easily composable and tiny changes in one part may have unexpected effects on seemingly unrelated other parts of

the system. In both cases the design effort grows more than linear with the system size.

If this property is guaranteed, the effort of adding new components to a working system only depends on the new components but not on the size of the reused system (figure 1.1). Thus, a corollary of the arbitrary

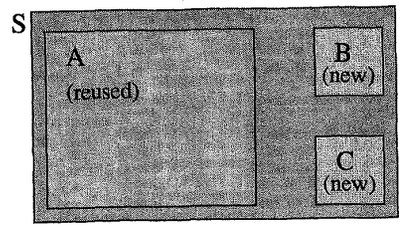


Figure 1.1. With the arbitrary composability property the design effort to add new components to an existing system depends on the integration effort and on the new components but not on the size of A because the design effort to build A has already been spent and is reused as well: $\texttt{Deffort}(S) = \texttt{Deffort}(B) + \texttt{Deffort}(C) + \texttt{Ieffort}(3)$

composability property is the following linear effort property.

> **Linear Effort Property:** *Given is a set of components and a set of combinator operators which allow to connect and integrate the components into larger component assemblages. A design process which builds a system from the components and combinators has the linear effort property if a given set of* n *assemblages* $A_1, \ldots, A_n$ *can be integrated into a system* S *by means of the combinators with an effort dependent on* n *but not on the size of the assemblages:* $\texttt{Ieffort}(n)$. *Thus the total design effort for* S *is*
>
> $$\texttt{Deffort}(S) = \texttt{Deffort}(A_1) + \cdots + \texttt{Deffort}(A_n) + \texttt{Ieffort}(n)$$

Note, that this property implies that the interface complexity of an assemblage does not depend on the size of the assemblage. Obviously, this is not true in practice but it is equally obvious that this is a necessary precondition to build arbitrary large systems. Thus, we must approach this ideal as close as possible to be able to build larger and larger systems. The fact, that we have not been sufficiently close to this ideal is the fundamental reason for the design productivity gap.

We believe that NoC based platforms have a good potential to provide both the arbitrary composition and the linear effort properties to a high degree but they do not automatically guarantee them. We will keep these properties in mind throughout this chapter, but first we review

Qualcomm Exhibit 1006

some of the underlying trends and challenges that lead to NoC and similar architectures.

2. Trends and Challenges

IC manufacturing technology will provide us with a a few billion transistors on a single chip within a few years [1]. Assuming that these predictions hold and that the market will continue to absorb ever higher volumes of ICs, the key questions are: how will the future chips be organized and how will future systems, which include these chips, be designed? There are a few trends which, if continued, will bring about a significant change for architecture and design of integrated circuits.

Communication versus computation. Technology scaling works better for transistors than for interconnecting wires. This leads gradually to a domination of performance figures, power consumption and area by wires and make transistors of secondary importance. At the system level it has a profound effect by changing the focus from number crunching and computation to data transport and communication [2, 3, 4]. Communication becomes often the bottleneck because it seems much harder to design and get right.

Deep submicron effects. Cross-coupling, noise and transient errors are only some of the unpleasant side-effects of technology scaling [5, 6]. It requires significant skills, experience, knowledge and time to keep them under control while exploiting the limits of a technology. A digital or system designer with an expected design productivity of millions of transistors per day is not able to deal with these effects properly. Therefore, designers reuse blocks which are carefully designed by experts with the proper skills. However, it is of critical importance that the deep submicron effects don't pop up again when predesigned blocks are combined in arbitrary ways. Consequently, at the physical design level the property of arbitrary composability means that the electrical and physical properties of blocks are not affected when combining them.

Global synchrony. Physical effects of deep sub-micron technology make it increasingly difficult to maintain global synchrony among all parts of the chip [7]. The clock signal will soon need several clock cycles to traverse the chip, clock skew becomes unmanageable, and the clock distribution tree is already today a major source of power consumption and cost. The trends of scaling to smaller geometric dimension and higher clock frequency make these problems more significant every year.

Thus, it is unlikely that large chips will be synchronous designs with only one clock domain.

Design productivity gap. Synthesis and compiler technology development do not keep pace with IC manufacturing technology development [1]. As a consequence we need either exponentially growing design teams or design time to design and implement systems which fit onto a single IC. Since both alternatives are unrealistic we have in the past escaped from the problem by using ever more complex components as primitive design units. These primitive design units have evolved from individual transistors to logic gates to entire ALUs, multipliers, finite state machines and processor cores. In fact *reuse* of ever more complex design elements has been the main device to increase productivity and will likely remain so in the years to come. It has also kept us close enough to the ideal of the linear effort property to manage the increasing number of transistors.

Heterogeneity of functions. Obviously, systems that can be implemented on a single chip become increasingly more complex. As a result different functions and features with vastly different characteristics and history reside on the same chip. Signal processing algorithms, that recover and generate radio signals, will coexist with global control, maintenance and accounting functions as well as with natural language comprehension and generation functions. These functions are developed in different contexts, by different teams, with different design languages and tools. However, they need to be integrated into a single chip.

In order to lead a concrete discussion we describe next a typical NoC architecture and in section 4 we investigate how a NoC approach could address the listed problems. To paint a fair picture section 5 illuminates the price to pay when adopting a NoC based approach. Finally, in section 6 we speculate how the design process will change.

3. Network on Chip

The Network-on-Chip (NcC) architecture, as outlined in figure 1.2, provides the communication infrastructure for the resources. In this way it is possible to develop the hardware of resources independently as stand-alone blocks and create the NoC by connecting the blocks as elements in the network. Moreover, the scalable and configurable network is a flexible platform that can be adapted to the needs of different workloads, while maintaining the generality of application development methods and practices.

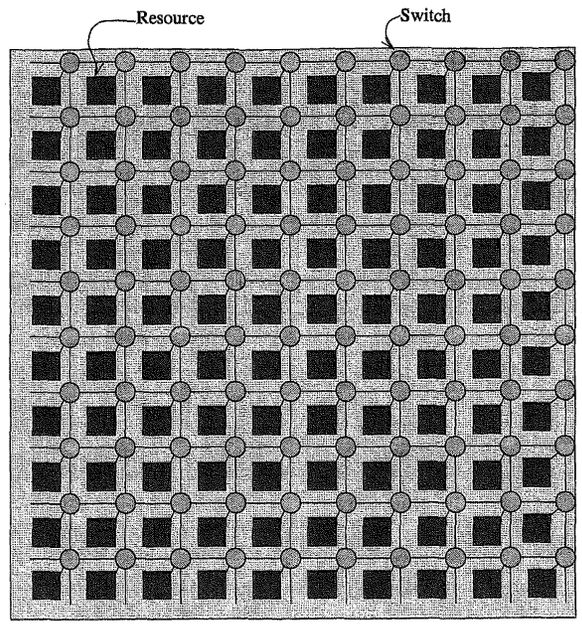


Figure 1.2. Each node in the mesh contains a switch and a resource.

A two dimensional mesh interconnection topology is simplest from a layout perspective and the local interconnections between resources and switches are independent of the size of the network. Moreover, routing in a two-dimensional mesh is easy resulting in potentially small switches, high bandwidth, short clock cycle, and overall scalability. A NoC consists of resources and switches that are directly connected such, that resources are able to communicate with each other by sending messages. A resource is a computation or storage unit. Switches route and buffer messages between resources. Each switch is connected to four other neighboring switches through input and output channels. A channel consists of two one-directional point-to-point buses between two switches or a resource and a switch. Switches may have internal queues to handle congestion. We expect that the area of a resource is either the maximal synchronous region in a given technology or a cluster of computing elements and memory connected via a bus. We expect the size of a resource to shrink with every new technology generation. Consequently the number of resources will grow, the switch-to-switch and the switch-to-resource bandwidth will grow, but the network wide communication protocols will be unaffected.

Note, that this is one, fairly simple NoC variant described here only to substantiate the following discussion. Many other more sophisticated architectures have been proposed [8, 9, 10, 11, 12, 13, 14].

4. How does NoC address the problems

Although there are numerous factors and facets to investigate, we focus on two mechanisms that work in favor of a NoC based approach: *Reuse* and *Predictability*. We believe these are by far the most important factors overshadowing other secondary effects.

4.1 Reuse

As mentioned above, reuse has always been the primary means to bridge the technology gap [15, 16, 17]. More and more complex components, from transistors to gates to functional blocks, such as ALUs and multipliers, to microprocessors and DSP cores, have become the primitive building blocks. In this way, the designers could move up the abstraction levels and describe the system's functionality in more and more abstract terms relying on more and more powerful "primitive" components. Curiously, synthesis technology has mostly been used to bridge one, relatively shallow, level of abstraction. This can be observed in both the hardware and the software domain. Technology mapping, logic and RTL synthesis are in principle straight forward steps with a well characterized optimization space. Synthesis techniques that attempted to bridge more abstraction levels and to make more profound design decisions have typically failed. Examples are high level and architectural synthesis. In software the mapping from C to microprocessor instruction sets is more or less direct and the corresponding compiler technology is tremendously successful. In contrast, compilation from functional, logic and other higher level languages lacks efficiency and has never become mainstream. Thus, we believe that reuse by providing more complex components will continue to be the main mechanism to exploit the potential of technology. Synthesis and compilation techniques will provide the surface mapping from more convenient descriptions onto the primitive components.

However, as a difference to the past, communication "components", or better, communication structures and services have also to become primitive design elements. And this is precisely what a NoC based approach is all about: The reuse of communication services.

Let us briefly review what can be reused in a NoC based approach.

Qualcomm Exhibit 1006

Components and resources. Arbitrary computation elements can be connected to the communication network. In fact we expect that typical NoC based systems may contain processor cores, DSP cores, memory banks, specialized I/O blocks such as Ethernet or Bluetooth protocol stack implementations, graphics processors, FPGA blocks, etc. The size of a resource can range from a bare processor core to a local cluster of several processors and memory connected via a local bus. Resources have to comply to the interfaces of the communication network in order to connect to it and use its services.

The reuse of processor cores has been developed during the last ten years by defining bus interfaces. By defining network interfaces, NoC takes this concept further because it allows to integrate an arbitrary number of resources into a network. In a bus-based system, adding a new resource has a profound impact on the performance of the rest of the system because the same communication resource is now shared among more resources. In a NoC adding new resources also means to add new communication capacity by adding new switches and interconnects. This scalability property is a necessary precondition for the arbitrary composability property but it is not sufficient to guarantee it. The communication network must further be able to guarantee allocated bandwidth and to enforce a decent behaviour of the resources to avoid, for instance the monopolization of the entire communication bandwidth by a single resource.

However, it is important to acknowledge, that a NoC based approach has the potential to provide the arbitrary composability property with tremendous benefits for the design productivity.

Communication infrastructure. The main immediate benefit from a NoC based approach is clearly due to the possibility to reuse the communication network. The switches, the interconnects and the lower level communication protocols can be design, optimized, verified and implemented once and reused in a large number of products. If requirements on performance, reliability and cost differ too much in different application domains, domain specific communication networks can be reused at least for all products in the same domain. For instance, it is likely that mobile, hand-held devices have so different demands on power consumption and performance than infra-structure equipment that different NoC platforms for these two domains are well justified.

Apart from the hardwired communication infra-structure, higher level network services can as well be reused. There is in fact a long list of services that would benefit many applications but can impossibly be developed from scratch for each new product. Examples are

- the detection, monitoring and management of faults in the network;
- the allocation and management of network resources and possibly task migration for load balancing and power optimization;
- the management of global and shared memory;
- the provision of sophisticated communication services such as channels with guaranteed bandwidth and quality of service, multi-cast and broadcast communication, etc.

Most of these tasks are typically provided by the operating system in today's uniprocessor applications. Similarly, a NoC operating system will be very generic and can be reused for many products. Due to the increased complexity of future systems as compared to today's systems, the operating system will be much more sophisticated and complex, thus making the case for reuse even stronger.

Application parts and feature reuse. Reuse will not stop with components and generic services. New products can be composed of existing, complete features. Peeking into the future we can envision a traditional mobile phone which is enhanced by speech analysis subsystem, a speech synthesizer and a language analysis and processing sub-system to provide a spoken language interface to the phone. The same modules or features are apparently useful in a wide range of products and should therefore be reused as much as possible. Obviously, the main challenge will be to define and standardize the high level interfaces between these features to allow for an efficient communication and sharing of information. However, we can observe that a NoC provides an excellent ground for this kind of feature reuse. For optimized implementation a feature may come fully implemented either in software or partially as a dedicated hardware block. Either way, the feature can be plugged into one or several resource slots and the NoC provides at least the low level communication and network services for free for the interaction between the feature and the rest of the system.

Design, simulation and prototype environment. A significant part of system development costs are typically spent in setting up simulation environments and building prototypes. Since many products are based on the same NoC platform much of this investment can be shared by many products. Furthermore, even if different domain specific NoC platforms vary in their performance and power characteristics, they are

Qualcomm Exhibit 1006

sufficiently similar to allow the reuse of much of the design and verification environment across very different application domains. In contrast, application specific platforms which are not derived from the same principle concepts but are developed in an ad-hoc way to suit a particular application domain, will not provide as much potential for cross-domain reuse.

Verification effort. The system verification effort is frequently as high as the design effort itself. Hence, by reusing predesigned and preverified parts and services, verification time can be as drastically reduced as design time. But reuse is even more important for the verification than for the design activity because the uncertainty about the required verification time is much higher and more difficult to plan. Moreover, the uncertainty about the resulting product quality is high and the potential cost of undetected errors in the final product can be enormous. Since risk and uncertainties around verification and verification effects are much higher, verification benefits more from reuse than the design activities. Reused components are typically much better verified, because they have already been used in other products. Moreover, system verification can be done much more effectively when correctness and reliability of the components can be assumed, because errors are identified faster.

In summary, the usage of a NoC based platform boosts the potential for reuse in many ways.

4.2 Predictability

The second main aspect of our focus is predictability and it is in fact closely related to reuse.

Communication performance. Due to its regular geometry and communication network, communication performance becomes potentially much more predictable. "Potentially" and not "necessarily" because the regular communication hardware will significantly help to analyze and assess performance but measures at higher protocol levels and network services have to realize this potential. Since the communication hardware is shared by many resources, the activity of one resource can delay the communication of other resources. One mis-behaving resource can monopolize part of the communication hardware and indefinitely block other resources from accessing it. Therefore, the network has not only to provide the raw bandwidth but also a policy for bandwidth allocation. There are many ways to do this with different advantages and disadvantages. One possibility is to allocate channels with a maximum bandwidth and fixed latency to two communicating resources. By properly allocating and managing these channels many communication activities can coexist peacefully and all of them know exactly the available bandwidth and what latency each data packet will exhibit. This makes systemwide performance analysis feasible and accurate.

As indicated above, a reliable communication resource allocation policy resulting in predictable communication performance for all tasks and applications in the NoC is a central part for establishing the property of arbitrary composability. If a task or a feature can request, obtain and use communication bandwidth independent of all other tasks and features in the NoC, then the cost of reusing, mixing and matching existing tasks and features is relatively small, since they have not to be modified internally.

Electrical properties. Due to the regularity of the layout the electrical and physical properties of the network are well known. The switch-to-switch wires, most likely the longest on the chip except for clock, power and ground wires, have all exactly the same well defined length. Potential uncertainties and irregularities are confined to the resource slots and have most likely no global impact. Thus, the regular and known geometry leads to the accurate analysis of electrical properties. Furthermore, since the NoC platform is reused in many products, elaborate electrical and power models can be developed e.g. to model the dynamic power consumption caused by the communication in the network. High level traffic pattern models can be combined with electrical models for the better assessment of noise and power consumption.

Design and verification time. As already elaborated above, reuse will decrease uncertainties and risk in particular for the verification tasks. This naturally makes design and verification time more predictable. Another reason for increased predictability of the design process is reduced design freedom. Since the network structure and basic services are fixed as well as the size of resource slots, the design space is significantly reduces and the remaining tasks are well separated resulting in more smaller and independent tasks. The remaining system level architectural decisions are the selection of the platform, the network size and the resource allocation. Then all the resources can be designed and implemented separately. Hence, the main challenge is the development of the overall system functionality, its partitioning into processes and their mapping onto resources. However, due to the relatively fixed architecture and the predictable performance and electrical properties (see above) the behavioural design is to a large extent independent from

Qualcomm Exhibit 1006

the implementation phase. This increased modularization of the design tasks makes the overall process more predictable.

5. Disadvantages

Obviously, all these nice properties, outlined above, do not come for free. Essentially we pay for each of them by loosing optimality. Reusing components always means that we use something more general, thus less optimal, than necessary for a particular task. Adopting a fixed and relatively inflexible network topology means that we exclude all the other topologies that may be more suitable for the problem at hand. A particular protocol stack with associated services will not be ideal for all embedded systems and any NoC operating system will often be too general with unnecessary overhead and lacking important services at the same time. Only time will tell if it is possible to find a sensible balance between generality and efficiency which fits sufficiently many applications.

One way to alleviate the problem may be offered by a layered protocol stack and a layered set of services. An application may select a NoC platform only up to a particular level of the stack, thus avoiding the overhead of the upper layers. This flexibility however comes at a price. A strict layering of protocols and services may not be the optimal solution. An integrated implementation of several layers may result in a more efficient design at the expense of the possibility to have direct access to the lower layers.

6. Effect of NoC on the design process

Let us briefly review a design process based on a NoC platform. The main activities are:

Configuring the platform. Depending on the available configuration options this phase may be extensive or insignificant. A more general platform suitable for a wider range of applications will have more configuration options.

Every platform will allow to select the size and most likely offer a selection of communication services that can be included. Every design will need the core service for transporting raw data from sending to receiving resources. This core service may be packet oriented or based on virtual channels. More sophisticated communication means with varying levels of fault tolerance, bandwidth and latency control can be optional. Still higher level services could support transparent task migration from one resource to another or a virtual global shared memory.

Selecting resources. Since every resource slot can receive an arbitrary resource, all the resources have to be chosen by the designer. There may be some constraints because network services like a NoC operating system have to be implemented also. Part of the resource selection task is the design of the memory architecture and I/O architecture, which require specific resources distributed in a particular way.

Reuse of features. The selection of resources is of course intimately connected to the reuse of features. In our terminology a feature is a particular functionality together with its implementation. E.g. a speech analysis feature can be modeled as a task graph and implemented on one or several DSP processors or custom hardware blocks. Thus, a feature can occupy several resource slots but it can also share a resource, e.g. a DSP, with other features. Resource sharing of features has to be considered carefully because it will most likely compromise the property of arbitrary composability unless precautions are taken.

Evaluation and integration. The integration of all features and resources into the final system, the evaluation of performance and the verification of functionality is the final and perhaps most challenging task. An efficient and elaborate simulation and prototyping environment, that can be shared by many design projects, will significantly aid the successful system integration. But above all, the simplicity and predictability of the interfaces of resources, the communication network and features at all levels of the protocol stack will determine how well the property of linear effort can be approximated.

7. Conclusion

Developing a system with several dozens or hundreds of processor like resources is a formidable task. It can only be accomplished if features and components are aggressively reused and if they are *arbitrarily composable* at the physical level, at the architectural and structural level and at the application level (figure 1.3). Or, in other words, the protocol stack from the physical layer to the network and transport layer to the application layer must be well defined such that arbitrary components and features can be connected to the NoC backbone via the protocol stack without affecting the rest of the system.

We have focused on two issues which are particularly crucial for providing the properties of arbitrary composition and linear effort. *Reuse* from blocks to services and features allows designers to productively combine existing parts to new configurations and innovative products. *Predictability* makes reuse efficient because adding new components and

Qualcomm Exhibit 1006

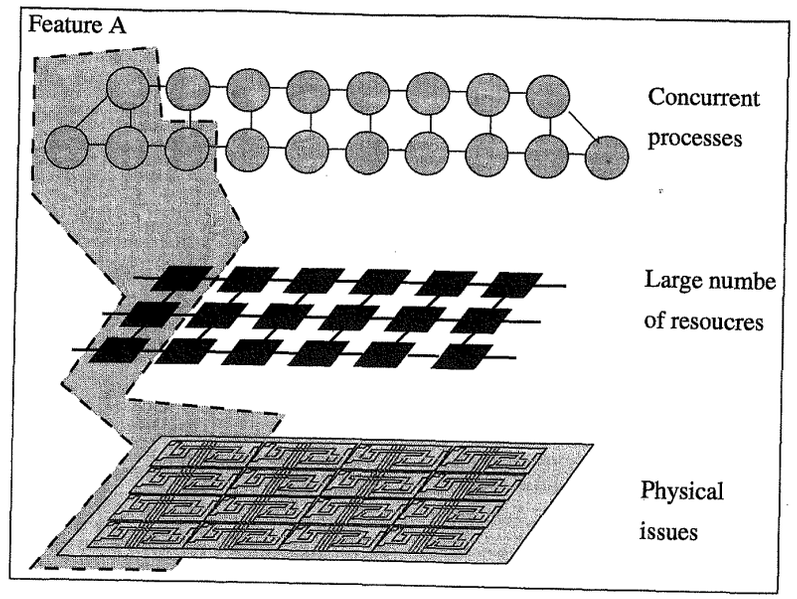


Figure 1.3. Features and components must be arbitrarily composable at the physical, the architectural and the application level to facilitate seamless feature integration.

features does not require redesigning and reverifying the existing parts, thus avoiding a more than linear growth of the design effort.

If NoC based platforms are able to provide the arbitrary composability and the linear effort properties while still allowing for sufficiently efficient product implementations, the design of systems-on-chip will be revolutionized, eventually leading to a "design by feature combination". The NoC paradigm is highly suited to provide SoC platforms scalable and adaptable over several technology generations because it opens the opportunity to define and standardize a communication service infrastructure and a protocol stack from the physical to the application layer. If these are well defined and facilitate the arbitrary composability and the linear effort properties, it will allow the efficient reuse of large resources, communication and network services, and application features. The TCP/IP protocol stack and the Internet, which also revolutionized the use of computers, can be an inspiring example. Indeed, due to their inherent scalability NoC platforms may allow the design productivity to grow as fast as technology capabilities and may eventually close the design productivity gap.

References

[1] Semiconductor Industry Association. Interbational technology roadmap for semiconductors. Technical report, World Semiconductor Council, 1999. Edition 1999.

[2] James A. Rowson and Alberto Sangiovanni-Vincentelli. Interface-based design. In *Proc. of the 34th Design Automation Conference*, 1997.

[3] Kurt Keutzer, Sharad Malik, Richard Newton, Jan Rabaey, and Alberto Sangiovanni-Vincentelli. System-level design: Orthogonalization of concerns and platform-based design. *IEEE Trasnactions on Computer-Aided Design of Integrated Circuits and Systems*, 19(12):1523–1543, Decmber 2000.

[4] Marco Sgroi, M. Sheets, A. Mihal, K. Keutzer, S. Malik, J. Rabaey, and A. Sangiovanni-Vincentelli. Addressing the system-on-a-chip interconnect woes through communication-based design. In *Proceedings of the 38th Design Automation Conference*, June 2001.

[5] Dennis Sylvester and Kurt Keutzer. Getting to the bottom of deep submicron. In *Proceedings of the 1998 IEEE/ACM international conference on Computer-aided design*, pages 203–211, 1998.

[6] Dennis Sylvester and Kurt Keutzer. Getting to the bottom of deep submicron ii: a global wiring paradigm. In *Proceedings of the 1999 International Symposium on Physical Design*, pages 193–200, 1999.

[7] Thomas Meincke, Ahmed Hemani, S. Kumar, P. Ellervee, J. Öberg, T. Olsson, P. Nilsson, D. Lindqvist, and H. Tenhunen. Globally asynchronous locally synchronous architecture for large high performance ASICs . In *Proc. of IEEE Int. Symp. on Circuits and Systems (ISCAS)*, volume II, pages 512–515, Orlando, USA, May 1999.

[8] Pierre Guerrier and Alain Greiner. A generic architecture for on-chip packet-switched interconnections. In *Proceedings of Design, Automation and test in Europe*, pages 250–256, 2000.

[9] Shashi Kumar, Axel Jantsch, Juha-Pekka Soininen, Martti Forsell, Mikael Millberg, Johnny Öberg, Kari Tiensyrjä, and Ahmed Hemani. A network on chip architecture and design methodology. In *Proceedings of IEEE Computer Society Annual Symposium on VLSI*, April 2002.

[10] William J. Dally and Brian Towles. Route packets, not wires: On-chip interconnection networks. In *Proceedings of the 38th Design Automation Conference*, June 2001.

[11] Edwin Rijpkema, Kees Goossens, , and Paul Wielage. A router architecture for networks on silicon. In *Proceedings of Progress 2001, 2nd Workshop on Embedded Systems*, October 2001.

[12] Drew Wingard. MicroNetwork-based integration of SOCs. In *Proceedings of the 38th Design Automation Conference*, June 2001.

[13] K. Goossens, J. van Meerbergen, A. Peeters, and P. Wielage. Networks on silicon: Combining best-effort and guaranteed services. In *Proceedings of the Design Automation and Test Conference*, March 2002.

[14] Michael Bedford Taylor, Jason Kim, Jason Miller, David Wentzlaff, Fae Ghodrat, Ben Greenwald, Paul Johnson Henry Hoffman, Jae-Wook Lee, Walter Lee, Albert Ma, Arvind Saraf, Mark Seneski, Nathan Shnidman, Matt Frank Volker Strumpen, Saman Amarasinghe, and Anant Agarwal. The Raw microprocessor: A computational fabric for software circuits and general-purpose programs. *IEEE Micro*, 22(2):25–35, March/April 2002.

[15] Michael Keating and Pierre Bricaud. *Reuse Methodology Manual for System-on-Chip Designs.* Kluwer Academaic Publishers, 1998.

[16] Terry Thomas. Technology for ip reuse and portability. *IEEE Design & Test of Computers*, 16(4):6–15, October 1999.

[17] Henry Chang, Larry Cooke, Merrill Hunt, Grant Martin, Andrew McNelly, and Lee Todd. *Surviving the SOC Revolution - A Guide to Platform-Based Design.* Kluwer Academic Publishers, 1999.

Chapter 2

A DESIGN METHODOLOGY FOR NOC-BASED SYSTEMS

Juha-Pekka Soininen and Hannu Heusala
VTT Electronics (Technical Research Centre of Finland), P.O. Box 1100, Kaitoväylä 1, FIN-90571 Oulu, FINLAND

University of Oulu, Department of Electrical and Information Engineering, P.O. Box 4500, FIN-90014 University of Oulu, FINLAND

Abstract: Diversity of computational requirements of information technology products will increase. The products will be based on computers, but the full exploitation of silicon capacity will require improvements in design productivity and system architectures. The chapter presents a backbone-platform-system methodology for development of products that are based on a network-on-chip concept. Encapsulation of dedicated subsystems into a NOC system is the key idea of an "integrated distributed embedded system" approach. The cornerstones of the methodology are the reuse of a NOC backbone and a platform architecture. The NOC backbone encapsulates the physical integration problems into communication solution for system architect. The NOC platform provides application area optimized computation and storage resources for system developers. Essential parts of the methodology are decision-support methods for both NOC platform and NOC system designers.

Key words: Design methodology, platform-based design, reuse, decision-support

1. INTRODUCTION

Exploitation of a billion-transistor capacity of a single ASIC requires new system paradigms and significant improvements to design productivity [1].

A. Jantsch and H. Tenhunen (eds.), Networks on Chip, 19–38.
© 2003 *Kluwer Academic Publishers. Printed in the Netherlands.*

The Network-on-Chip types of approaches presented in earlier chapters try to divide and conquer the use of available silicon surface. Integration of tens or hundreds of computers into a single chip together with asynchronous message passing network partitions the design into more manageable units and allows to optimize the subsystems according to application requirements [2]. Such scalable and modular architectures are natural steps in system architecture evolution especially when physical implementations are considered.

Structural complexity and functional diversity of such systems are the challenges for the design teams. Simple extrapolations of design productivity roadmaps reveal that already in the near future only the very largest companies can complete their ASIC projects in case the full silicon capacity is used. In the longer period, it is clear that design capacity is going to be the limiting factor for the system complexity.

Structural complexity can be increased by having more productive design methods and by putting more resources in design work. Design method improvement using higher abstraction levels leads to the more complicated and expensive design space exploration and synthesis tasks. When increasing the number of designers, the communication overheads consume almost all the benefits. The most attractive approach is the reuse of earlier designs, because if the design is reused at least three times the cost of making design reusable is justified. In platform-based design, the reuse is extended to architectures [3] and recent reconfigurable platforms duplicate regular structures of computational units [4].

Increasing number of gates can be and will be used for increasing the functionality of the products. The billion-transistor capacity on silicon and operation frequency at gigahertz range means that we shall have teraoperations per second capacity for applications. In addition to software and computer hardware, the same system can have DSP functions, analog and radio frequency functions, advanced controlling functions, artificial intelligence, etc. The system level decisions must take into account all the different needs of application types. It will require common language, concepts and terminology. The optimization of the implementations of applications will mean the integration of a variety of memory technologies, analog electronics, reconfigurable technologies and computer technologies into same platform.

The structural complexity will force us to operate at the system level when developing NOC-based systems. It will not be possible to go into details when making decisions concerning reuse or application mappings, for example. It will also force us to develop platform-type of architectures and resources. We have to replace the target product requirements with more abstract concepts. We have to use requirements of the product area,

characteristics of typical applications, forecasts on near future applications, etc. This will mean the shift from synthesis or mapping based methods to analysis and estimation based approaches.

Design methodology is a coherent set of theories, methods, techniques and/or principles used to analyze and/or develop methods for a particular domain. The main purpose of the methodology is the partitioning of the design problem into manageable tasks and definition of the tools and practices for those tasks. When considering NOC-based systems, the first problem arises from the diversity of functionality. Most of the existing design methodologies are application area specific. However, these methodologies are results from long evolution and they have built-in knowledge and expertise that should not be ignored. The challenge is therefore to combine these different approaches.

This chapter presents the principles of a NOC design methodology. Since the NOC-based systems have not been designed or manufactured yet, our approach is based on the extrapolations and forecasts. Section 2 gives a brief review of most important existing and evolving design methods and tools. Section 3 describes the requirements of NOC design methodology in rapidly changing technological and economical environment. Section 4 describes the NOC architecture and quality characteristics of NOC. The backbone-platform-system design methodology for NOC-based systems is presented in Section 5. The novelties of the approach are in the layered structure of the methodology, in the integration of physical issues into architecture design, and attempt to manage system complexity by emphasizing the evolutionary nature of products and product development. The decision-making support required at early phases of design is the most challenging part in the presented approach. Finally, Section 6 summarizes this chapter.

2. DESIGN METHODOLOGIES FOR IC BASED SYSTEMS

The motivation behind technological development is the need to produce products that increase our welfare. Such products are needed, used, and profitable. The same basic need is also behind the technological development, which in turn enables us to introduce increasingly complex products. The problem is that the increasing complexity requires more advanced design methodologies, because complexity increases the number of steps necessary for building them.

Design methodology is dependent on the underlying technology, but it is also a cultural issue and a business issue. Sexes, professions, companies, nationalities have their background, history and differences, and they may

need different type of operational environments in order to perform optimally. Inside the companies, the cumulated knowledge, the existing products, and the strategic decisions affect to the design methodologies [5].

The design methodology is responsible for product success in a sense that it must guarantee that the company performs the activities that are needed in quality validation. The main quality factors are performance, cost and variability, but these are very generic and cannot be directly used in product development. Important features of methodology are the transformation of user needs to physical requirements for different abstraction layers and partitioning of those needs and requirements to subsystems and technologies. The quality function deployment (QFD) for example, tries to solve these problems by introducing correlation matrixes for quality characteristics and technical requirements [6].

2.1 Evolution of ASIC techniques and design methods

Since its invention over forty years ago, IC-technology has offered for designers increasing number of switching elements in planar form. Basic logic operations needed to be performed in the digital systems on the surface of a silicon chip are still AND, OR and NOT. So far, the functions of digital systems are based on the Boolean algebra and combinational logic blocks are constructed of those basic logic gates. Memory function is brought to the system by adding feedback connections to a combinational block.

Switching elements are manufactured on the surface of a silicon chip in a silicon foundry applying different available semiconductor manufacturing technologies. Nowadays, and most likely for next ten to fifteen years, it is usually CMOS technology, where every switching element consists of two complementary connected transistors. The function and elementary structure of SOC and NOC systems are still, like in the first digital ICs from early 60's, based on the CMOS switching elements implementing AND, OR and NOT Boolean functions.

Over forty years, the semiconductor industry has been able to develop new manufacturing technology generations. The increasing number of switching elements is offered for designers in different implementation formats. Therefore, we have general-purpose logic (GPL) devices, memories, microprocessors, micro-controllers, programmable logic devices (PLD), field programmable gate arrays (FPGA), ASICs, SOCs and NOCs. Implementation format can be understood to be a form specifying how switching elements are grouped and connected on the chip surface. The format specifies also the customer's and the manufacturer's responsibility in circuit design process.

The GPL devices, e.g. 74/54 and 4000 families, were the first implementation format offered by the semiconductor industry in early 60's. Billions units of them are still used by the electronics industry every year for every kind of products. Schematic entry is most efficient design method with GPL format. GPL format is an example of reuse of silicon layout blocks in the form of SSI/MSI level designs of 10 to 100 gates each. If 50 to 100 pages of schematics is a manageable amount of documentation then a digital system consisting of 500 to 1000 SSI/MSI devices could be possible to be designed in proper time.

Soon after the introduction of GPL format, the semiconductor industry implemented also the first memory devices and then in 1971 the first microprocessor with 2300 transistor switches on a single chip. The combination of microprocessor, memory and GPL devices on the Printed Circuit Board has formed the heart of digital systems known as embedded systems [7]. Embedded Systems are an example of implementation format where the HW reuse is in maximum. Many different systems may have the same HW architecture. The designer personalizes the system by writing the SW code and compiling it into a binary pattern of memory devices. The same amount of man-hours is needed to write 50 to 100 pages of SW code than 50 to 100 pages of schematics. One line of the SW code corresponds to a GPL device in schematic documentation. If one line of SW code could implement more functionality than one SSI/MSI device then SW designers productivity is higher.

Next step was the micro-controller device (MCU) in 70's. MCUs are whole microprocessor systems on a single chip. The microprocessor of a micro controller system cannot have the leading edge complexity because a large share of the silicon area has to be reserved for memory on peripherals. Since late 70's, MCU devices have been the most popular general-purpose implementation format of digital systems.

Pure HW formats, after the introduction of GPL devices, have followed the development from Simple PLD devices, Complex PLDs and FPGAs to different ASIC formats [8]. The problem with schematic entry of PLD/FPGA and ASIC designs is the management of complexity. Schematic entry is applicable to the point where the number of pages is less than 50 to 100. If the symbols used in schematics are normal library components then maximum number of gates in design could be in the range from 5K to 10K. HW description languages like VHDL and Verilog, and logic synthesis tools were the solution for complexity management problems in the 90's [9,10]. The VHDL model of a digital system includes specification of both the HW structure and behavior of the system. Synthesis tool compiles a VHDL model to logic schematics. VHDL improves designer's productivity if synthesis of 50 to 100 pages of VHDL model produces more schematic

Qualcomm Exhibit 1006

sheets. Complexity of HW modeled ASICs is then in the range from 10K to 1M gates and every page of the VHDL model specifies 250 to 10K gates of the implemented logic.

System-on-Chip is an ASIC or PLD/FPGA implementation format where one single silicon chip contain mostly reused IP based logic blocks like whole microprocessor systems. Manageable logic complexity of SOCs is somewhere between 1M gates to 100M gates. The new product specific logic of a SOC could be no more than 1M gates unless more productive design methodology is available.

Networks-on-Chip implementation formats are needed when the complexity available on a single chip requires a network of SOCs. Design documentation of a 1G gate NoC could be 100 pages of schematics where one page describes a structure of 10 million gates. Basic element, schematic symbol, could be 1 million gates IP block taken from the IP library of the design system.

2.2 System-on-chip design methodologies

In the late 1980's, it became possible to develop and implement computer organizations and architectures using ASIC design methods. It was realized that this introduces new possibilities for system optimization, because hardware/software partitioning is more flexible and not constrained by the costs as it was earlier. The first approaches to *software/hardware partitioning* were based either on software or hardware specifications, but very soon the need for functional specification and functional verification were identified, which resulted to *SW/HW codesign approaches* [11,12]. Functional specification was done using either a single language for the complete system [13] or by combining subsystem models that were each specified using application-specific languages [14]. Granularity at which the partitioning was done varied in different approaches from fine-grained instructions and medium-grained coprocessors to coarse-grained architecture level objects. The verification of the partitioned system required *cosimulation* tools, which could simulate both software and hardware [15].

The *system synthesis* approaches generate both the hardware implementation and software generation from the characteristics of functional specification. The design space exploration and evaluation of cost function are critical phases [16]. In the mapping based approaches the idea is to evaluate different mappings of functions to architectural elements. These *function/architecture codesign* approaches allowed using existing architectures or separately modeled architectures and the role of quality evaluation increased [17]. The intellectual property (IP) blocks and virtual components were proposed as a solution for design productivity problems in

late 1990's. The IP-block concept contains both the functionality and implementation. In *IP-based design,* the main problems are related to the evaluation of IP-block quality and integration issues. *Platform-based design* integrated and extended the earlier methods by reusing also system architectures and topologies in addition to components. Platform can also contain the software layer that helps in application development [18].

2.3 System design methodologies

The *waterfall model* divides the software development process into requirement analysis, architecture design, coding, testing and manufacturing phases that are executed sequentially in a top-down fashion [19]. In the *V-model,* the role of testing and verification is emphasized. Each phase of the waterfall model is accompanied by respective validation or verification phase [20]. The *spiral model* is based on the idea that the phases of waterfall model are repeated for different versions of the system [21]. These sequences of versions include conceptual models, specifications, prototypes, and product versions. This successive refinement approach is also called as *incremental development* approach, because the idea is to bring in more details during the development. The *meet-in-the-middle* principle combines the top-down and bottom-up approaches. The idea is to define the primitive components and objects and to develop them and to system structure simultaneously, so that in the middle of design process the final system is constructed from these primitives. ASIC synthesis tools with libraries and IP-based design are derivatives of this principle. In the *hierarchical design* methods, the system is partitioned to subsystems that are designed separately. *Concurrent engineering* attempts to take into account all the aspects of system design from the very beginning of the development process. It aims in developing subsystems simultaneously and increasing the communication between different design teams. Cross-functional teams and information sharing are the main enablers [22]. In *product family development*, the idea is to develop simultaneously a complete set of product variants [23]. These approaches are based on the ideas of mass customization, development of platform product that is easy to modify, or development of core product that can be extended by product features.

3. NOC METHODOLOGY REQUIREMENTS

NOC technology is a solution for complex systems, e.g. systems that can not be implemented otherwise. NOC-based products will have enormous amount of computational capacity and storage capacity in a very small

volume. NOCs will not be feasible unless the number of implementation is large. In consumer products, the mass production quantities can be achieved for example in personal communication terminals, such as mobile phones. Combination of diverse functionality and optimized product characteristics is the main challenge. In infrastructure products, such as information servers and networking products, the NOC technology can provide performance and capacity, but the success depends on how usable the NOC-based platform will be, when different types of products are considered. Third alternatives are generic computation platforms that are implemented using NOC platforms. The challenge is the programming model. The methodology presented in this chapter is mainly targeted to the consumer product segment.

3.1 Technology capacity effects

According to technology forecasts, the number of transistors is going to increase for the next ten years. When we use the 65 nm technology, it is possible to have over 2.4 billion transistors in a single chip. Current system architectures and design styles do not scale up to such dimensions and complexities.

If we consider the available silicon surface and the characteristics of gates and wires from the digital design view, it is obvious that we have to partition our chip area to several clock domains that communicate with each other asynchronously [24]. Similar partitioning is required for testability, energy distribution, and communication reasons. We need to reserve a part of the chip area to this kind of infrastructure and to partition the rest for application purposes, where each part has its own synchronization. This clearly speaks for joint optimization of physical architecture and design methodology.

When we take the computer science view, then the silicon surface is a high-capacity computation platform. Sequential processor can not exploit such a capacity effectively due to earlier mentioned reasons. Operation level parallelism leads to extremely complex design of dedicated hardware. Instruction-level parallelism (ILP) leads to extremely complex control of execution and besides, it is difficult to extract enough ILP from large applications. Thread-level parallelism could be viable alternative, but it requires significantly more threads than in current processors. Task and process level parallelism leads to parallel computer architectures, which have problems related to shared memories and communication, but these architectures could exploit the area more effectively. Besides, the programming model would be simple [25]. Application level parallelism means integration of embedded systems into a single chip and design

approach that resemble distributed system design. Such architecture could exploit all the possible area, but the feasibility requires that the ratio of local and global computations is high [2].

A NOC design methodology has to support both integration and effective use of different implementation technologies. Technology embedding means additional costs in ASIC manufacturing process. On the other hand, the use of different technologies should be justified by final product characteristics. This linkage between the costs and benefits of technologies and applications must be created in the design process.

3.2 Product requirements and economics

Already today, the complex SOC projects stretch the capabilities of companies and all the forecasts imply that product development capacity is lagging behind the technology capacity. Managing functional diversity and complexity are product requirement related issues that must be solved by NOC architecture and methodology.

For example, personal communication devices will have both local and mobile communication capabilities, various types of multimedia features, intelligent user interfaces, etc. Good implementation will require various types of technologies and computing architectures, which means that the implementation of total functionality may consist of a set of optimized subsystems that are integrated by common functional "glue".

Commercial products must meet both the market window and target cost constraints. Design productivity and design reusability are the critical parameters in this respect. The cost/object can be used as measure of effectiveness. If we analyze the cost/transistor in different types of designs it is easy to see that dedicated logic is more costly than IP-block based design or memory design, which benefits from regular and repeatable blocks. If we use cost/function, the software approaches are better than reconfigurable or hardware approaches. This means that in NOC systems we must favor flexibility over fixed solutions and regularity over ad hoc solutions.

The cost of design of NOC will be such that design reuse must be exploited at all levels and reusable units must be very large. In addition to IP-block reuse, we have to reuse the computation architectures, network architectures, software platforms, existing chips, and design methods. This means that we have to have very clearly defined interfaces for connecting resources to the network and for using the network resources with hardware or software. The reuse of the architecture requires means to modify and configure the architecture model so that application area specific platforms can be designed. The reuse of complete chip needs standardized application

mapping procedures that hide the architecture and hardware details from application designers.

3.3 Diversity and complexity

The product development consists of several areas of expertise and in the decision-making process, we have to be aware of effects to all aspects of design. When we start to develop our ideas into the products, the number of details is increased rapidly and the design methods and practices diverse. The design process is actually a sequence of design decisions that limit the design space. In the beginning, the number of possible implementations is largest and each design decision actually removes possible implementations until we have only one left. Working simultaneously in multiple technologies requires firstly that we partition our system functionality to technologies. Partitioning requires that we have portable models that can be transferred between technologies and subsystems. Secondly, we must derive quality criteria for each abstraction level from the quality criteria of the product. Thirdly, we have to make trade-off between different technologies and optimize the complete system instead of subsystems. This means that we need analysis and estimation methods that can be used at early phases of design and that take into account the coupling effects between technologies. Fourthly, we have to validate our decisions at various levels of abstraction. This needs an ability to combine different models. Finally, we have to be able to use technology dependent methods and tools. This requires encapsulation and interfacing capabilities.

In addition to management of work and implementation dependent issues, also business and economical issues significantly affect on how we should develop our products. Economical constraints such as time to profit require the reduction of design effort and design time, which can be achieved with platform-based approach with reuse both at implementation level and design level. The problem is how to guarantee adequate performance and variability. The market success depends on product's capability to satisfy users. User needs are diverse and the trend is to customize the products and services. If it is done with software, we sacrifice the performance features. If we do it in hardware, it is not cost effective. Naturally, the problem is more complex and we should consider the competitive strategy and market situation too.

In NOC design methodology, we must find out how to balance the system characteristics. Most important are complexity vs. feasibility, flexibility vs. suitability for purpose, and generic vs. dedicated solutions. These relate to the problem when and how we should apply reuse instead of design. Complexity/feasibility relates to the principles on how we design our

NOCs. It relates to both hardware architectures and software architectures. Flexibility issues relate to how we identify and encapsulate those parts of the system that must be optimized without loosing the programmability and configurability of our system. The generic/dedicated trade-off is mainly a design effort issue. The problem is the separation of common or generic problems from those that need special methods. The generic problems should be solved so that they become as a part of NOC infrastructure.

4. NOC-BASED SYSTEMS

Networks can be applied to very different kinds of integrated systems and the success of a NOC depends on how effectively it can satisfy the often-contradictory requirements of applicability and performance. Applicability requires flexibility and generality, simple programming models, and simple development of final functionality. Performance requires efficiency, power awareness, dedicated structures, and optimal combinations of resources and functions.

The NOC design methodology presented in this chapter is meant for NOC systems that are based on product family type of product development presented in Figure 7-1. The NOC systems also use a specific architectural approach that is called "*integrated distributed embedded system*". The target application domain is telecommunication systems, but the ideas can be applied to other types of systems as well.

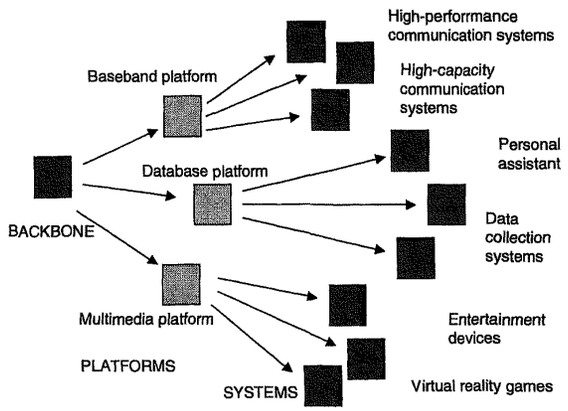


Figure 2-1. NOC-based product family development

We believe that NOC-based systems are economically feasible, if implemented chips can be used in several product variants, and if we can reuse the design in various application areas. However, we also believe that successful product must provide well enough quality for all users, which often requires solutions that are designed to specific needs. Achieving this requires balancing between flexibility and optimality in platforms.

If we compare the different implementation formats for ASIC systems, it is obvious that good performance characteristics need dedicated solutions. The configurable hardware or software based systems are lagging behind in area efficiency and power consumption, for example. If we compare the design effort and design cost the situation is opposite. We combine the benefits of both by providing hardware optimized, customizable computation *platforms* for each application area. The hardware optimization is done by embedding computation resources that are application domain specific into the chip. The design effort is reduced by providing a common basis for the integration, e.g. the *backbone*. The role of the backbone is to provide mechanisms for "distribution" and "integration". Infrastructure services, such as communication and interfaces are the best examples.

4.1 Architecture of NOC-based system

The NOC architecture is presented in Figure 7-2. The architecture is 2-D mesh, which consists of *resources* and network. The network elements are *switches*, *channels* and *resource-network interfaces*. The resources are embedded systems integrated into this architecture. The idea is to separate infrastructure area and application area also in the physical layout.

The only limitations to resources are related to the sizes of resources and to the interfaces. Any type of resource that can be implemented in synchronous area with the silicon technology is possible. From system perspective, the resources are independent embedded systems that have standard interfaces to the infrastructure services. The whole network of these embedded systems can be considered as a distributed system, where the network elements provide communication services.

The *regions* are used for embedding different technologies such as large memory systems, parallel computers, or reconfigurable arrays into our NOC. The size of the region can be larger than the size of resource, and the communication between the region and rest of the NOC can be arranged via specific switches only.

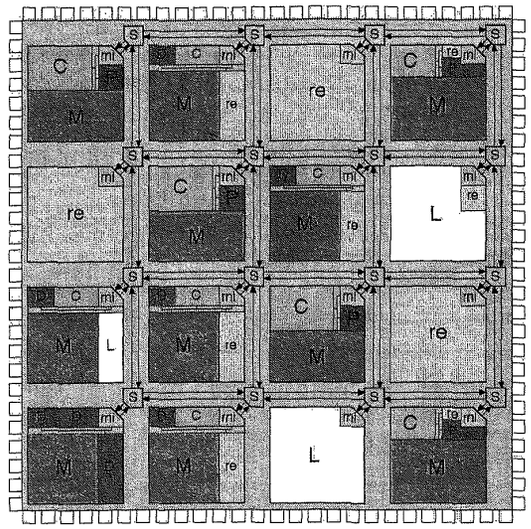


Figure 2-2. Example of a network on chip architecture. S = switch, rni = resource-network interface, P = processor core, C = cache, M = memory, D = DSP core, re = reconfigurable logic, L = dedicated hardware.

4.2 Quality characteristics

NOC based systems have similar quality characteristics as any other system with respect to performance, cost and variability. However, in addition to generic quality characteristics, we can distinguish the quality characteristics for the NOC platform, which are different from product characteristics. Identification of these is important, because they affect our design decisions and the quality assurance procedures.

When considering the NOC platform quality characteristics, we have to consider communication systems, flexibility, and system integration issues. Communication system has to provide performance guarantees for delay, bandwidth, power consumption, and communication reliability. The basic communication functionality has to be accompanied by performance figures to allow application engineers to build reliable applications with predictable performance.

Flexibility is a major concern, since the main point of a platform is to support a variety of concrete products and applications. Hence, the platform has to support different traffic types with widely varying requirements. Safety critical systems will put higher demands on predictability than multi-

media applications. Power management and efficiency is of highest concern in hand-held devices. The matter is complicated further by the prospect that various application types will co-exist in future NOC based products. Thus, a NOC platform has to be careful at offering a sufficient range of choices to application engineers.

In order to support system integration and application mapping, a methodology has to provide sophisticated means to for system analysis. Since the final product will contain a large number of different resources, it will be a formidable task to integrate the performance assessments of the resources with the performance guarantees provided by the network into a reliable system performance assessment.

5. BACKBONE-PLATFORM-SYSTEM METHDOLOGY

The organization of work in our backbone-platform-system NOC design methodology is based on heavy reuse of existing designs, reuse of software systems and reuse of platforms. Our NOC concept encapsulates embedded systems into a network, computers, reconfigurable fabrics, or embedded memories into resources, and intellectual property blocks, virtual components and software into embedded computer systems. The hierarchical encapsulation introduces well-defined interfaces between different layers of NOC system and allows constructing the complete system from black-box type of units.

We have partitioned our design methodology in three ways. Firstly, we have divided the NOC system development in three main layers that are backbone, platform and system development. Secondly, we have divided the methodology according to what kind of methods we need for the development of subsystems. Thirdly, we have divided each design activity into phases according to the purposes of those phases.

5.1 NOC layers

The layered NOC development separates technology specific e.g. backbone issues, application area specific e.g. product platform issues and product instance specific e.g. system issues from each other. The layering enables the separation of infrastructure design from resource design and application design.

The *NOC backbone layer* consists of infrastructure and especially communication services. Infrastructure part consists of physical components and wiring for clocking, energy distribution, and testing, for example. The

communication services consist of physical components such as channels, switches, and network interfaces, and basic services such as protocols for physical and data link layers. The purpose of the backbone is to provide an *integration framework* for platform and resource designers and to hide physical level details from them. It also provides a *communication model* for application developer in a form of protocols. The physical design issues have an important role, because the system-level communication challenges the limits of technology.

The *NOC platform layer* describes the computation platform for target application area. Its main purpose is to hide the hardware from application designer and to provide a *programming model* for them. Platform design requires the understanding of how the target systems operate and what kind of computation they have. Scaling of the network, definition of regions, design of the resource nodes, and definition of the system control are the activities that must be based on abstract models of applications. The platform encapsulates the hardware design problems and serves as a manufacturing integration platform for system developers.

The *NOC system layer* describes programmed chip in the final product. The resource allocation, optimization of network usage and verification of performance and correctness are the main problems that are basically similar to what distributed and parallel system designers have to face.

5.2 NOC design methods

The integrated distributed embedded system concept does not define the embedded resources. The platform designer can use all his creativity to improve the applicability of the platform in the application area domain.

Freedom of choices challenges the design methodology, because consistent methodology must provide tools and techniques that allow designing and modifying everything that are not fixed. We have organized our methodology as a hierarchy of subsystem specific methodologies. The lower level methodologies can be used in implementing the subsystems of upper levels, as long as upper level constraints are fulfilled. The methods at each level are following:

1. System-level design and networked system's design methods. System-level design deals with application modeling and analysis, while network design methods are applied for mapping and load balancing.
2. Parallel computer design, distributed computer system methods, embedded system design, etc. The choice of the method depends on the type of region that we are developing.
3. Computer system design, software system design, codesign, or configuration design depending on the resource, are examples.

Qualcomm Exhibit 1006

4. Basic software, logic and synthesis methods and design flows.

The feasibility of hierarchical methodology depends on how easy it is to model and present the constraints for lower levels and how much these constraints actually effect on the final system. Application programming interfaces and virtual component interfaces are essential elements in successful implementation.

5.3 NOC design phases

The complete design flow consists of a sequence of design activities. The exact set and order of activities is always product specific. In our methodology, a generic design activity is divided into analysis, estimation, decision, and validation phases. *Analysis* phase is needed for understanding what we have as input for our design decision. For example, when software system is designed the first task is requirement analysis, where the objective is to find out what are the user needs that should be implemented but implementation considerations are strictly forbidden. *Estimation* is part of design space exploration where to objective is to forecast the outcome of possible decision without putting too much effort into the implementation of the decision. *Decision* is the phase when selections are implemented as a new system model. Decision includes the creation of a new model, which typically includes the refinements and transformations. In the *validation* phase, the effects of decisions are measured using the new system model as input. The validation should consider all the quality characteristics of the system.

5.4 NOC design flow

The overview of the design flow for NOC-based systems is presented in Figure 7-3. In this figure, the design flow is presented as path through the design plane, where the x-axis direction represent the development of system resources and the y-axis the development of system functionality. In the bottom left corner of the plane, the system does not exist, and in the top right corner, the final NOC-based system is ready. In NOC system development, we divide the problem according to NOC layers, e.g. the backbone development, platform development, and system development.

The *backbone design* aims at providing infrastructure services for different types of NOC platforms. In backbone design, we have to combine the technology dependent implementation and very generic architecture considerations. The analysis phase must consider the silicon characteristics and general principles of system construction. During the estimation phase, the focus must be on production yield, power management and

infrastructure/application area efficiency. In the decision phase, network topology, switches, channels, resource-network interfaces and protocols must be designed as well as the principles of resource integration e.g. how the resource are connected to the infrastructure. The quality of these services should be measured during validation phase as simplicity of resource integration, cost and performance.

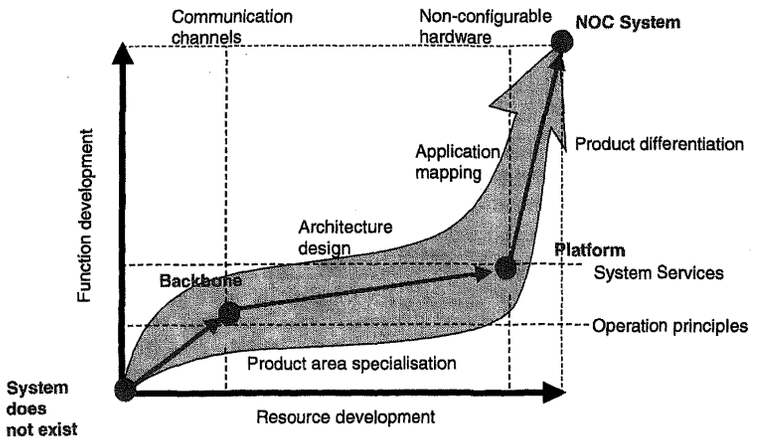


Figure 2-3. NOC design flow

The *platform design* aims at architecture and set of system services that is optimized for a certain product area. The three major decisions concerning the platform architecture are the dimensions of the network, types and sizes of technology regions, and the contents of resources. Understanding the characteristics of typical application workload is the main requirement for analysis phase. The definition of network size and shape requires the analysis of the total complexity of computation, communication and storage. The characteristics of applications affect to technology regions. For example, do we need memory capacity that does not fit into synchronous resource slot, or large areas of reconfigurable arrays or processors, or parallel computer level capacity? Finally, we have to analyze the characteristics of individual functions in order to see what kind of computing resources are needed inside the resource areas. Because of complexity, we have to be able to do these analyses very rapidly and at very abstract level, and current methods and tools are not capable to deal with this problem.

The platform architecture decisions must be supported by feasibility estimations that take into account the application characteristics, resource characteristics and function/architecture mappings. These estimations are

Qualcomm Exhibit 1006

needed both in network level and resource level, and they have to support the evaluation of relative costs of different implementation formats. Performance and efficiency are the behavioral criteria to be considered.

In the decision phase, we must design all the regions and resources. Applying existing design methods and tools is essential. Integration of the models of different blocks both for implementation and estimation purposes requires a *NOC description language* for the modeling of the complete structure and how resources connect to infrastructure services.

The validation at platform level must cover the quality criteria above. Since the platform architecture is a programmable and configurable system that can be used for various types of products, the performance validation must be done using the methods of distributed and computer system design. Analytical performance estimation, workload model based performance simulations, and benchmarks are therefore feasible alternatives.

The final programming and configuration of a NOC system is called the *application mapping*. Application mapping consists of four activities. First, we have to model the whole system. Secondly, we have to partition the functionality into network resources. Thirdly, we have to implement the behavior with the resources. Finally, we have to download the complete system description into our NOC platform and verify it. In various phases of application mapping, we need to construct a complete description of a system although the description itself may consist of several different descriptions. A *NOC assembly language* is therefore needed for presenting the application's mapping to resources and interfacing to platform's system services.

During the application mapping the modeling of the behavior of the system and its workload for the platform elements are main tasks. Modeling of complete functionality so that we can reuse existing subsystem implementations requires high-abstraction level methods and means to add interfacing functions that are platform and resource dependent. Workload models are required during mapping phase for optimization purposes.

Mapping functions to resources is the decision that must be supported by estimates on system quality. Feasibility of platform for chosen set of functions, performances of functions with chosen mappings and utilization of platform resources are figures of merits that must be considered before committing to implementation design.

Validation and verification of complete NOC-based system are problems in which we do not have answers. Built-in self-test structures are most likely alternatives for verification of the platform, but also hierarchical testing approaches with separation of communication, data storage and computation must be developed.

The presented NOC design flow requires new languages for representing the NOC specific structures and services and new methods for supporting NOC related decisions. Proactive decision support methods are needed for efficient reduction of design space. Complexity and capacity estimations are needed for sizing and scaling so that we know the quantities of objects. Mappability estimates are needed for selections of physical and functional objects, so that we know the suitability of objects. Performance and workload estimation is needed for allocation and validation, so that we know the effectiveness of our design. Cost and quality estimation is needed for feasibility analysis. We want to know as early as possible if the work is worth the effort.

Validation methods are needed after decision for analysis of consequences. Validation methods can be more detailed because only a small number of alternatives are designed in more detail. However, the validation methods must also be very effective. Validation must cover all the quality criteria, but if we consider the performance, for example, then the quality validation procedure may consist of network simulation, transaction-level architecture simulations, and instruction-level processor simulations.

6. SUMMARY

NOC-based products will be extremely complex and difficult to design. Management of structural complexity and functional diversity requires that the physical issues, architecture issues and methodology issues are considered and optimized together. Most likely economically feasible products can be developed only if both designs and implementations are reused, and if product functionality and system resources can be separated during the design.

Principles of a backbone-platform-system design methodology have been presented. The BPS methodology is based on the manufacturing platform-based product family development. The application-area specific platforms are based on the NOC backbone that is a generic NOC architecture allowing integration of dedicated computation and storage resources and a set of supporting infrastructure services.

The BPS methodology supports the encapsulation of the design of network elements, e.g. resources and their functionality, and the capability to support design decisions. The most important decisions concern technology and computing architecture selections, system dimensions and mappings at different abstraction levels.

REFERENCES

[1] Allan, A. et al, 2001 Technology Roadmap for Semiconductors, Computer, Vol. 35, No. 1, 2002, pp. 42-53

[2] Kumar, S., et al, A Network on Chip Architecture and Design Methodology, Proc. of 2002 IEEE Computer Society Annual Symposium on VLSI, 2002, pp. 117-124

[3] Chang, H. et al, Surviving the SOC Revolution – A Guide to platform based design, Kluwer, 1999, 235 pp.

[4] Compton, K. and Hauck, S., Reconfigurable Computing: A Survey of Systems and Software, ACM Computing Surveys, Vol. 34, No. 2, 2002, pp. 171-210

[5] Porter, M., Competitive Strategy, Free Press, 1980, 365 pp.

[6] Day, R.G. Quality Function Deployment – linking a company with its customers, ASQC Qulaity Press, 1993, 245 pp.

[7] Wolf, W. Computers and Components - Principles of Embedded Computing System Design, Morgan-Kauffman Publishers, 2001, 662 pp.

[8] Smith, M. J. S., Application-specific integrated circuits, Addison-Wesley, 1997, 1040 pp.

[9] Gajski, D. (ed.), Silicon Compilation, Addison-Wesley, 1988, 450 pp.

[10] MacMillen, D. et al, An Industrial View of Electronic Design Automation, IEEE Trans. on Computer Aided Design of Integrated Circuits and Systems, Vol. 19, No. 12, 2000, pp. 1428-1448

[11] Ernst, R. et al, Hardware-Software cosynthesis for microcontrollers, IEEE Design and Test of Computers, Vol. 10, No. 4, 1993, pp. 64-75

[12] De Micheli, G., Computer-aided hardware-software codesign, IEEE Micro, Vol. 14, No. 4, 1994, pp. 10-16

[13] Kumar, S. et al, The codesign of embedded systems: a unified hardware/software representation, Kluwer, 1996, 274 pp.

[14] Edwards, S. et al, Design of Embedded Systems: Formal Models, Validation and Synthesis, Proceeding of IEEE, Vol. 85, No. 3, 1997, pp. 366-390

[15] Staunstrup, J. and Wolf, W., Hardware/software co-design: principles and practice, Kluwer, 1997, 395 pp.

[16] Eles, P. et al, System Synthesis with VHDL, Kluwer, 1998, 370 pp.

[17] Balarin, F., et al, Hardware-Software Co-Design of Embedded Systems – The POLIS Approach, Kluwer, 1997, 297 pp.

[18] Kreutzer K. et al, System Level Design: Orthogonolization of Concerns and Platform-Based Design, IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, Vol. 19,No. 12, 2000, pp. 1523-1543

[19] Royce, W., Managing the Development of Large Software Systems, IEEE WESCON, 1970, pp. 1-9

[20] McDermid, J.A. (ed.), Software engineer's reference book, Butterworth, 1991, 1500 pp.

[21] Boehm, B.W, A Spiral Model of Software Development and Enhancement, IEEE Computer, Vol. 21, No. 5, 1988, pp. 61-72

[22] Linton, L. et al, First principles of concurrent engineering: A competitive strategy for product development, Report of the CALS/Concurrent Engineering Electronic System Task Group, 1992

[23] Wheelwright S.C. and Clark, K. B., Revolutionizing Product Development, Free Press, 1992, 364 pp.

[24] Hemani, A. et al, Network on chip: an architecture for billion transistor era, Proc. of IEEE NorChip Conference, 2000, 8 p.

[25] Forsell, M., A Scalable High-Performance Computing Solution for Network-on-Chip, IEEE Micro, Vol. 22, No. 5, 2002, pp. 46-55

Chapter 3

MAPPING CONCURRENT APPLICATIONS ONTO ARCHITECTURAL PLATFORMS

Andrew Mihal
Kurt Keutzer
University of California, Berkeley
mihal@eecs.berkeley.edu
keutzer@eecs.berkeley.edu

Abstract Embedded system designers are faced with an expanding array of challenges in both application and architecture design. One challenge is the task of modelling heterogeneous concurrent applications. Another is the task of finding a programming model for heterogeneous multiprocessor architectural platforms. Compounding each of these challenges is the task of implementing heterogeneous applications on heterogeneous architectures. We believe that the key problem underlying each of these challenges is the modelling of concurrency, and the key to modelling concurrency is to capture concurrent communication formally in models of computation. This chapter broadly outlines a disciplined approach to the design and implementation of communication structures in embedded applications. Our approach combines the Network-on-Chip paradigm with the models of computation paradigm. We use models of computation to capture the communication requirements of an application as well as to abstract the capabilities of a communication architecture. Then, application requirements and architectural capabilities are matched using a discipline based on Network-on-Chip principles. In this chapter we describe this approach and present a case study where a Click network routing application is implemented on a multiprocessor architecture using this discipline.

Keywords: model of computation, network on chip, application development environment, programming model, programmable platform, heterogeneous concurrency, multiprocessor architecture, design space exploration, mapping

1. Introduction

Embedded system designers are faced with an expanding array of challenges in both application and architecture design. One challenge is the task of mod-

A. Jantsch and H. Tenhunen (eds.), Networks on Chip, 39–59.
© 2003 *Kluwer Academic Publishers. Printed in the Netherlands.*

elling heterogeneous concurrent applications. The burden of this task is typically born by an *application development environment*. An application development environment is a framework that is tailored to modelling (and principally simulating) system applications. Successful application development environments have considerable specialized support for application-domain-specific modelling including domain-specific libraries and domain-specific models of concurrency.

Examples include environments for Internet packet processing[1], signal processing[2, 3], and reactive systems[4, 5] among many others. Although there are many issues that must be addressed in application development environments, the critical issue is allowing the system developer to naturally organize, partition, and model the concurrency of the application.

Another set of challenges arises with each successive generation of architectural platforms. Moore's law enables an exponentially growing number of architectural features in these platforms. In this chapter we focus on programmable architectural platforms or simply *programmable platforms*[6]. Features in the current generation of programmable platforms include: processing parallelism, special-purpose functional units, intelligent memory structures, networks-on-chip, and peripherals[7]. Processing parallelism comes in three varieties. Process-level parallelism is associated with independent programs and memory spaces. Instruction-level parallelism involves the parallel execution of operations from a single thread of program control. Gate-level parallelism encompasses the varieties of parallelism offered by small special-purpose execution units and reconfigurable structures. The principle challenge of programmable platforms is finding a way to harvest their capabilities in a programming environment.

Many programming environments for programmable platforms consist simply of an assembler. Some go as far as to offer a C compiler and a debugger. Sequential languages like C and assembler do not help the software developer deal with process-level concurrency. Beyond the issue of merely offering primitive software development tools there is the deeper issue of allowing developers to effectively access and exploit the important architectural features of the programmable platform. Developers need a *programming model* of the programmable platform. A programming model is a complete abstraction of the key features of a programmable platform. Its goal is to enable developers to exploit the capabilities of the device while hiding less essential details.

A specific example is the challenge of developing a programming model for the Intel IXP1200 processor[8]. A successful programming model for this device must allow programmers to efficiently exploit the parallelism of six microengines with four thread contexts each, a special-purpose hash engine, and a complex communication architecture with Ethernet peripherals. Although

there are many issues involved in the development of a programming model, the central issue is exposing and managing parallelism.

1.1 The Implementation Gap

Suppose that we have been fortunate enough to find an application development environment that naturally fits our system application. Further suppose that we have found a programming model that adequately abstracts and exports the key features of our target programmable platform. The remaining challenge is to *correctly* map from our application development environment to the programming model of our programmable platform in a way that *efficiently* implements our application. We denote this challenge the *implementation gap*.

Depending on the application domain, there may be a number of problems in crossing the implementation gap. We believe that the fundamental issue is mapping the concurrency captured in the application development environment to the parallelism in the programming model of the programmable platform. This transition needs to be made correctly and efficiently.

Concurrent communication is one aspect of this issue. For example, a signal processing application may be designed using an application development environment like YAPI[3] that is natural for signal processing applications. YAPI contains an abstraction for concurrency based on Kahn Process Networks[9]. This abstraction defines a system of queues that concurrent processes use for communication. The target architectural platform may provide only a single shared memory for communication between all processing elements. The way that the application wants to perform concurrent communication does not match the structures that the architecture provides for performing concurrent communication. This implementation gap problem makes the task of mapping from application development environment to programming model difficult.

This example shows that the implementation gap exists even when designers start with a good application development environment that provides formal abstractions for concurrency. The implementation gap problem can be much worse. If designers choose poor abstractions for concurrency, or design heterogeneous applications that use multiple different abstractions for concurrent communication, the implementation gap grows in complexity.

1.2 A Disciplined Design Methodology

We maintain that modelling concurrency is the core problem behind the design of a successful application development environment, the exportation of a successful programming model, and the correct and efficient mapping from the application development environment to the programming model. The key to the embedded systems design challenge is modelling concurrency, and the key to modelling concurrency is to capture concurrent communication formally

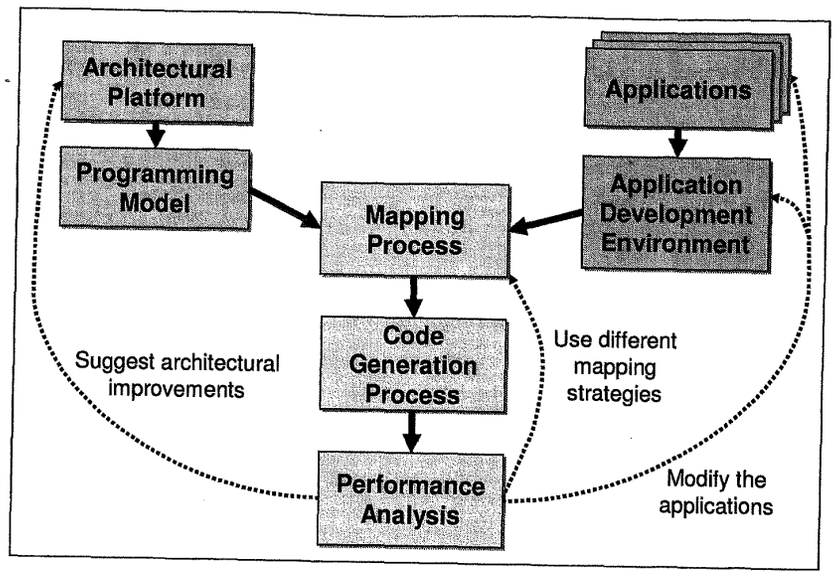


Figure 3.1. Y-chart Design Methodology

in *models of computation*. A model of computation is a mathematical formalism for describing the interaction between components in a concurrent system. Space does not permit a detailed discussion of models of computation, but to understand the principles that are referenced in this chapter the reader should see [10]. Our design methodology uses models of computation to capture the concurrency contained in an application as well as the parallelism provided by a programmable platform. This allows us to compare application requirements and architectural capabilities in a common theoretical framework. Then, we combine models of computation with Network-on-Chip principles to form a discipline for implementation where application requirements are matched with architectural capabilities. The fundamental Network-on-Chip principle we use is the concept of viewing a programmable platform as a composition of computational modules that communicate over an interconnect topology using network protocols.

Figure 3.1 graphically describes the steps in our methodology. The arrangement of steps forms a Y-chart[11]. The Y-chart is an iterative strategy for design space exploration that enables the co-evolution of hardware and software.

In this approach, applications are designed with a high-level application development environment that uses models of computation. This provides software programmers with the right abstractions for modelling heterogeneous,

concurrent applications. Section 2 describes how application environments use models of computation to define the concurrency in an application. In particular, we present the *Communication Implementation View*, an abstraction of the application and the model of computation that formally captures the communication between concurrent components.

At the same time, models of computation are used to export the functionality of the target programmable platform to the developer. The programming model for a specific architecture is that of a machine which performs concurrent communication according to the rules of a particular model of computation. This style of programming model is an appropriate match for application development environments that use models of computation themselves. The concurrent communication implemented by the programmable platform is also captured in the Communication Implementation View. This technique is described in Section 3.

Next, the application development environment and programming model abstractions are combined in a mapping process. In this step, designers match the concurrent communication requirements of an application with the concurrent communication capabilities of an architectural platform using the Communication Implementation View. Section 4 describes a discipline for this task based on the Network-on-Chip principle of viewing the architecture as a communication network. The result of the mapping process is a formal model that describes the implementation of an application on an architectural platform. A code generation process is then used to produce a concrete implementation of the system.

The use of our design methodology and mapping discipline is demonstrated with a network routing application in Section 5. We summarize and draw conclusions in Section 6.

2. Application Development Environments

The goal of an application development environment is to provide programmers with abstractions that are useful for a particular application domain. This is accomplished by separating the concerns of computation and concurrency. Designers focus on implementing the functionality of the application using a domain-specific library of computational components. A model of computation provides a model of concurrency that is natural for the application. It does the work of defining the interaction between computational components. Without this separation of concerns, designers would have to use *ad hoc* techniques to implement the application's computation and concurrency together. Then the definition of the application's concurrency would be implicit and informal. Models of computation define the application's concurrency explicitly and formally.

To implement an application that uses models of computation on an architectural platform, we must implement not only the computation defined by the application's computational components but also the concurrency imposed by the application's model of computation. The Ptolemy II system provides a general strategy for implementing models of computation[12]. Although Ptolemy primarily targets simulation on desktop workstations, the separation of concerns and modular design principles that it uses have broad utility. These underlying theories enable several features that are important for application development environments, especially the ability to design heterogeneous application models that use several different models of computation in combination. In this work we consider applications designed in application development environments that use Ptolemy principles to separate the concerns of computation and concurrency. We extend Ptolemy's general strategy for implementing models of computation to target execution on programmable platforms. Next, we describe the division of labor in the Ptolemy system.

Ptolemy works by separating the functionality of a model of computation into two parts. First, the *communication* portion of the model of computation describes how concurrent application components share data (see Section 2.1). Second, the *control* portion of the model of computation describes how application components execute relative to each other (see Section 2.2). These aspects of a model of computation are implemented with modular components that perform separate communication and control tasks. The *model of computation components* work together with domain-specific library components to make complete, functional application models.

2.1 Communication

Ptolemy library components, called *actors*, communicate through consistent interfaces called *ports*. Ports are connected to other ports with *relations*. The semantics of communication of data *tokens* over a relation is implemented by a model of computation component called a *receiver*. A receiver implements the communication aspect of a model of computation.

Each model of computation has its own receiver, but all receivers implement the same modular interface. The primary functions in the receiver interface are *put*, *get*, *hasToken*, and *hasRoom*. The communication semantics of any model of computation can be implemented through these functions.

2.2 Control

The *director* model of computation component implements the control aspect of a model of computation. Directors tell actors when to execute by calling functions through the *executable* modular interface. The primary functions in this interface are *initialize*, *prefire*, *fire*, *postfire*, and *wrapup*. Directors can

implement any combination of sequential or parallel firing semantics that a model of computation may require.

The director is a centralized control component. Since Ptolemy primarily targets simulation, it is not inefficient to implement even parallel execution semantics with a centralized controller. However, programmable platforms must make a firm distinction between distributed and centralized control. Processes that can execute in parallel should not have to rely on a centralized controller unless there is an explicit execution dependency. We therefore extend Ptolemy's system to include a model of computation component called a *subdirector*. One subdirector is instantiated for each actor in an application model. The director and subdirectors cooperate to implement the control semantics of a model of computation. The director implements the control functionality that is centralized in nature, and the subdirectors implement the control functionality that is distributed in nature.

2.3 The Communication Implementation View

In an application development environment like Ptolemy, the details of how models of computation are implemented are purposefully hidden. Developers do not use directors, subdirectors and receivers directly. Rather, a model of computation is simply specified as a property of the application model. This is exactly the abstraction of concurrency that makes application development environments useful for modelling concurrent applications. However, the details of how models of computation are implemented become important once designers are ready to target a programmable architectural platform.

Our design methodology provides a useful abstraction for capturing these details called the Communication Implementation View. The Communication Implementation View is part of the *multiple views methodology*[13]. In the multiple views methodology, designers use specialized abstractions for different aspects of the embedded system design process. In this case, the abstraction is used for the task of mapping concurrent applications onto programmable platforms.

The Communication Implementation View provides an abstraction of a concurrent application that graphically depicts the details of how the application's concurrency is implemented. Here we make use of the separation of concerns described above that powers Ptolemy's general strategy for implementing models of computation. The Communication Implementation View shows designers an abstraction of an application model that contains the application's computational components together with the model of computation components defined by Ptolemy. This view captures the way in which the application is divided into computation, communication and control aspects.

Qualcomm Exhibit 1006

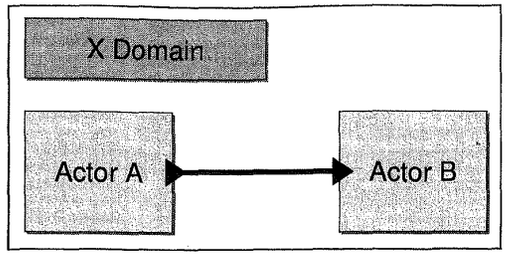


Figure 3.2. Simple Ptolemy Application

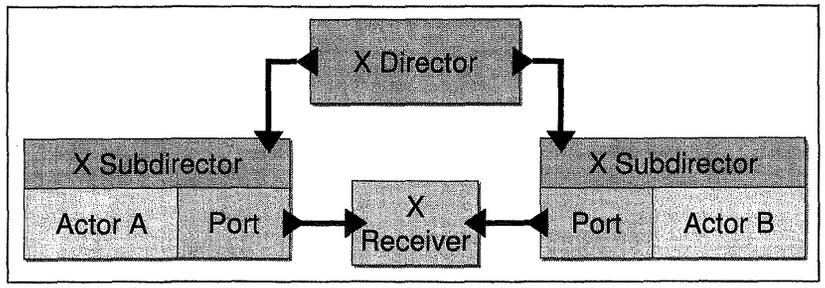


Figure 3.3. Communication Implementation View

For a concrete example, consider the basic Ptolemy application shown in Figure 3.2. This application model consists of a producer actor and a consumer actor that communicate over a single relation. The model has an annotation that indicates the model of computation used. The Communication Implementation View generalizes to all models of computation, so in this example we refer to the application's model of computation with the variable X.

Figure 3.3 shows the Communication Implementation View for this application. The components in the Communication Implementation View are derived from the original application actors and their underlying model of computation components. There is one director that implements the central control for the model, and one receiver that implements the semantics of communication for the relation between the application actors. Each application actor has a subdirector that implements the distributed control of the model.

The Communication Implementation View is useful in particular because it formally captures the concurrent communication of the application. It divides the application into components that *implement* concurrent communication (receivers) and components that *utilize* concurrent communication (actors). We believe that capturing concurrent communication in this manner is the key to

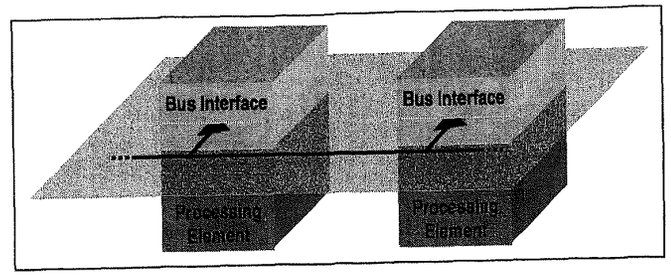


Figure 3.4. General Architecture

mapping concurrent applications onto programmable platforms. In the next sections we elaborate on how the Communication Implementation View is used in our design methodology.

3. Programming Models

Formally capturing an application's concurrency is only part of the design challenge. There is still the issue of formally capturing the capabilities of an architectural platform in a programming model. In particular, we are interested in capturing the concurrent communication capabilities of the target platform's communication architecture. In this section, we describe how models of computation can be used to export this functionality to the programmer.

What characteristics of a communication architecture are important to capture in a programming model? To answer this question, consider the architecture shown in Figure 3.4. In this figure, two processing elements are connected with a bus. This example is a generalization of a more complicated communication architecture. It illustrates the level of granularity that is used to identify the important features of a communication architecture. Each processing element has a hardware bus interface. The bus and bus interfaces together implement a communication link that transfers data between the processing elements. The programming model for the device must describe how software running on the processing elements can utilize this communication link. Examples of the important characteristics of the communication link include:

- *Memory:* The communication link may exhibit the behavior of a shared storage location. Transmitting and receiving data is abstracted as reading or writing to the shared storage. An example is a bus that uses a FIFO for the intermediate storage of data. An important characteristic of this type of communication link is the amount of space available for data.

- *Control:* How does the communication link influence the flow of control of the software running on the processing elements? Transmit and receive operations may exhibit blocking or non-blocking behavior.
- *Data:* A communication link may be designed to handle continuous streams of data, or it may be appropriate only for shorter, intermittent messages. Are data items guaranteed to arrive at the receiver in the same order they were transmitted? Does the communication link guarantee that no data items will be lost, or that the data will arrive without bit errors?
- *Time:* Are there any guarantees on the time it takes to transfer data? Programmers must also know what happens when there is a mismatch in the transmit and receive rates. If a processing element does not read incoming data in a timely fashion, it may be overwritten with new data or the new data may be lost.

The characteristics of a communication link that are used to define a programming model are identical to the characteristics used to describe a receiver, the component that implements the semantics of concurrent communication in the application development environment. Therefore we can use the concept of a receiver as an abstraction for the communication link. The functionality of the communication link is exported to the programmer as a component in the Communication Implementation View called a *virtual receiver*. This is shown in Figure 3.5. This abstraction applies to a specific communication link in the communication architecture. In general, a complex communication architecture will export a set of these virtual receiver components. Each can implement a different communication semantics.

The virtual receiver abstraction tells the programmer that the architecture implements the communication semantics of a particular model of computation. This is a good abstraction for a programming model because it enables a direct comparison between the communication semantics required by an application and those implemented by the architecture. Application requirements appear in the Communication Implementation View in the form of components that *want to use* a particular type of receiver. Architectural capabilities appear in the Communication Implementation View in the form of components that *implement* a particular type of receiver. The next step in the design methodology is to match the application requirements with the architectural capabilities.

4. Bridging the Implementation Gap

In the mapping stage of a design methodology, designers make a transition between the application development environment and the programming model. The challenge in this step is to *correctly* map the concurrent communication

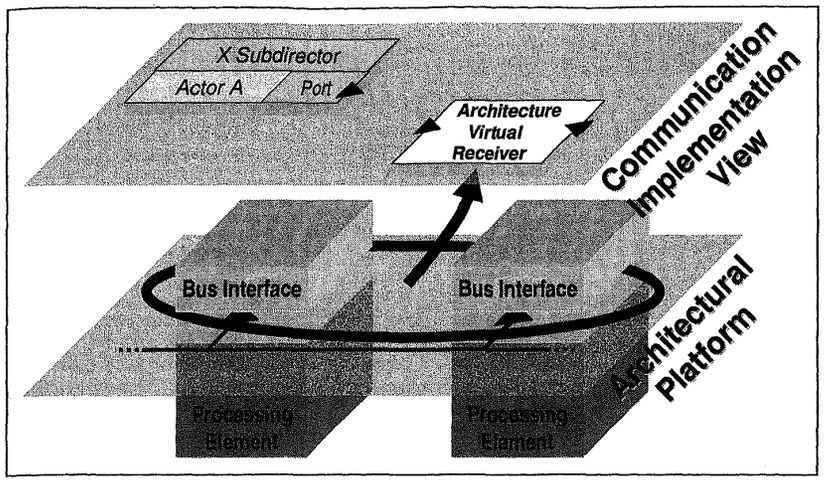


Figure 3.5. Exporting Virtual Receivers

requirements of the application to the concurrent communication capabilities of the programmable platform in a way that *efficiently* implements the application. In our methodology, designers tackle this challenge with the Communication Implementation View abstraction and a discipline based on Network-on-Chip principles.

In the previous section we placed application components and components exported from the architecture together in the Communication Implementation View. The goal of the mapping process is to replace receivers from the application with virtual receivers from the architectural platform. Then, the Communication Implementation View will contain a model that describes how the computational components of the application interact with each other using the communication architecture of the system. This model is called the *communication implementation model*.

Due to the implementation gap, the communication semantics implemented by the communication architecture may or may not match the communication semantics specified by the application's model of computation. Therefore the architectural virtual receivers may or may not be acceptable substitutes for the original application receivers. It is a special case when the application and the architectural platform match naturally. Then, the mapping problem is simplified considerably and high performance implementations can be achieved. Examples of this "natural fit" principle are given in [11]. Designers can make connections in the Communication Implementation View such that the com-

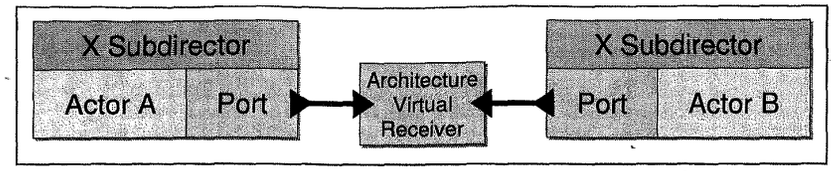


Figure 3.6. Natural Fit Mapping

putational components of the application are directly connected to the virtual receivers exported from the architectural platform. This is shown in Figure 3.6.

This communication implementation model represents a complete solution to the mapping problem. It is a formal model of the interaction between the application and the architectural platform. A designer can then run a code generation process on this model to obtain a concrete implementation of the application that executes on the target programmable platform. This is further described in Section 5.4.

If the application and the architectural platform are not a perfect match for each other, additional work is required to solve the mapping problem. Consider a communication architecture that implements non-blocking transmit and receive operations. The communication semantics implemented by this communication architecture are not an exact match for an application designed using the Communicating Sequential Processes model of computation, which calls for a rendezvous style of communication[14]. Alternatively, consider a communication architecture that does not guarantee that data items will be transmitted without error. These semantics are probably not a good match for any high-level application model of computation. The functionality of the architectural platform and the application can differ in all of the ways that characterize communication semantics - memory, control, data and time.

We can leverage Network-on-Chip principles to close these semantic gaps. The Network-on-Chip paradigm introduces the concept of viewing the architectural platform as a network of communicating computational modules. In our discipline, we extend this idea to formulate the implementation gap as a communication network problem. By definition, a networking protocol provides a particular communication abstraction on top of a set of lower level abstractions. A protocol can be designed to implement rendezvous-style communication on top of a non-blocking infrastructure, or to guarantee error-free transmission on top of a lossy bus. Designers solve the implementation gap problem by creating protocol stacks to match the communication semantics implemented by the architectural platform with those required by the application.

Figure 3.7 shows a model of a typical protocol. In this model, two instances of the protocol communicate with each other using a communication link that

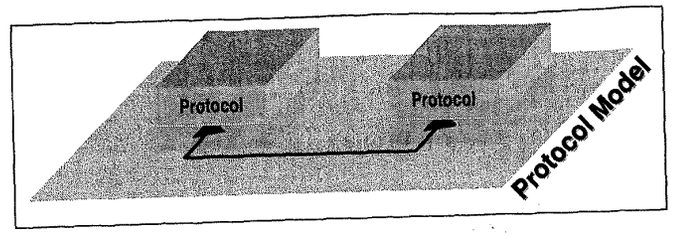


Figure 3.7. Protocol Model

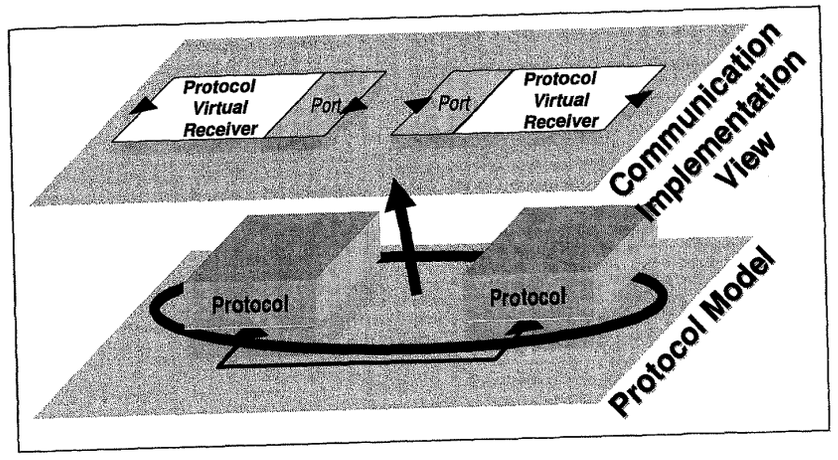


Figure 3.8. Exporting Protocols as Virtual Receivers

implements a particular communication semantics. The combination of the two protocol instances and their communication link implement a new communication link that provides a higher level communication semantics. The functionality of a protocol-based communication link can be exported into the Communication Implementation View in much the same way as an architectural communication link. The difference between exporting a protocol-based link and an architectural link stems from the fact that the protocol-based link not only *implements* the receiver interface, it also *requires* a receiver for the lower-level connection. Therefore the virtual receiver for the protocol-based link is split into two halves, as shown in Figure 3.8. The gap in the middle indicates the protocol's requirement for a lower-level connection. It is meant to be filled in with a lower-level virtual receiver.

An architectural virtual receiver that was not directly appropriate for the application components may be a suitable receiver for the communication link between the protocol components. Figure 3.9 shows this relationship graphi-

cally. The bottom layer in this figure represents the architectural platform. The communication link shown in that layer provides communication functionality for the protocol components in the middle layer. The protocol builds on top of this functionality to provide a higher level of communication semantics for the application actors in the top layer. Figure 3.10 shows how these communication abstractions are described in the Communication Implementation View.

The use of layers of abstraction to bridge the implementation gap is the core of our discipline for solving the mapping problem. Designers can chain any number of protocols together to implement the desired communication abstractions, using both top-down and bottom-up techniques. Protocol stacks reminiscent of the OSI model follow directly[15]. Behavioral type checking is used to determine that all connections between components that require the receiver interface (application actors and protocols) and components that implement the receiver interface (protocols and architectural virtual receivers) are valid[16]. This is used to prove that the protocol stacks designers create in the Communication Implementation View correctly implement the semantics of concurrent communication required by the application's model of computation.

In summary, this discipline is important and useful because it gives designers the necessary framework for understanding the implementation gap. Instead of using *ad hoc* techniques to match the concurrent communication required by an application with that provided by an architectural platform, designers can think about the problem as a communication network problem and use networking principles to reach a solution. Network-on-Chip protocols implement a transition between the application development environment and the programming model. The Communication Implementation View provides a useful high-level abstraction for making this transition. The result is a formal model that describes how the computational components of the application interact with each other using the communication architecture of the system. This technique quickly reveals the roots of the mapping problem and provides the right abstractions for a disciplined solution.

5. Design Example

In this section we will apply this design methodology to a network processing application. This application domain is interesting because there exists a remarkable implementation gap between network processing application development environments and network processor programming models.

It is important to clearly distinguish between the network processing application area and the Network-on-Chip paradigm for system design. The network processing application area is concerned with traditional computer network problems such as IP routing, network address translation, quality of service, and others. A network processor is an architecture designed specifi-

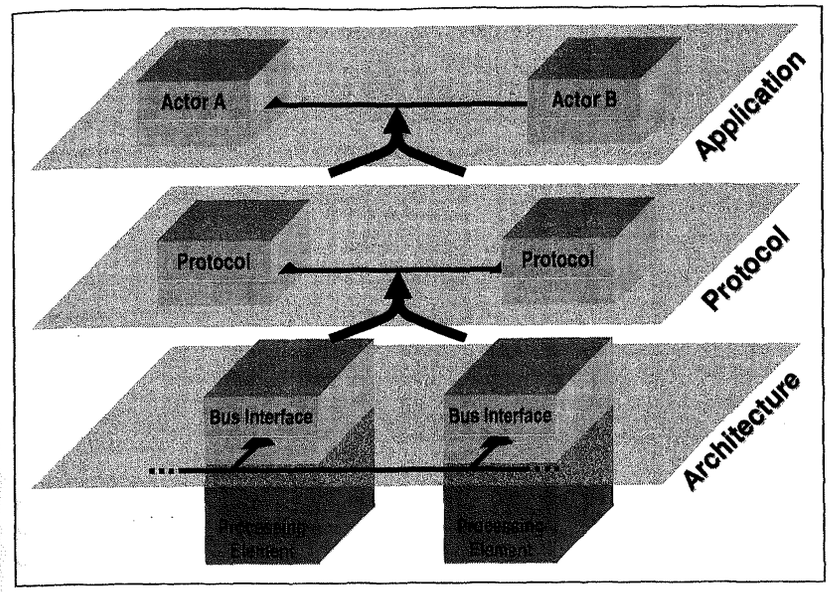


Figure 3.9. The Layers of Abstractions Behind a Protocol Stack

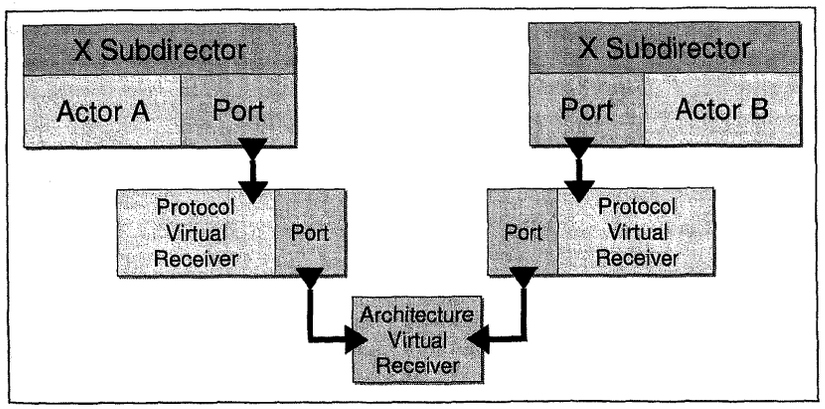


Figure 3.10. A Protocol Stack in the Communication Implementation View

Qualcomm Exhibit 1006

cally for network processing applications[7]. In this example, we consider a Network-on-Chip architectural platform that implements a network processing application. Network-on-Chip and network processing share many of the same concepts and concerns. The similarity between the application that the system performs (the handling of Internet packets) and the internal workings of the system (communication via Network-on-Chip packets) is merely coincidental.

An example of a network processing application development environment is the Click Modular Router[1]. The goal of Click is to provide designers with the right abstractions for describing the inherent characteristics of packet processing applications. These characteristics include manipulating bits in packets, buffering packets, and controlling the flow of packets (both spatially and temporally). Click combines an extendable library of components with a model of computation and a tool set for producing executable implementations. Users assemble dataflow-like graphs of library components to describe a packet processing task. The Click model of computation defines the concurrency in these graphs. It abstracts away the details of how parallel streams of packets flow through the graph.

High-level Click models can be executed with reasonable performance on desktop machines, but there is no current methodology to target any of the numerous network processor architectures that have been proposed[7]. These heterogeneous multiprocessor architectures are supposed to excel at packet processing. Unfortunately, they are extremely difficult to program. Programming models for current network processors do not give developers the abstractions they need to implement concurrent applications like Click routers efficiently and correctly. The challenges here are twofold. The first challenge is to export a programming model from a network processor architecture that captures the parallelism implemented by the machine. The second challenge is to map the concurrency in a Click application to the concurrency captured in the programming model. Our disciplined design methodology can tackle these challenges.

5.1 Application Model

Figure 3.11 shows a graphical model of a Click router application based on the design in [1]. To simplify this example, we have eliminated the error handling functionality of the router and have shown only two channels. This application model was designed in Teepee, one of the core tools of the Mescal project[13]. Teepee is based on Ptolemy II and provides an implementation of the Click model of computation following the principles described in Section 2. This Click model can be simulated within Teepee to test for functional correctness using Ptolemy's standard simulation capabilities.

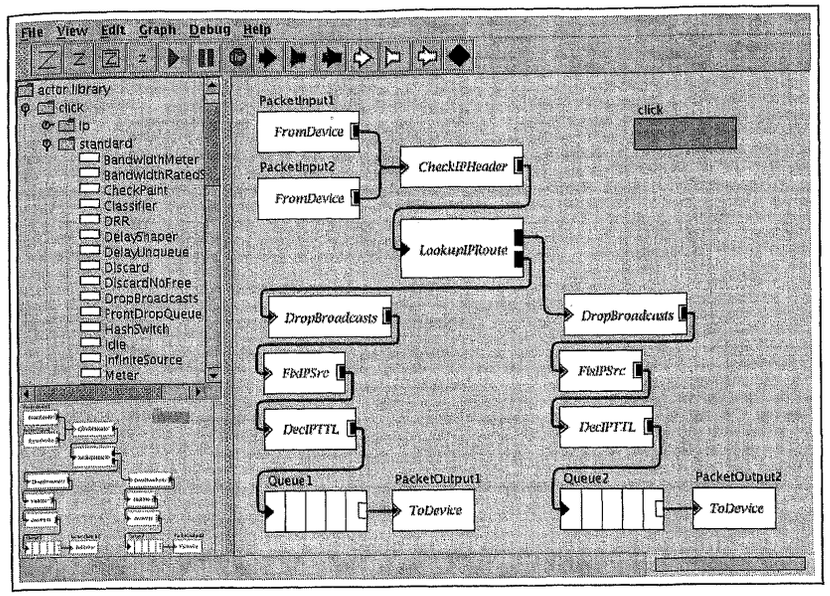


Figure 3.11. Simplified Click Router Application

5.2 Target Architectural Platform

The target architectural platform consists of six processing elements that are connected with two shared buses as shown in Figure 3.12. This architecture is similar to the microengines and internal buses of the IXP1200 processor. Two external interface units connect the on-chip shared buses to off-chip buses. One external bus is connected to Ethernet MACs and the other is an external memory bus. Teepee extends Ptolemy's Discrete Event modelling capabilities to describe and simulate architectural platforms such as this one.

5.3 Mapping

The first step of the mapping process is to choose a partitioning of the computational components of the application onto the processing elements in the architectural platform. Currently this is done manually. Here we choose to dedicate one processing element to each FromDevice packet source element. These elements perform the work of reading in packets from the Ethernet MACs and separating packet headers from data payloads. A third processing element will perform the CheckIPHeader and LookupIPRoute operations. Two final processing elements are used for the output portions of the router application.

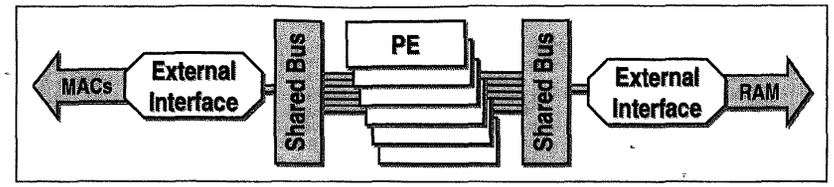


Figure 3.12. Target Architectural Platform

The second step is to implement the connections between the application components in the Communication Implementation View. The partitioning we have chosen requires that some of the application components communicate with application components on different processing elements. These connections will be made using stacks of virtual receivers.

The communication semantics specified by the Click model of computation state that the flow of control follows packet headers as they are passed from actor to actor. Compared to other models of computation, these semantics are not restrictive and are therefore compatible with a broad range of behavioral types. However, the broadcast nature of the shared buses makes the hardware virtual receivers unusable directly. A protocol is required to provide the abstraction of multiple point-to-point communication links on top of this broadcast medium. Teepee provides such a protocol, called Address. The Address protocol works by prepending Network-on-Chip messages with a data field that contains a destination address. Instances of the Address protocol only keep messages if they match the destination address.

The communication semantics implemented by the Address protocol actors are an acceptable match for the Click application actors. A portion of the communication implementation model is shown in Figure 3.13. This model describes how the two FromDevice actors communicate with the CheckIPHeader actor using a shared bus and the Address protocol. The shared bus is exported into the Communication Implementation View as a SharedBusInterface virtual receiver with six ports (one for each processing element that the shared bus connects to). Instances of the Address protocol break the shared bus into separate point-to-point links. The remainder of the connections in the Communication Implementation View are made in a similar fashion.

5.4 Code Generation

A complete communication implementation model is a formal model of the interaction between the application and the architectural platform. Together with the model of the target architectural platform, it contains all of the information necessary to produce a concrete implementation of the system. The

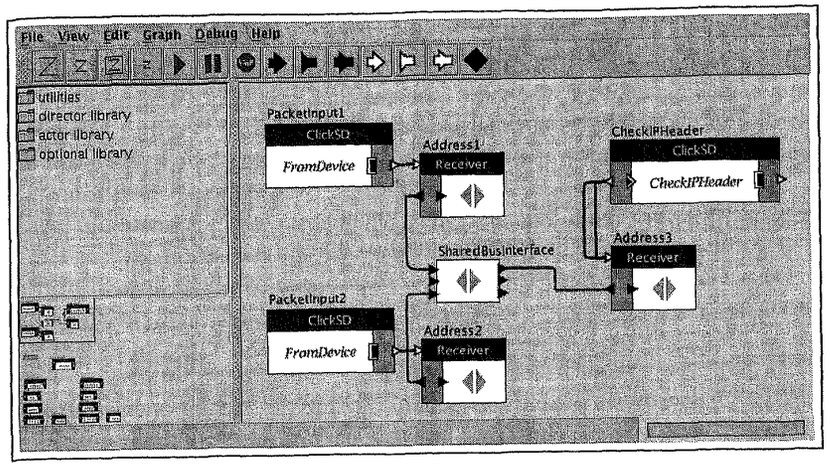


Figure 3.13. Portion of Click Router Communication Implementation View

mathematical formalisms behind these models allow us to produce implementations in a correct-by-construction fashion. This process is called code generation. Examples of code generation targets include compiled-code simulators, synthesizable HDL, and executable machine code.

To produce executable machine code for the programmable platform, the code generation algorithm implements each object in the communication implementation model as a software object on the programmable platform. Application actors such as FromDevice and LookupIPRoute are realized as software processes. The same is true for protocol components such as the Address actors. Architectural virtual receivers (e.g. SharedBusInterface) are synthesized as device drivers for the programmable platform's communication architecture. Each software object is assigned to a particular processing element according to the partitioning made in the mapping step.

Our current code generation algorithm produces an individual C++ program for each processing element. These programs initialize software objects and manage threads of control. The initialization and execution semantics for all software objects are specified by the mathematical formalism that powers the Communication Implementation View. No additional operating system functionality is required. Standard C++ compilation techniques are used to produce machine code for each processing element.

6. Conclusion

Embedded system designers are faced with three important challenges. First, there is the challenge of modelling a heterogeneous concurrent application in an application development environment. Second, there is the challenge of capturing the important capabilities of a programmable platform architecture in a programming model. The third challenge is to correctly map from the application development environment to the programming model of the target platform in a way that efficiently implements the application.

We believe that the key to overcoming these challenges is to properly model concurrency in both the application and the architectural platform. In this chapter, we have introduced a new disciplined design methodology that focuses on formally modelling concurrency using models of computation. In this approach, application developers use the abstractions provided by models of computation to model heterogeneous, concurrent applications. At the same time, models of computation are used to abstract the parallelism of an architectural platform in a programming model. The use of models of computation in both the application development environment and the programming model gives designers a common theoretical framework for comparing application concurrency and architectural concurrency. This enables a disciplined approach to the solution of the mapping problem. Designers use the Communication Implementation View to abstract the mapping problem as a communication networking problem. Then, Network-on-Chip principles are employed to close the implementation gap efficiently and correctly.

Moore's law drives the exponential growth in the complexity of embedded system architectures. The latest generation of heterogeneous, concurrent architectures are accompanied by the latest generation of heterogeneous, concurrent applications. Without a disciplined design methodology, designers will have to resort to *ad hoc* techniques to implement concurrent applications on complex architectural platforms - a doubtful proposition. Using the appropriate abstractions to correctly model concurrency, designers can tackle the design challenges put forward by these systems.

References

[1] E. Kohler et al. The Click Modular Router. *ACM Transactions on Computer Systems*. 18(3), pg. 263-297, August 2000.

[2] The MathWorks, Inc. SIMULINK User's Guide. 1993.

[3] E.A. de Kock et al. YAPI: Application Modeling for Signal Processing Systems. *Proc. of the Design Automation Conference*. June 2000.

[4] G. Berry and G. Gonthier. The Esterel synchronous programming language: Design, semantics, implementation. *Science of Computer Programming*. 19(2), 1992.

[5] D. Harel. Statecharts: A Visual Formalism for Complex Systems. *Science of Computer Programming*. 8(3), pg. 231-274, 1987.

[6] K. Keutzer, S. Malik et al. System-level design: orthogonalization of concerns and platform-based design. *IEEE Transactions on CAD*. vol. 19, pg. 1523-1543. 2000.

[7] N. Shah. Understanding Network Processors. *Master's Thesis*, Dept. of Electrical Engineering and Computer Science, Univ. of California, Berkeley. 2001.

[8] Intel Corporation. IXP1200 Network Processor Datasheet.

[9] G. Kahn. The semantics of a simple language for parallel programming. *Proc. of IFIP Congress*. August 1974.

[10] E. A. Lee. Computing for Embedded Systems. *IEEE Instrumentation and Measurement Technology Conference*. May 2001.

[11] B. Kienhuis, E. Deprettere et al. A Methodology to Design Programmable Embedded Systems. In LNCS series of Springer-Verlag, Volume 2268, *SAMOS: Systems, Architectures, Modelling and Simulation*. Eds. F. Deprettere et al. November 2001.

[12] J. Davis, R. Galicia et al. Ptolemy II: Heterogeneous Concurrent Modeling and Design in Java. *Technical Report UCB/ERL No. M99/40, University of California, Berkeley*. 1999.

[13] A. Mihal, C. Kulkarni et al. A Disciplined Approach to the Development of Architectural Platforms. *IEEE Design and Test of Computers*. November/December 2002.

[14] C. A. R. Hoare. Communicating Sequential Processes. *Prentice Hall International Series in Computer Science*. 1985.

[15] M. Sgroi, M. Sheets et al. Addressing the System-on-a-Chip Interconnect Woes Through Communication-Based Design. *Proc. of the Design Automation Conference*. pages 667-672, June 2001.

[16] E. A. Lee and Y. Xiong. System-Level Types for Component-Based Design. *First Workshop on Embedded Software, EMSOFT2001*. October 2001.

Chapter 4

GUARANTEEING THE QUALITY OF SERVICES IN NETWORKS ON CHIP

Kees Goossens, John Dielissen, Jef van Meerbergen,
Peter Poplavko†, Andrei Rădulescu, Edwin Rijpkema,
Erwin Waterlander, and Paul Wielage
Philips Research Laboratories, Eindhoven, The Netherlands
† *Technical University of Eindhoven, Eindhoven, The Netherlands*
Kees.Goossens@philips.com

Abstract Users expect a *predictable quality of service* (QOS) of embedded systems, even for future, more dynamic, applications. System-on-chip designers use networks on chip (NOC) to solve deep submicron problems, and to divide global problems into local, decoupled problems. NOCs provide services through protocol stacks, and introducing *guaranteed* services enables IP re-use and platform-based design. It also provides globally predictable behaviour, as required by the user, when combining local, decoupled solutions. There are several levels of QOS commitment (correctness, completion, completion bounds), with increasing cost. A combination of guaranteed and best-effort (no commitment) services combines their respective attractive features: predictable behaviour, and good average resource utilisation. The ÆTHEREAL NOC is an example of this approach, and forms the basis of a QOS-based design style, as advocated in this chapter.

1. Future applications and systems on chip

In this section we raise the following question: *why and how must systems implementing future applications guarantee the quality of service* (QOS)? We look at each of the components in turn, in the context of embedded systems and systems on a chip (SOC). We then observe that new SOC architectures and design methods divide global problems into local ones, and rely on networks on chip (NOC) to compose local solutions. We answer the question, in Section 2, by presenting a synthesis of a QOS-based design style and networks on chip. Section 3 explains that QOS can be decomposed into several levels of

A. Jantsch and H. Tenhunen (eds.), Networks on Chip, 61–82.
© 2003 *Kluwer Academic Publishers. Printed in the Netherlands.*

Qualcomm Exhibit 1006

commitment, with different resource requirements. In Section 4 we introduce the ÆTHEREAL NOC, and compare it to other NOCs, in Section 5.

1.1 Characteristics of future applications

We are witnessing the *convergence* of previously unrelated application domains: computation (personal digital assistants, mobile computing), communication (telephone, videophone, networking), and multimedia (audio, photography, video, augmented and virtual reality). Convergence leads to increased functionality and heterogeneity, as previously unrelated functions are combined. Systems become more dynamic, or even unpredictable, as new algorithms seek to take advantage of the disparity between average and worst-case processing. Compressed audio and video data of MPEG2 is an example. Algorithms are also shifting to higher semantic levels, and the semantic complexity of data is less predictable than its volume. An example is the shift from constant pixel processing (scaling, edge enhancement, and so on), to MPEG4 object detection and synthesis, to face recognition and scenario detection. Finally, increased interaction with the environment also makes systems more dynamic. System functionality can be influenced by the current location (affecting communication or computation capabilities), environmental conditions (precipitation and multipath interference can affect mobile communication), and other systems in the proximity (e.g. car guidance, ad hoc networking).

Trends such as ambient intelligence and the networked home make future applications *embedded* and *pervasive*. Moreover, applications acquire a responsibility for the control of physical objects, such as home heating systems, cars, and so on. Real-time performance and safety are critical in many of these applications.

Below, we examine how users interact with these applications, and then what is required to design these systems.

1.2 Quality of service: a user view

Users expect a certain behaviour of applications; in other words, they must be *predictable*. While those expectations may be low, as is often the case for personal computers, a certain fitness for purpose is always assumed. Consumer electronics are subject to higher demands: a television must have a robust user interface and is not allowed to crash or be unresponsive. Expectations are stricter yet for real-time applications (e.g. involving audio and video, or control systems); a television must display at least 50 pictures of a constant quality per second, for example. The essence of quality of service (QoS) is therefore the offering of a predictable system behaviour to the user.

A central question is how to reconcile the dynamic, unpredictable nature of future applications with the requirements for predictable services.

1.3 Implementing future applications on chip

Having identified the characteristics of future applications and their QoS requirements, we now turn to the question of their implementation in SOCs. Moore's law describes the *exponential growth* over time of the number of transistors that can be integrated in an IC. It predicts that chips in 2010 will count over 4 billion transistors, operating in the multi-GHz range [1]. It is this abundance of computational power that has fuelled the convergence of application domains described above. However, Moore's law is only a prediction, and two major obstacles must be solved to make it a reality [2]. First, to use future VLSI technologies, several *deep-submicron problems* must be solved: the increasing disparity between transistor and wire speeds, power delivery and dissipation, and signal integrity. Second, the intrinsic computational power of an IC must not only be used efficiently and effectively, but the time and effort to design a system containing both hardware and software must also remain acceptable. The so-called *design productivity gap* states that the increase in our ability to design SOCs does not match Moore's law. To close the gap, system design methods to implement applications with the latest VLSI technology (including the definition of architectures, mapping applications to architectures, and programming) must harness exponential hardware resource growth in a scalable and modular manner.

The following two examples show that until recently, architectures and design methods at both the deep-submicron and system levels have been often been *global* in nature. This hampers scalability, and in the next section we show that approaches that compose local solutions are becoming popular.

The physical communication between intellectual property blocks (IP) has made use of a mix of local and global wires. When timing constraints of the design must be verified, IPs cannot be checked independently because they are interconnected; correcting a timing violation in one IP may invalidate the timing of another. This process does not necessarily converge to a solution, and is design specific (not re-usable). This global timing closure problem results from a tight (timing) coupling [3].

The second example considers the model of time. Until now, the dominant design style has been synchronous; a global notion of time has been implemented by globally synchronous clocking. Increasing processing variations on a single IC make this style untenable in the future [4]. This will impact clocking regimes, but also the programming model. When using shared memory, which has been the dominant method to communicate between tasks, if tasks are not tightly synchronised (requiring a common, global time frame) then task scheduling may lead to interference and unpredictable behaviour.

1.4 Composing local subsolutions

To avoid the exponential complexity of global methods and solutions, there is increasing interest in subdividing global problems into *local, decoupled* problems, and then composing the local solutions. This requires, foremost, *compositionality* to assemble a global solution from local ones (Figure 4.1). But solutions must also be *scalable* (and likely hierarchical, Figure 4.1(c)) because there will be an exponential number of local solutions. This applies at all

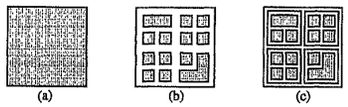


Figure 4.1. Divide global problem (a) into local problems that are composed flat (b) or hierarchically (c). Examples are GALS, local timing closure, local shared memory with global fifo communication or message passing, local busses or switches with global packet switching.

levels of SOC design: lay-out, timing verification, clocking, and programming. This trend commenced some time ago with IP-based re-use, platform-based design [5]. The interest in decoupling and composing local solutions can be observed in hierarchical lay-out and wiring, local clocking strategies such as GALS (globally asynchronous, locally synchronous) [6], software-programmable fixed IPs or tiles [7, 8], and synthesisable tiles for dedicated silicon [9] or FPGAs [10, 11]. At the task level, these scalable system architectures (including e.g. chip multiprocessing [12, 13, 14]) use tasks with local, independent address spaces and time frames that are composed by means of timing-independent fifo channels (for example, Kahn process networks for stream-based processing).

1.5 Networks on chip

All the approaches that advocate local solutions have a common reliance on a scalable and compositional communication medium to efficiently combine the large number of (hardware and software) IPs or subsystems in a working system. This challenge is addressed by networks on chip (NOC), which therefore play a pivotal role in future SOCs. NOCs help to solve both deep-submicron problems (timing closure, wiring, lay-out, etc.) and compositional design methods (services and protocol stacks). More detailed argumentation for the use of NOCs in future SOCs can be found in [15, 9, 16, 17, 2], and elsewhere in this volume.

In the following section we show that NOCs and QoS naturally combine to solve the two problems stated in this section: NOCs combine decoupled local solutions, and a service-based design style makes both the design process and the QoS of the resulting SOC more predictable.

2. NoCs and QoS: a synthesis

We advocate system design centred on NOCs and QoS for three reasons. First, services relieve the inherent tension between the *dynamic nature of future applications* and the user requirement for predictable services. This addresses concerns raised in Sections 1.1 and 1.2. Second, IP re-use and platform-based design aim to re-use applications and architectures by decoupling them. The use of NOCs and their associated network protocol stacks are a way to achieve this (cf. Section 1.5). Finally, it is our tenet that QoS, in particular *guaranteed services*, allow us to move towards *globally predictable, locally predictable* methods and solutions for SOC design. This, in combination with NOCs solves a need of recent architectures and design methods, introduced in Section 1.4. We discuss each point in turn.

2.1 Dynamic applications and predictable QoS

A QoS-based design style alleviates the demanding task for the system *designer*. He or she must plan systems that reliably provide users with the expected QoS for future applications. Moreover, systems must always be cost effective. Higher costs are acceptable when safety is critical, but they are always under pressure for consumer products, whether they be television sets or cars. Systems, especially SOCs, therefore have limited resources, and they must be shared and managed as the application unfolds its dynamic behaviour. Resource management depends on two phases: negotiation to obtain resources, followed by a steady state in which allocated resources are used. As applications become more dynamic, the first phase, *renegotiation*, will become more frequent. Thus, users can be given a predictable QoS by giving resource management and QoS a prominent place in system design.

2.2 Platform-based design, NoCs, and QoS

The aim of *platform*-based design [5, 18] is to reduce the cost of system design through re-use of applications and architectures. A platform decouples applications and system architectures, by defining a template architecture and programming model. In other words, by limiting the freedom in which an application can be implemented on an architecture, the interdependence of application and architecture is concentrated and reduced (Figure 4.2(a)). However, the convergence of applications entails an increasing diversity and dynamics in resource usage (such as communication and computation patterns), and this results in an increasing need for *differentiated services* [15]. It is therefore important that the platform offers the appropriate services (for communication, computation, power management, etc.). As a result, the communication infrastructure is a critical component of a platform, because it must solve the

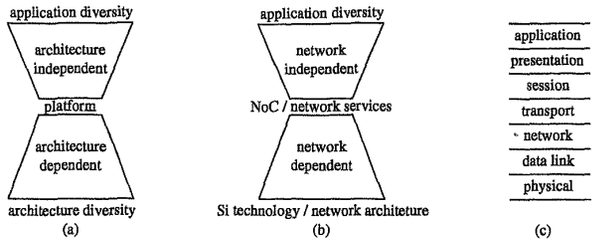


Figure 4.2. Decouple applications and architectures by means of a platform (a), or network (e.g. NOC, Internet) services (b), or network protocol stacks (c).

apparent contradiction of implementing diverse application behaviours with application-dependent IPs in an application-independent manner. Using a NOC for the platform interconnect tackles both problems: it integrates heterogeneous IPs in a standard fashion (in other words, the NOC services largely define the platform, Figure 4.2(a)); and it naturally provides differentiated services by means of a (partially application-dependent) protocol stack [17].

Figure 4.2(b) shows that NOCs offer a small set of services ("NoC services"), on top of which different kinds of communication can be implemented (e.g. shared-memory cache-coherent traffic, message passing). The OSI transport layer [19] is a natural place to separate the application and architecture, because it is the first network-independent layer. The set of services should be small, yet allow the upper layers in the protocol stack to offer differentiated services [20]. This method has been used successfully in computer networks [21], where the internet protocol (IP) plays a similar role. Higher layers offer more specialised protocols such as TCP and UDP, and FTP and HTTP. For NOCs, there are several proposals [22, 23, 24], but convergence has yet to occur.

Concluding, NOCs help IP re-use and platform-based design to combine IPs, and NOC services aid in decoupling applications and architectures.

2.3 QoS enables predictable composition

As we have seen in Section 1.3, to fully exploit Moore's law, it is key to combine and control many local, perhaps autonomous, components in an efficient and flexible manner resulting in the required QoS. To localise timing, communication, data, scheduling, and so on, means that components (subsystems, IPs, tasks, threads, etc.) must first be identified. After composition, their interaction is controlled by means of arbitration, scheduling, or resource management. But we must address the apparent contradiction of providing *globally predictable* QoS (cf. Section 2.1) while current approaches tend to *globally unpredictable, locally predictable* (GULP!) regimes. How can global

timing guarantees be given when locally synchronous components are combined asynchronously [6]? How can global performance be quantified when tasks are combined in a latency-insensitive fashion [14, 7]? It is our tenet that QoS, in particular guaranteed services, allow us to move towards *globally predictable, locally predictable* methods and solutions. We elaborate this claim below, by first defining what role QoS plays, then stating how the local solutions (e.g. IP design) benefit, and finally how QoS eases the composition of local solutions.

2.3.1 Services. Effective steering of the limited resources in a NOC (cf. Section 2.1), and hence effective QoS, is contingent on a reliable reaction of resources to instructions. In other words, a predictable or guaranteed behaviour of system components is a prerequisite to providing the user with expected QoS, at whatever level. The essence of QoS-based system design is to restrict the interaction between components to well-defined services. Two phases can be distinguished: (a) negotiation, followed by (b) resource steering or scheduling, and observation or inspection. As an example, consider a microprocessor which negotiates a connection to a memory with guaranteed bandwidth, but without a constraint on transaction ordering. The NOC that offers communication services will honour the request if has sufficient resources available, and will reject it otherwise. Later, the microprocessor may renegotiate its connection, e.g. to a higher bandwidth. The NOC will release the resources of the connection for use by other connections, when it is closed by the microprocessor.

2.3.2 Advantages for local solutions. QoS-based interaction has a number of advantages for IP design. By limiting component interaction to a set of well-defined services, their interfaces are simplified because there are fewer eventualities to take into account. Moreover, failures of service users (IPs) are concentrated at the reconfiguration points: after a service provider (such as a NOC) has committed to the request (such as a connection with bounded jitter), its provision can be relied upon. Similarly, IPs do not interfere with each other because the service provider and user have a local contract. This allows components to be designed and implemented in isolation. An advantageous side effect of negotiation is that the requirements of IPs must be stated explicitly, aiding the design process.

2.3.3 Advantages in composing local solutions. We now consider the impact of QoS-based interaction when combining IPs to obtain a SOC. First, services that are guaranteed to an IP are not affected by other IPs in the network, making reasoning about the IP in isolation possible. This is essential for a *compositional construction* (design and programming) of SOCs.

Qualcomm Exhibit 1006

Moreover, SOCs can be more *robust*, because rather than relying on cooperation to share resources, resource management enforces the contracts between service providers and users. An IP that (maliciously or erroneously) does not adhere to a cooperative protocol, such as TCP/IP, cannot, therefore, disturb the system as a whole [25]. Failure is therefore local in space (one IP) and time (at one of its reconfiguration points).

Next, when components are combined, the performance (i.e. the QoS) of their composition must be validated. This can be done by analysing the complete system implementation. However, this analysis may be too hard, because the behaviour of individual components or their interactions are complex to model accurately (e.g. traffic analysis of NOCs [26], cache behaviour [27]). *Statistical* approaches are therefore frequently used [28]. Unfortunately resource requests often do not fit models (e.g. bursty traffic versus fractal or normal traffic distributions [26]) invalidating the verification. The oxymoron "statistical guarantee" does therefore not guarantee a QoS, but implies a (usually post hoc) analysis relying on a statistical model of the resources and their usage. Instead, we propose to validate the composition of the local solutions at the level of services (their interface), instead of having to open them up, and consider their (global) combination. This *abstraction* is essential in reducing the verification state space.

Finally, services can make QoS provision *architecture independent*, because how a component, such as a NOC, offers its services is not relevant. This removes the need to second-guess the inner workings of a component, because its services describe all that is required to know. Interaction becomes prescriptive (state what is required, i.e. what must happen), rather than prognostic (try to predict the service provider's behaviour) or reactive (act on how the service provider behaves).

We can therefore conclude that by abstracting local IP behaviours to their service requirements or provision, makes the global QoS of their composition more predictable and hence easier to reason about. This simplifies the design of SOCs.

3. On the cost of guaranteeing QoS

We have seen that a QoS-based approach is required to implement future applications, and to offer a predictable performance to users. In essence, offering a QoS requires a commitment. In this section, we present different levels of commitment, and their effect on predictability and cost in terms of resource usage. Although guaranteed services, which offer commitment, have many advantages over so-called best-effort services, which offer no commitment, we show that their combination is beneficial. The ÆTHEREAL NOC, described in the next section, is an example of that approach.

3.1 Different levels of commitment

As has been explained before, to offer a certain QoS with finite resources, the service provider and user negotiate to arrive at a contract, i.e. a *commitment* by the service provider to honour the request. If a commitment has been given, the service is *guaranteed*, otherwise it is a *best-effort* service. Commitment exists at several levels: 1) *correctness* and integrity of the result, if and when it is delivered. Examples are parity checking and error correcting codes for uncorrupted data transmission, and redundant computation with majority voting for safety-critical systems. 2) Promise of *completion* or delivery. This involves the cumulative availability (over time) of sufficient resources (e.g. memory, cpu cycles), and their performance (e.g. absence of deadlock or livelock). Note that a minimum number of resources may be required: for example, an in-place FFT needs to store all samples simultaneously, and a mobile communication may require a minimum battery charge to generate a sufficiently strong signal. 3) *Bounds* on the performance. Examples are the completion time (when is the result available), cumulative time to completion and its variations (e.g. bounded latency and jitter), and (peak and average) energy consumption.

Almost any form of commitment to progress leads to resource allocation (e.g. memory space, cpu cycles, communication bandwidth, battery power) and hence requires resource management. Moreover, levels of commitment depend on those below them. We illustrate this with two examples.

An IP connected to a NOC may not always be able to accept incoming data. Assuming its input buffers are finite, several solutions are possible, with direct consequences for the service level that can offered. Packets that arrive at a full buffer are dropped, instantly precluding completion bounds such as latency and jitter guarantees. General computer networks typically use this approach. Alternatively, packets are not dropped and are left waiting in the network. Care then has to be taken to not introduce deadlock (e.g. if re-ordered packets that wait for free buffers do not overtake each other), or livelock (e.g. in deflection routing [29] packets keep moving, but may never arrive at their destination). In both cases, again, completion bounds may be in jeopardy. An approach to offer completion bounds (e.g. for a bounded communication latency) is to ensure packets never wait in the network by reserving buffer resources at the receiver, and by using end-to-end flow control to constrain the sender to never send more than the available space.

The second example concerns data transmission in unreliable media. This is a topic of importance in NOCs because wires suffer increasingly from interference, such as cross-talk and voltage drops. To ensure data is transported unchanged, it can be retransmitted when corrupted, or error correction can be used. The energy efficiency of both approaches is investigated by [30] for unreliable busses. What concerns us here, is that using retransmission takes a

Qualcomm Exhibit 1006

variable, possibly unbounded, amount of time, whereas error correction takes place in a constant time. Therefore, time-related guarantees, such as minimum throughput, can only be given by the latter.

We conclude that the strictest guarantees, namely those involving performance bounds, pervade the system (all protocol layers): they cannot be grafted on as an afterthought [17, 31]. In NOCs we should and can choose communication protocols and styles to offer the required services, but, as the examples show, this requires vigilance.

3.2 Commitment and resource usage

The effect of the level of commitment on resource usage is illustrated in Figure 4.3. Suppose that the resource plotted vertically is bandwidth. For

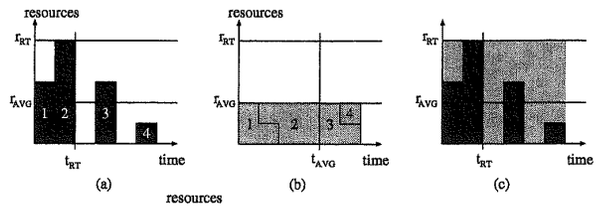


Figure 4.3. (a,b) Commitment to completion or bounded completion is reflected in resource usage. (c) Combining guaranteed services with best-effort services, discussed in Section 3.3.

real-time performance, the requested bandwidth must be offered in the same interval, see Figure 4.3(a). Thus, as many resources must be available as the largest request (r_{RT} at time t_{RT}). With resources dimensioned for the average resource usage (r_{AVG} in Figure 4.3(b)), completion times may shift to the future: the peak resource request at time t_{RT} is only completed at time t_{AVG}, in the example. Depending on the required completion bound and variability in resource requests more or fewer resources can be added. However, more resources than r_{RT} do not improve performance, while fewer resources than r_{AVG} mean that no completion commitment can be given, because the backlog of requests keeps growing.

Architectures offering only best-effort services do not reserve resources, and hence can have a *better average resource utilisation*, at the cost of *unpredictable or unbounded worst-case behaviour*.

3.3 Both best-effort and guaranteed services

A service is guaranteed if a commitment is given, and best effort otherwise. This holds for individual services, not their ensemble. For example, data transport may be uncorrupted (commitment to correctness), and lossless (commitment to delivery), and without throughput guarantees (best-effort throughput, i.e. no commitment to a completion bound). Moreover, a given service can be offered both with and without commitment, to flexibly use the available resources. For example, a data transport service can be offered both with and without completion bounds, to serve different users, in a single SOC. In Figure 4.3(c), the critical dark traffic, uses the guaranteed service, for real-time performance. The remaining $r_{RT} - r_{AVG}$ resources are, on average, available to other traffic, for example with less strict completion bounds, or with no completion bound (best-effort completion).

Figure 4.3(c) shows that when resources must be dimensioned for the worst-case for a given service commitment (e.g. guaranteed latency), the resources can also be used to give less stringent commitments (e.g. guaranteed delivery), or even a best-effort service. A combination of best-effort and guaranteed services gives the advantages of guaranteed services to only part of the system, but the available resources are used more efficiently. In the next section we show how this can be put in practice, in the ÆTHEREAL NOC.

4. The Æthereal network on chip

In this section we introduce the ÆTHEREAL NOC [17, 2]. It offers guaranteed services to obtain the advantages in composability, robustness, and so on of QOS-based design. These services are used for real-time and critical functions. The ÆTHEREAL NOC also provides best-effort services, to take advantage of their lower resource requirements and potentially better average performance. We describe the NOC services [24] in the next section.

In Section 4.2, we show how the services can be efficiently implemented, using a mix of time-division-multiplexed circuit switching, and packet switching [32, 31]. The programming model (connection creation and closing) and its advantages are also explained.

4.1 Æthereal Services

The ÆTHEREAL NOC offers differentiated services with *connections*. A connection describes communication between one master and one or more slaves, with an associated service level, such as fifo transaction ordering, and maximum latency. Connections must be *created* stating the requested service level; this is the negotiation phase. The NOC either accepts or rejects the request for the connection. Connection acceptance may lead to resource reservations in the NOC, e.g. buffers or a link bandwidth percentage. After usage, *closing* a connection frees the resources. Different connections are created and closed independently, possibly at different points in time. Configurations can be computed at compile time (i.e. off-line), or at run time. To ensure that

the programming model scales as NOCs become larger, negotiations can be distributed.

4.1.1 Transactions. Once a connection has been created, the master initiates *transactions* by means of *requests* which zero or more slaves *execute*, perhaps leading to a *response*. Examples of transactions are read, write, acknowledged write, test and set, and flush. By offering these transactions the ÆTHEREAL transaction model is similar to existing bus protocols, to ease migration of IP from current interconnects to NOCs. However, to be able to take full advantage of increased NOC performance, transactions can also be pipelined, split, and posted [24].

4.1.2 Connection Types. The ÆTHEREAL NOC can offer three kinds of connections (Figure 4.4). A *simple* connection contains a master and

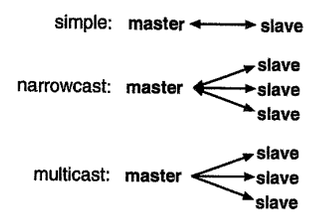


Figure 4.4. Connection types.

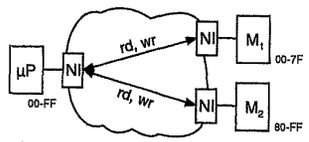


Figure 4.5. A narrowcast connection.

a single slave. The master initiates transaction which the slave executes, possibly resulting in a response (e.g. for a read). On a *narrowcast* connection, containing a master and one or more slaves, the request from the master is sent to and executed by exactly one slave. An example of the narrowcast connection is shown in Figure 4.5, where the master performs transactions on an address space that is mapped on two memory modules. Depending on the transaction address, a transaction is executed on one of the two memories. A *multicast* connection is a connection between one master and one or more slaves, in which requests are duplicated and each slave receives a copy of those requests. Currently, in a multicast connection no return messages are allowed, because of the volume of traffic they may generate (i.e., one response per destination), and the increased complexity in the master (because individual responses from slaves must be merged into a single response).

4.1.3 Connection Properties. We distinguish the following services, or connection properties: 1) data integrity, 2) transaction ordering, 3) transaction completion, 4) connection flow control, and 5) connection

throughput, latency, and jitter. A connection can request any combination of these properties (e.g. a throughput guarantee, flow control, but no transaction ordering). Recalling the levels of commitment, of Section 3.1, properties 1 and 2, above, commit to correctness (including order) of results. Completion is guaranteed by property 3 and 4 (4 implies 3). Connection latency and jitter give completion bounds. We discuss each property in turn.

1) *Data integrity* means that data is transported unchanged. We assume that data integrity is solved at the data link layer; every connection therefore offers data integrity. However, as noted in Section 3.1, this must be done in a way that supports higher-level commitments, in particular bounds on transport completion (latency).

2) *Transaction ordering.* Transaction orders are only defined on a single connection; transactions on different connections may be transported and executed in any order. In general, responses and requests may be reordered during their transport in the network. This means that requests (responses) may arrive in a different order at a single slave (master) than they were sent. (Note that we refer to requests and responses of different transactions. Within a single transaction, requests and responses are always ordered as follows: the master sends a request, a slave receives the request and executes it, the slave sends the response, the master receives the response.) We refer to [24] for a detailed analysis of different orderings. Here it suffices to state that three connections orderings are useful. *Unordered* connections, in which no order is assumed between any request and response. *Locally ordered connections*, where requests sent by the master for each slave are delivered to that slave in order they were sent. Requests for different slaves are unordered. Responses are delivered to the master in the order the requests were generated. For example, on a narrowcast connection with multiple single-port memory slaves, the read and write order to a single memory is important, but not between memories. *Globally ordered connections*, in which requests for (responses from) all slaves are delivered to the slaves (master) in the order the requests were sent. This last ordering is not offered as a service, because local ordering is usually sufficient, and because the user can emulate global ordering by means of acknowledged transactions. Global ordering may be used when the order in which different devices (slaves) are programmed is of importance.

3) *Transaction completion* stipulates that transactions requesting a response (e.g. acknowledged write) guarantee that the master always receives a response. The response contains a) a report that the request was not delivered to the slave, or b) the response of the slave, after it successfully executed the request, c) a notification that the slave successfully executed the request, but the response was dropped, d) a report that the slave failed to execute the request. For connections with flow control, no data will be dropped, and transaction completion is automatic. Without flow control fewer resources are required in

the network, but data may be dropped. For example, it may be fine to drop a transaction, as long as you know it has been lost, so that you can resend it.

4) *Connection flow control* guarantees that data that is sent will fit in the buffers at the receiving end. End-to-end flow control is one of main techniques to avoid network congestion, if data cannot be dropped (cf. Section 3.1). If a slave is slower than its master, and the slave buffers fill up, then the master will be blocked until there is sufficient free space for a transaction. Similarly, a slave may be blocked by a slower master on the return path.

5) *Bounds on connection throughput, latency, and jitter.* The ÆTHEREAL NOC provides connections that guarantee a bandwidth per fixed time interval. Thus, a combined throughput, latency, and jitter guarantee is given. Depending the time interval, the latency and jitter bounds may be rather large. This is acceptable in many applications, such as audio and video streaming, where throughput is more important than latency. In the next section give the reasons for giving the throughput guarantee in this form.

The ÆTHEREAL services include transactions of bus protocols, to ease migration from current bus-based systems to NOCs. But where in OCP [23] and VCI [22] connections are used only to relax transaction ordering, we offer a more general connection property model. Not only are more properties considered, but the request and response communications can be independently configured (e.g. for flow control, and throughput guarantees), allowing more fine-grained resource management. Connection-based service provision therefore allows better differentiation of services, and allows users to make full use of NOC performance.

4.2 Æthereal architecture

The challenge of designing a NOC lies in finding a balance between the NOC services and their implementation complexity and cost. Moreover, as has been mentioned in Section 3.1, different services (levels of commitment) are dependent on each other. Offering time-related guarantees (a completion bound) influences the NOC to the core. The ÆTHEREAL NOC has both guaranteed and best-effort services, and an architectural challenge is how to combine these efficiently. In other words, how can *guaranteed worst-case behaviour be joined with good average resource usage*. In the following sections we describe two conceptually disjoint networks that each solve one part. An efficient combination is then presented. Both networks contain two components: routers and network interfaces. Routers transport data and can be described at the OSI network layer. As the ÆTHEREAL services are offered to IP at the transport layer and are end to end (master to slave and vice versa), network interfaces are required to bridge the network layer and transport layer views on communication.

4.2.1 A guaranteed-throughput router.

To give time-related guarantees on a connection, such as throughput guarantees (on a finite time scale) or latency bounds, the interference of other traffic in the NOC must be limited and characterised. Circuit switching gives strong guarantees but at a high cost: connections have to be dimensioned for the worst case traffic. Time-division multiplexed circuit switching reduces this cost, but creating and closing of circuits still takes much time, and grows with the size of the network. The life-time of circuits must grow to amortise this cost [33]. Rate- and deadline-based scheduling [34, 35] can characterise the worst-case contention and offer bounds on latency, by regulating the inflow of data, but at the a high buffer cost.

The ÆTHEREAL NOC uses *contention-free routing*, which is based on a time-division-multiplexed circuit-switching approach, where one or more circuits are set up for a connection, which is assume to be relatively long-lived. Guaranteed throughput (GT) packets never use the same link at the same time, i.e. all contention is avoided. This can be achieved by controlling both the time GT packets enter the network, and their speed in the network. All routers logically have a common notion of time, embodied in a slot counter (see Figure 4.6). GT packets propagate at the rate of one router per slot counter increment. By regulating the time a GT packet is injected in the network, it is in effect scheduled to use each successive link in its path in a successive slot, see Figure 4.6. This method avoids the emptying and filling of circuits between

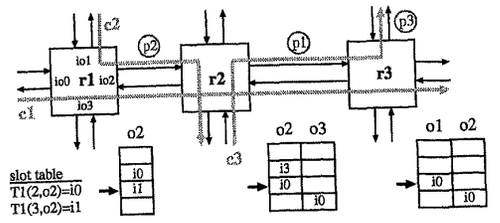


Figure 4.6. Three routers, all in slot $s = 3$, switch packets p_i of circuit c_i to outputs o_i, for $T(s, o_i) = c_i$. For each table, only the relevant columns (outputs) are shown. Input/output link pairs are labelled io_i.

switching of circuits. Conceptually it resembles input queuing with store-and-forward routing, which results in low buffering costs because GT packets are small. The end-to-end (network interface to network interface) latency is the number of hops multiplied by the size of the GT packet. This is minimal, once the packet has entered its slot. This model also allows multicast circuits. GT packets are headerless, and are routed by means of slot tables in every network interface and router. Section 4.2.3 explains how the slot tables are programmed.

Qualcomm Exhibit 1006

4.2.2 A best-effort router. The best-effort router uses packet switching and has a more conventional structure. Our experiments [31] indicate that both input queueing with worm-hole routing or virtual-cut-through routing, and virtual output queueing with worm-hole routing are feasible, in terms of buffering costs. Input queueing uses fewer buffers, but suffers from head-of-line blocking. Virtual output queueing has a higher performance but at the cost of more buffers.

4.2.3 A combined GT-BE router. The logically separate guaranteed (GT) and best-effort (BE) routers are combined (Figure 4.7(a)) to share the router resources (e.g. switch and data path, Figure 4.7(b)), and to obtain the advantages of both. The GT router offers a fixed end-to-end latency for its traffic, which has the highest priority, enforced by the arbiter. The BE router

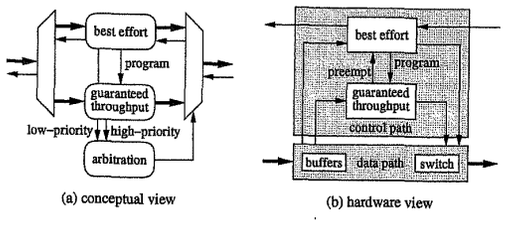


Figure 4.7. Two views of the combined GT-BE router.

uses all the bandwidth (slots) that has not been reserved or is not used by GT traffic. This allows the sharing of links and data path. Resources are therefore never left unused, when there is data, cf. Figure 4.3(b). They are either used for critical traffic with real-time requirements (for which a completion bound has been given), or for best-effort traffic (without a completion bound).

To allow distributed and scalable programming of connections, the GT router slot tables (cf. Figure 4.6), are programmed by means of BE packets, see the arrow "program" in Figure 4.7(a&b). This can be done in a pipelined and concurrent (multiple simultaneous negotiations, also from the same the source), and distributed (active in multiple routers) fashion. Negotiations, resulting in slot allocations, can be done at compile time, and be configured deterministically at run time. Negotiations can also be done at run time, centrally or distributed.

4.2.4 A network interface. ÆTHEREAL network interfaces convert the OSI network layer services of the routers to transport layer services for the user. All connection properties (cf. Section 4.1) that are end-to-end are implemented by the network interfaces. These are: reordering, transaction

completion, and flow control. Unordered connections require that transaction identifiers are used, these are added at the network interfaces. For locally ordered connections, reordering buffers are required in the network interfaces. Since our routers do not reorder traffic in a connection, this is only required for narrowcast connections at the master for responses. Transaction completion also requires resource reservations in the network interfaces. Flow control serves to prevent overflow of buffers at the slave or master network interface. In a credit-based approach, this requires state (credits for the amount of space available), and additional communication (when data is consumed credits are returned to the producer). This is taken care of by the network interface.

IPs negotiate with network interfaces to obtain connections with certain properties. For this network interfaces may reserve resources, such as network interface buffers and credit counters, and slots in router tables.

5. Related work

Differentiated services have received attention in general computer networks in the context of ATM [36] and the Internet [25]. Real-time traffic schemes have been described using rate-based and deadline-based scheduling [34, 35].

The differences between single-hop on-chip communication such as busses and switches and NOCs are described in [24]. Bus protocols such as [22] often have time-division multiplexing and/or priorities added, such as MicroNetwork [37], to give throughput guarantees. To reduce the guaranteed but average high latency, statistical approaches can be used (e.g. Lotterybus [38]). The bound on completion time is then lost (cf. Section 3.1). Switches [39, 40], also single-hop interconnects, can also offer performance guarantees.

On-chip busses with bridges [41, 42, 43], but also [44], are potentially full-blown NOCs. By judiciously placing restrictions on topology, bridging, and transaction models, many problems that arise in general networks can be avoided. For example, an appropriate topology simplifies routing to a single path, and avoids transaction reordering. Limiting a master to a single outstanding transaction has the same effect. By not buffering in bridges, effectively an end-to-end circuit is set up per transaction, avoiding reordering and the need for end-to-end flow control. In both cases, the increasing latency of larger networks make these simplifying assumptions untenable in the future [33, 45].

The octagon interconnect [46] is an interesting combination of packet and circuit switching. An octagon corresponds to a single 8×8 router that uses circuit switching per transaction (or equivalently, packets of unlimited size), with a high performance for a connection in an octagon. An individual octagon can be made larger (the so-called core and edge node strategy), but this does not scale to larger networks for the reasons mentioned above. Packet switching can then be used instead, by using so-called bridge and member nodes strategy.

Qualcomm Exhibit 1006

At this point all general NOC issues appear. This NOC seems a best-effort architecture, at least when multiple octagons are used.

The Spin [28, 29] NOC uses packet switching with worm-hole routing and input queuing in a fat tree topology. It is a scalable network for data transport, but uses a parallel network (bus) for control. It is a best-effort network, and is optimised for average performance (e.g. by the use of optimistic flow control coupled with deflection routing). Commitment is given for packet delivery, but latency bounds are only given statistically.

Dally et al [9] describes a preplaced mesh NOCs, with IPs that are synthesised in the tiles. It uses packet switching on lossless virtual channels with end-to-end flow control. No details are given on the programming model, but guaranteed throughput (and latency) services are supported.

In the context of FPGAs using bit-level circuit switching to implement inter-IP communication is expensive. A first optimisation is to use coarser circuits [47] to reduce the cost. A packet-switched interconnect with a predefined set of services allows sharing of communication resources, just like for ASICs, and enables dynamic reconfiguration [10, 11]. The NOC can be synthesised [10], but a preplaced hard-wired NOC is the next logical step.

New SOC architectures that rely on NOCs include chip multiprocessing [12, 13, 14, 7, 8, 48], to interconnect the homogeneous or heterogeneous tiles, and network processors [49].

6. Conclusions

We observe that, although future applications will be more dynamic, *users* expect embedded systems to behave predictably. A design style based on guaranteed quality of services (QoS) can solve this apparent contradiction. *Designers* of systems on chip (SOC) use networks on chip (NOC) to keep up with Moore's law. NOCs solve both deep-submicron problems (e.g. signal integrity), and narrow the design productivity gap (the efficiency with which we design SOCs) by dividing global problems into local, decoupled problems, e.g. GALS.

The combination of NOCs and QoS is natural, through network protocol stacks, and beneficial for several reasons. It enables IP re-use and platform-based design to decouple applications and architectures. QoS-based design also ensures that when global problems, such as clocking, are solved by combining local, decoupled solutions (e.g. GALS), the combination has a globally predictable behaviour, as required by the user.

There are several levels of QoS *commitment*, building on each other: correctness (e.g. uncorrupted transport), completion (e.g. packet delivery), and completion bounds (e.g. maximum latency). A service without commitment (a *best-effort* service, such as unbounded latency), can have a better average

resource utilisation than a *guaranteed* service, but at the cost of unpredictable or unbounded worst-case behaviour. The *combination of best-effort and guaranteed services is advantageous*: critical parts of the system are predictable, while resources are used more efficiently.

The ÆTHEREAL NOC combines best-effort and guaranteed services at different levels. Its programming model and architecture are also described. The ÆTHEREAL NOC is at the basis of a QoS-based design style, as advocated in this chapter.

References

[1] Semiconductor Industry Association. *The International Technology Roadmap for Semiconductors*. 2001.

[2] Paul Wielage and Kees Goossens. Networks on silicon: Blessing or nightmare? In *Euromicro Symposium On Digital System Design*, Dortmund, Germany, September 2002. Keynote speech.

[3] T. N. Theis. The future of interconnection technology. *IBM journal of research development*, 44(3):379–390, May 2000.

[4] Marcel J. Pelgrom, Hans P. Tuinhout, and Maarten Vertregt. Transistor matching in analog CMOS applications. In *IEDM*, pages 915–918, 1998.

[5] K. Keutzer, S. Malik, A. Richard Newton, Jan M. Rabaey, and A. Sangiovanni-Vincentelli. System-level design: Orthogonalization of concerns and platform-based design. *IEEE Trans. on CAD of Integrated Circuits and Systems*, 19(12):1523–1543, 2000.

[6] Jens Muttersbach, Thomas Villiger, and Wolfgang Fichtner. Practical design of globally-asynchronous locally-synchronous systems. In *6th International Symposium on Advanced Research in Asynchronous Circuits and Systems (ASYNC)*, April 2000.

[7] Paul Stravers and Jan Hoogerbrugge. Homogeneous multiprocessing and the future of silicon design paradigms. In *VLSI-TSA*, 2001.

[8] Ken Mai, Tim Paaske, Nuwan Jayasena, Ron Ho, William J. Dally, and Mark Horowitz. Smart memories: A modular reconfigurable architecture. In *ISCA*, June 2000.

[9] William J. Dally and Brian Towles. Route packets, not wires: On-chip interconnection networks. In *Design Automation Conference*, pages 684–689, June 2001.

[10] Théodore Marescaux, Andrei Bartic, Dideriek Verkest, Serge Vernalde, and Rudy Lauwereins. Interconnection networks enable fine-grain dynamic multitasking on FPGAs. *FPL*, 2002. LNCS 2438.

[11] Edson L. Horta, John W. Lockwood, David E. Taylor, and David Parlour. Dynamic hardware plugins in an FPGA with partial run-time configuration. In *Design Automation Conference*, June 2002.

[12] Lance Hammond, Basam A. Hayfeh, and Kunle Olokotun. A single-chip multiprocessor. *IEEE Computer*, pages 79–85, September 1997.

[13] Jaehyuk Huh, Stephen W. Keckler, and Doug Burger. Exploring the design space of future CMPs. In *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques (PACT)*, 2001.

[14] Eylon Caspi, André DeHon, and John Wawrzynek. A streaming multi-threaded model. In *Third Workshop on Media and Stream Processors (MSP-3)*, December 2001.

[15] M. Sgroi, M. Sheets, A. Mihal, K. Keutzer, S. Malik, J. Rabaey, and A. Sangiovanni-Vincentelli. Addressing the system-on-a-chip interconnect woes through communication-based design. In *Design Automation Conference*, pages 667–672, June 2001.

[16] Luca Benini and Giovanni De Micheli. Networks on chips: A new SoC paradigm. *IEEE Computer*, 35(1):70–80, 2002.

[17] K. Goossens, J. van Meerbergen, A. Peeters, and P. Wielage. Networks on silicon: Combining best-effort and guaranteed services. In *Proceedings of Design Automation and Test Conference in Europe*, pages 423–425, March 2002.

[18] A. Ferrari and A. Sangiovanni-Vincentelli. System design: traditional concepts and new paradigms. In *International Conference on Computer Design*, pages 2–12, 1999.

[19] J. D. Day and H. Zimmerman. The OSI reference model. In *Proceedings of the IEEE*, volume 71, pages 1334–1340, 1983.

[20] K. G. W. Goossens and O. P. Gangwal. The cost of communication protocols and coordination languages in embedded systems. In Farhad Arbab and Carolyn Talcott, editors, *Coordination languages and models*, number 2315 in Lecture notes in computer science, pages 174–190. Springer Verlag, April 2002.

[21] Steve Deering. Watching the waist of the protocol hourglass. In *6th IEEE International Conference on Network Protocols*, October 1998. Keynote speech.

[22] VSI Alliance. Virtual component interface standard, 2000.

[23] OCP International Partnership. Open core protocol specification, 2001.

[24] Andrei Rădulescu and Kees Goossens. Communication services for networks on silicon. In Shuvra Bhattacharyya, Ed Deprettere, and Juergen Teich, editors, *Domain-Specific Processors: Systems, Architectures, Modeling, and Simulation*. Marcel Dekker, December 2002.

[25] Vijay P. Kumar, T. V. Lashman, and Dimitrios Stiliadis. Beyond best effort: Router architectures for the differentiated services of tomorrow's internet. *IEEE Communications Magazine*, pages 152–164, May 1998.

[26] Girish Varatkar. Traffic analysis for on-chip networks design of multimedia applications. In *Design Automation Conference*, June 2002.

[27] Sungjoo Yoo, Kyoungseok Rha, Youngchul Cho, Jinyong Kung, and Kiyoung Choi. Performance estimation of multiple-cache IP-based systems: Case study of an interdependency problem and application of an extended shared memory model. In *International Workshop on Hardware/Software Codesign*, May 2002.

[28] Pierre Guerrier and Alain Greiner. A generic architecture for on-chip packet-switched interconnections. In *Proceedings of Design Automation and Test Conference in Europe*, pages 250–256, 2000.

[29] Pierre Guerrier. *Un Réseau D'Interconnexion pour Systémes Intégrés*. PhD thesis, Université Paris VI, March 2000.

[30] Davide Bertozzi, Luca Benini, and Giovanni De Micheli. Low power error resilient encoding for on-chip data buses. In *Proceedings of Design Automation and Test Conference in Europe*, March 2002.

[31] E. Rijpkema, K. Goossens, A. Rădulescu, J. van Meerbergen, P. Wielage, and E. Waterlander. Trade offs in the design of a router with both guaranteed and best-effort services for networks on chip. In *Proceedings of Design Automation and Test Conference in Europe*, March 2003.

[32] Edwin Rijpkema, Kees Goossens, and Paul Wielage. A router architecture for networks on silicon. In *Proceedings of Progress 2001, 2nd Workshop on Embedded Systems*, Veldhoven, the Netherlands, October 2001.

[33] André DeHon. Robust, high-speed network design for large-scale multiprocessing. A.I. Technical report 1445, Massachusetts Institute of Technology, Artificial Intelligence Laboratory, September 1993.

[34] Hui Zhang. Service disciplines for guaranteed performance service in packet-switching networks. *Proceedings of the IEEE*, 83(10):1374–96, October 1995.

[35] Jennifer Rexford, John Hall, and Kang G. Shin. A router architecture for real-time communication in multicomputer networks. *IEEE Transactions on Computers*, 47(10):1088–1101, October 1998.

[36] ATM Forum. *ATM User-Network Interface Specification*. Prentice Hall, July 1994. Version 3.1.

[37] Drew Wingard. MicroNetworks-based integration for SOCs. In *Design Automation Conference*, 2001.

[38] Kanishka Lahari, Anand Raghunathan, and Ganesh Laskhminarayana. Lotterybus: A new high-performance communication architecture for system-on-chip designs. In *Design Automation Conference*, June 2001.

[39] Jeroen A.J. Leijten, Jef L. van Meerbergen, Adwin H. Timmer, and Jochen A.G. Jess. Stream communication between real-time tasks in a high-performance multiprocessor. In *Proceedings of Design Automation and Test Conference in Europe*, pages 125–131, 1998.

[40] Paul J.M. Havinga. *Mobile Multimedia Systems*. PhD thesis, University of Twente, The Netherlands, February 2000.

[41] Kyeong Keol Ryu, Eung Shin, and Vincent J Mooney. A comparison of five different multiprocessor SoC bus architectures. In *Euromicro*, 2001.

[42] Tycho van Meeuwen, Arnout Vandecappelle, Allert van Zelst, Francky Catthoor, and Diederik Verkest. System-level interconnect architectures exploration for custum memory organizations. In *International Symposium on System Synthesis*, pages 13–18, October 2001.

[43] Milenko Drinic, Darko Kirovski, Seapahn Meguerdichian, and Miodrag Potknojak. Latency-guided on-chip bus network design. In *Proc. of IEEE/ACM International Conference on Computer Aided Design*, pages 420–423, November 2000.

[44] John Bainbridge and Steve Furber. CHAIN: A delay-insensitive chip area interconnect. *IEEE Micro*, 22(5), 2002.

[45] Frederic Chong, Henry Minsky, André deHon, Matthew Becker, Samuel Peretz, Eran Egozy, and Frank F. Knight, Jr. Metro: A router architecture for high-performance, short-haul routing networks. In *International Symposium on Computer Architecture*, April 1994.

[46] Faraydon Karim, Anh Nguyen, Sujit Dey, and Ramesh Rao. On-chip communication architecture for OC-768 network processors. In *Design Automation Conference*, June 2001.

[47] André DeHon. Rent's rule based switching requirements. In *SLIP*, April 2001. Extended abstract.

[48] Shashi Kumar, Axel Jantsch, Juha-Pekka Soininen, Martti Forsell, Mikael Millberg, Johny Öberg, Tiensyrjä, and Ahmed Hemani. A network on chip architecture and design methodology. In *ISVLSI*, 2002.

[49] David Whelihan and Herman Schmit. Memory optimization in single chip network switch fabrics. In *Design Automation Conference*, June 2002.

II

HARDWARE AND BASIC INFRASTRUCTURE

Chapter 5

ON PACKET SWITCHED NETWORKS FOR ON-CHIP COMMUNICATION

Shashi Kumar
Department of Electronics and Computer Engineering, School of Engineering, Jönköping University, Sweden

Abstract:

Designing a system on a chip with large number of cores poses many challenging problems. Designing a flexible on-chip communication network, which can provide the desired bandwidth and can be reused across many applications, is the key problem. In this chapter, we discuss how we can borrow and adapt the concept of packet switched communication from computer networks to design on-chip networks. In this new paradigm, the cores on the chip communicate among themselves by sending packets using a network of switches. We briefly describe the existing proposals, which have come up in the last couple of years. The network is built using a set of identical switches connected in regular topology. The important issues for design of the network are switch design, design of network access by cores and communication protocols. A particular protocol stack, called Nostrum, is described as a case study of protocols for on-chip communication. Ideas and tools can be borrowed from computer networks for evaluating on-chip networks.

Keywords: System on Chip (SoC), interconnection network, communication, protocols, IP cores

A. Jantsch and H. Tenhunen (eds.), Networks on Chip, 85–106.
© 2003 *Kluwer Academic Publishers. Printed in the Netherlands.*

Qualcomm Exhibit 1006

1. INTRODUCTION

In a few years time, it will be possible for designers to integrate more than fifty IP cores, each of the size of a small processor, along with many memory blocks of different types on a single chip. To avoid synchronization and clock skew problems, these multi-core SoCs will operate using Globally Asynchronous Locally Synchronous (GALS) paradigm, where each core will operate in a separate clock domain. The major challenge the SoC researchers face today is to come up with **structured**, **scalable**, **reusable** and **high performance** interconnection architectures/platforms for integrating a large number of heterogeneous cores on a single chip.

An interconnection platform is **structured**, if the underlying interconnection graph has nice properties for its layout and for mathematical analysis. A **scalable** architecture, in this context, implies that there is no inherent technical limitation in the architecture to build system with large number of cores. Reusability in all aspects of system design is key to fast time to design and test of complex and large SoCs. A **reusable** interconnection platform will require that the designer only concentrates on selection and specialization of cores and a new SoC is configured by plugging in the cores in a pre-designed interconnection network. The interconnection architecture must provide facilities for a large number of cores to efficiently exchange information. This implies that the interconnection architecture should allow high communication bandwidth between pairs of cores as well as concurrent communication among many pairs of cores. Many commercially important applications, especially in the area of multi-media communication and processing, require multi-core SoCs with high communication bandwidth among cores. Therefore, **high performance** in this context implies high communication bandwidth among cores. All these properties are related but one property in architecture does not imply all other. In fact, it is difficult to ensure all these desirable properties in an interconnection network simultaneously.

To meet these challenges, many research groups have concurrently proposed the idea of using a packet switched communication network, similar to one used in computer networks, for on chip communication. In this chapter, we discuss the development of this idea and various aspects of building on-chip networks.

The rest of this chapter is organized as follows. In section 2, we discuss the natural evolution of packet switched communication from earlier

methods used for connecting on chip components. In section 3, we define the necessary terms and describe concepts related to packet switched communication. In section 4, we briefly describe some important proposals for on chip communication and summarize the state of the art by providing a brief comparison among them. Section 5, describes in some details a concrete proposal for communication protocols for on-chip communication. Section 6, briefly discuss an attempt for evaluation of packet switched architecture and protocols. Section 7 summarizes the advantages and limitations of packet switched network

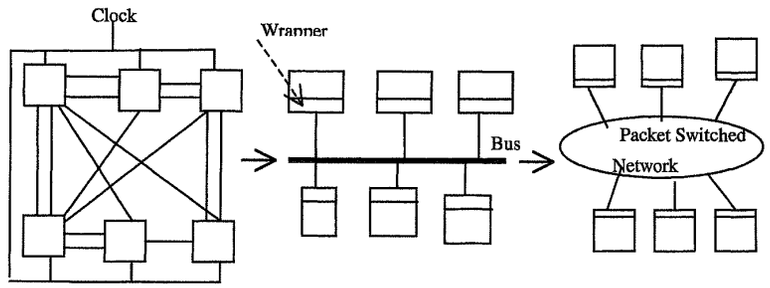
2. EVOLUTION OF ON-CHIP PACKET SWITCHED NETWORKS

The idea of using direct dedicated wires to connect pairs of communicating cores is not even considered, let alone used, since it will lead to large number of pins for every core, large routing time, large routing area, unpredictable delays in signals and unpredictable signal quality. Such a system is not scalable, leads to very low utilization of routing resources and hardly any part of it is reusable. In spite of these limitations, such a scheme to build SoCs has a potential of highest possible performance.

Bus-based interconnection is the most commonly used architecture for inter-core communication in present day SoCs. Buses help to time-share the wires among many communicating cores and lead to reduction of I/O pins in cores and simplifies wiring. Buses are more scalable and can achieve almost the same performance as compared to direct connection architecture. Multiple and hierarchical bus architectures have been proposed for providing efficient and high performance communication among many communicating units. Buses have been used to implement single board computers as well as small multi-processor systems [13]. Unfortunately, buses are also not scalable beyond some limit and may not provide required performance in large systems because the available communication bandwidth is shared among all the units connected to the bus.

Building bus based SoCs has another problem that it require interface designs or wrappers for every core before it can be connected to on-chip buses. The Virtual Socket Interface (VSI) Alliance [9] has tried to remove this problem by proposing a standard, called Virtual Component (VC) interface. Cores and buses need to be wrapped using this standard for easy interfacing. Figure 1 illustrates connection of a core to an on-chip bus. The main focus in VSI Alliance is to enhance the level of reuse from IP cores to SoC integration platform, which also include on-chip communication buses, software and SoC development methodology. This platform based

methodology has been strongly motivated by the success of reusing mother boards and templates for fast development of customized personal computers and application specific systems. Since the underlying interconnection architecture in these platforms is one or multiple buses, they are inherently unscalable in terms of size and performance beyond some limit.



(a)Point to point connection (b) Bus based integration (c) Packet switched network based system integration

Figure1. Evolution of packet switched networks

The computer architecture researchers who have been searching for scalable architectures to build chip multiprocessor [19,20] systems and now they have met the SoC designers at a crucial cross-road (Figure 2). The scalability and success of internet is beckoning them to borrow the idea of using routers (or switches) based networks and packet based communication from computer network design for SoC design and multi-processor systems on chip.

Packet switched communication, besides providing theoretically unlimited scalability also provides possibility of standardization and reuse of communication infrastructure. These features are crucial to chip designers to lower time to design and time to market new products. Recently many researchers have made proposal of packet switched on-chip networks for connecting cores [1-6], [10]. Most of the researchers call the proposals as Network on Chips (NoC). NoC research is still in its infancy and no real SoC has been built using NoC ideas to the best of our knowledge. However, it is expected that future generations of products will have NoC based components.

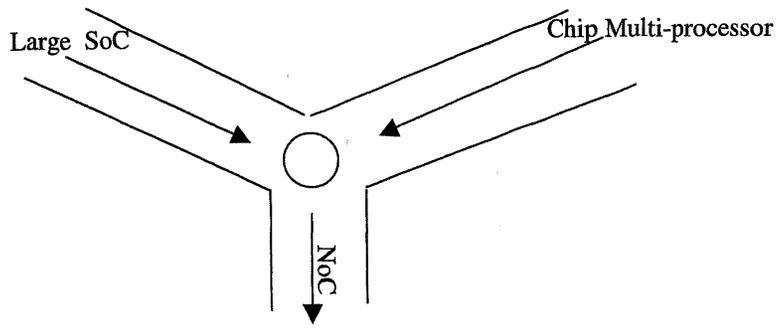


Figure 2: A cross-road for SoC and Chip-Multiprocessor designers

3. PACKET SWITCHED NETWORKS: BASIC CONCEPTS AND COMPONENTS

3.1 Definitions

Most of the terminology used for on-chip packet switched communication is adapted, naturally, from computer network and multi-processor areas.

3.1.1 Switch

The switch is the main component of the interconnection architecture for routing information from a source to a destination. The switch has a functionality of transferring information from one of its input ports to any one/more of its output ports. Figure 3 shows a schematic of a switch. The number of input and output ports is generally small. The time gap between when the information enters the input port to the time when the information leaves the switch through an output port is called switch-delay. Information may need to be buffered before going out through the output port. Figure 3 shows a schematic diagram of a generic switch required to implement a packet switched communication. The switch has buffers to receive incoming packets and buffers to store packets before they are put on the output port. A crossbar switch helps to connect any output of an input buffer to any output buffer. A **switch control** helps to configure the crossbar and management

Qualcomm Exhibit 1006

of the buffers. It may be noted that a specific design of the switch may include both input and output buffers or only one type of buffers. Researchers have also proposed buffer-less switch for packet switched communication [21].

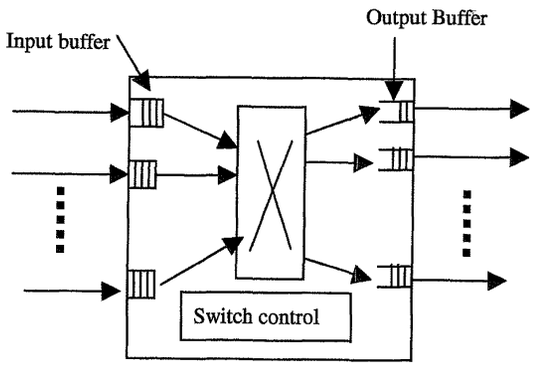


Figure3: A Generic Switch

3.1.2 Switch Network

A switch network consists of an interconnection of many switches to enable a large number of units (cores) to communicate with each other. In this chapter, we will call the communicating cores as resources. A switch network is characterized by its topology, which is basically the underlying graph, in which a switch is represented by a node and a link between two switches is represented by an edge. Various topologies have been used/proposed for building switch networks for multiprocessor systems. Various types of meshes, hypercubes of various dimensions, shuffle network, butterfly network, binary trees and fat trees are a few examples of the topologies used for switching networks for inter-processor communication or communication between processors and memory modules [13]. Figure 4 shows some important topologies. A topology is more suitable for SoC design if it has an area efficient layout on a two dimensional surface. Therefore, mesh and binary tree topologies are more suitable than hypercube and butterfly topologies.

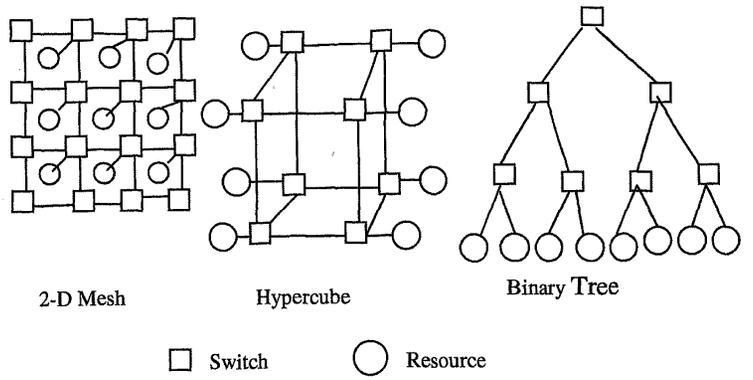


Figure 4: Typical Network Topologies

The inter-related parameters which characterize a switching network and which determine its performance are the network **diameter, connectivity, bandwidth** and **latency** [13]. Network **diameter** is the maximum number of intermediate nodes between a source and any destination pair. **Connectivity** refers to the number of direct neighbors of any switch node in the network. **Bandwidth** is a measure of maximum rate (in bits/second) of information flow in the network. It is possible to define bandwidth for a single link, single node or the aggregate bandwidth of the whole interconnection network. **Latency** is the time taken by a message to travel from the source to the destination.

3.1.3 Connection-oriented and Connectionless Communication systems

Communication is called connection-oriented if a route is established between the source and the destination for the information transmission. In the connectionless case information is encapsulated in entities, which travel through the network from source to the destination. In the traditional circuit switched communication, where a physical path is set up and is used for telecommunication, falls in connection-oriented communication. Packet based communication is possible on both types of systems. If it is possible to set up the network in such a way that the packets belonging to the same message reach the destination in proper sequence, then the communication scheme is called virtual channel or virtual circuit. Virtual circuit is obviously

implementable on connection-oriented systems; it is also implementable on connection less systems.

3.2 Communication Protocols

A set of rules and methods are required for transfer of information from one resource to another resource in any system. These rules are generally referred as protocols. Hardware handshaking with request and acknowledgement signals is a simple example of control signal protocols for communication between two units connected through direct wires. A large number of bus protocols exist which allow reliable communication among many connected units. The bus protocols provide rules for usage of shared communication wires for information exchange and for resolving conflicts among users. Besides specifying the timing relationship among various control and data signals, it also provides upper limits on the physical length and transfer rate or bandwidth [12]. Units can communicate with each other if they have the desired interface to the bus, which implement the bus protocols.

In a packet switched network, communication protocols determine how a resource is connected to the network as well as how the information flows from source to destination. The protocols used for packet switched communication are much more complex than bus protocols, and are generally partitioned and organized in many layers. The architecture defining the protocol layers is referred as a protocol stack. 7-layered OSI model has commonly been used as a standard and as a reference to study various protocol stacks used for communication in computer networks [12]. Most of the NoC researchers have also proposed layered stacks similar to OSI reference model.

3.2.1 Why layered communication?

In a layered protocol, various functions required for communication are divided hierarchically among various layers. Besides helping to manage complexity due to variations in requirements of various heterogeneous resources, it also provides mechanism to separate out various concerns. For example, in the case of a SoC in which many different processors and other cores have been integrated two cores can communicate messages, for example consisting of an array of pixels representing a picture, at a higher layer of protocol stack. At this layer of protocol the two cores need not worry about how the message will be broken up into packets and communicated through the physical network.

3.2.2 Functions of protocol layers in SoC context

The lower three layers of the OSI reference model, namely physical layer, data link layer and network layer, can be used with very little change in their functionality for on-chip packet switched communication. Because of the limited scope of on-chip communication, the top four layers are normally compressed into two layers. Below, we define the functionality of a five layered protocol stack for on-chip communication [2], [16].

- **Physical Layer**: This layer is concerned with physical characteristics of the physical medium used for connecting switches and resources with each other. In the context of SoC, it specifies voltage levels, length and width of wires, signal timings, number of wires connecting two units etc.
- **Data Link Layer**: This layer has the responsibility of reliably transferring information across the physical link. The layer provides functionality of transferring one word of information from one node to a connected node without error. Since the two connected units may be working asynchronously, the data link layer takes care of hardware synchronization besides error detection and correction. Data link layer may also include data encoding or data rate management for controlling power consumption etc.
- **Network Layer**: The network layer provides the service of communicating a packet from one resource to another using the network of switches. Its functionality includes buffering of packets and taking routing decisions in the intermediate switches.
- **Transport Layer**: This layer has the responsibility of establishing end-to-end connection and delivery of messages using the lower layers. Therefore its functionality includes packetization of a message at the source and depacketization or assembly of packets into a message at the destination node.
- **Application Layer**: For on chip communication, the relevant functionality of upper three layers of OSI reference model can be merged into this layer. Important services in this layer include message synchronization and management, conversion of data formats at receiver and application dependent functionality.

3.3 Communication backbone

In the context of on-chip communication, the communication backbone is the communication infrastructure provided for integrating cores for

Qualcomm Exhibit 1006

developing SoCs. The infrastructure consists of the following three components:

- Network of switches
- Three lower layers of communication protocol stack
- Communication services

The communication services corresponding to the lower three levels of protocol stack need to be implemented in every switch of the network. The services corresponding to the upper layers of protocol stack need to be implemented in the resources.

3.4 Network Access

The other important aspects of on-chip communication are the methods required to prepare resources to be integrated to the system. This includes methods to check whether the resource meets the physical, layout, electrical and timing constraints for integration. A wrapper or an interface needs to be added to the resource so that its internal working becomes transparent to the network and other resources in the network. We call this wrapper as Resource Network Interface (RNI).

3.4.1 Resources

The resources in the network could have any functionality. Therefore, a resource could be a standard processor or DSP core, memory of any type, an ASIC block implementing a sequential or parallel behavior, or a reconfigurable hardware block. The obvious requirement is that the resource should be implementable using the same technology as the switch network and its implementation should meet the constraints on metal layers and voltage levels. Different NoC architectural proposals may add a few more constraints on its physical size and layout and its clock frequency.

3.4.2 Resource Network Interface

As mentioned above, every resource requires a wrapper for interfacing to a given packet switched network. Figure 6 shows a generic resource network interface (RNI). This interface has the same purpose as a network card in a computer, which must be connected to the internet. RNI must implement at least the lower three layers of protocol stack. This means that RNI enables a resource to communicate with the network using packets. The internal design of an RNI can be partitioned into two parts. The part connected to

the network is resource independent and the other part connected is resource dependent. The resource independent part can be reused for interfacing any new resource. It depends on its design that is on the three lower level protocol layers implemented by the switch. In an ideal situation, a switch should not need to distinguish an RNI from other switches to which it is connected in the network.

The resource dependent part will depend on the signals available on I/O pins, like data and address bus widths, control signals etc. If a set of cores follows the same standard, like having standard I/O buses and control signals, then the same RNI can be used for all the resources in this set. The major functionality of resource specific part of RNI includes:

- Packetization and depacketization of information.
- Data encoding for error detection and correction

The resource independent part takes care of timing, buffering and synchronization aspects during data communication.

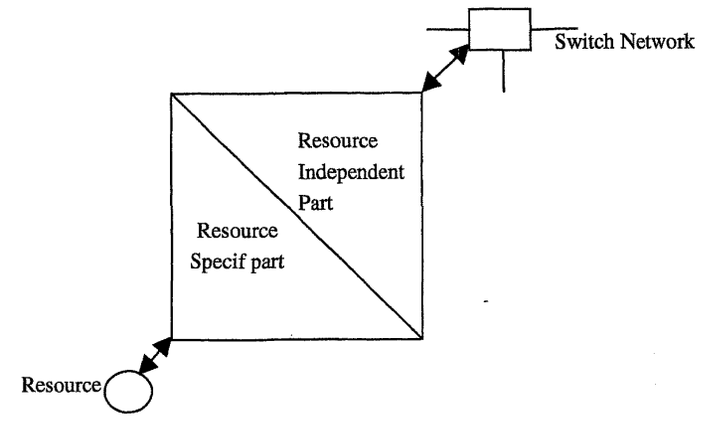


Figure 5: Resource Network Interface

4. EXISTING PROPOSALS

In this section we describe some proposals for building packet switched communication for on-chip communication. As the area of NoC is in its infancy, our aim is to illustrate the direction and the state of thinking in this area rather than completeness. There are four aspects of the packet switched network that are important. These are: the link, topology of the network, the packet structure and the communication protocol stack.

4.1 Scalable, Programmable, Integrated Network (SPIN)

Guerrier and Greiner have proposed a generic architecture for on-chip packet switched communication, called SPIN[6]. Their aim was to demonstrate that this idea is already feasible in the existing technology and can soon be adapted for building SoCs. They propose two one directional 32-bit data paths between two switches in the network. They also propose a Fat-tree topology for the switch network because of its lower diameter as compared to a mesh, and because of its efficient VLSI layout. A packet in SPIN consists of a sequence of variable number of 32-bit words, with the first word acting as a header and following words containing the payload. The packet ends with a special word, which includes checksum. It allows datagram approach implying that the packets belonging to the same message can follow different route in the network.

The protocol stack used in SPIN implements the message passing communication model. The protocol stack is divided in layers such that the lowest layer has functionality similar to the three lower layers of OSI reference model. The lowest layer is fully hardwired and is implemented in wrappers and switches. Other layers can be built over this layer to support models for data flow streams and address space [6].

They have demonstrated the implementation feasibility of building a router in 0.13 μ process, which can be used to connect 128 units (resources) and will provide a peak bandwidth of 1.82 Tbits/sec. The SPIN research group is actively engaged in solving the problem of higher latency in handling memory access using packet switched network.

4.2 On-Chip Micro-Networks [1]

Benini and De Micheli [1] in their conceptual paper on NoC, predict that packet switched micro-networks will be essential to handle complexity of SoC designs of the future. They classify the protocol stack layers for micro-networks in three parts. The lowest part of the protocol deals with issues related to physical wiring. The second part deals with the architectural and logical design of the micro-network and will correspond to data link, network and transport layers. The highest part of the protocol stack layers will take care of application and system issues. The researchers rightly conclude that the major effort in the micro-network design will go in specializing the protocols to suit the application and its performance requirement. No concrete details are specified for the network, packet or protocols.

4.3 Network on Silicon [5]

Philips Research Laboratory has proposed a communication network, called Network on Silicon, for integrating IP blocks and memory modules on the same chip which provides predictable and guaranteed performance [5]. The research group in Philips realized that any proposed communication network should be flexible and efficient. The network must support guaranteed-throughput (GT) for real time applications and best-effort (BT) communication where throughput is not guaranteed, but no data is lost. They propose a connection oriented crossbar based routers to build the network for GT routing. The data rate of the link is assumed to be same as that of the routers. Every router maintains a routing table for crossbar configuration. Every entry in the table, called slot, corresponds to an input to output connection.

For GT communication pairs, slots are reserved in the routing tables of the required routers. The un-reserved slots are used for BE traffic using packet switched communication. Therefore, it is possible to provide high performance for a few communicating pairs in a flexible general architecture.

4.4 Linköping SoCBUS [23]

The SoCBUS tries to add the benefits of circuit switched communication to packet switched communication. They define the concept of Packet Connected Circuit (PCC). The idea is similar to a virtual circuit. A path is initially searched and set up between the communicating IPs using datagram based packet switching and then the path is locked. After that the path works in a circuit switched mode for the rest of the communication period. Such a scheme can lead to high throughput, short latency and deterministic communication behavior. But such a scheme may lead to restricted routing flexibility for the rest of the communication traffic. A demonstrator chip is planned to be built to show the effectiveness of the scheme.

4.5 KTH-VTT Network on Chip

Royal Institute of Technology (KTH), Stockholm and VTT Electronics, Oulu have jointly developed a NoC platform [10] for developing large and complex systems on a chip. This platform consists of a two dimensional mesh of switches. Resources are placed in the slots formed by the switches. A direct 2-D layout of switches and resources is assumed providing physical-architectural design integration. Each switch is connected to one resource and four neighboring switches, and each resource is connected to

Qualcomm Exhibit 1006

one switch. A resource could be any computing or memory core, or it could be a configurable or fixed hardware block as long as it fits physically into the slot. The communication network supports packet switched communication using both datagram as well as virtual path based approach.

In order to reduce the packet delay in the network and reduce the hardware cost of the switch, researchers have designed a buffer-less switch with a specialized efficient dynamic routing algorithm. The algorithm uses the awareness about local traffic around the switch to take a good routing decision [21]. A five-layered protocol stack, called Nostrum, is proposed to implement layered communication. The switch in the network implements the lower three layers of protocol stack. We will discuss more details of the protocol stack in the next section.

4.6 Discussion

All the proposals discussed above have the common thread that they support packet based communication, are scalable and are capable of handling heterogeneity in cores. Various proposals differ in network topologies. Network on Silicon [5] and SoCBus [23] provide special mechanisms in the network to ensure that it is possible to set up high performance communication paths between, at least, a few pairs of resources in the network. These proposals make packet switched communication suitable for data intensive real-time applications. KTH-VTT [10] emphasizes the importance of physical level-architectural level design integration. Although all resources in their proposal must fit physically in a fixed size slot, they propose the concept of a **Region** to accommodate larger resources like shared memories [10]. All proposals assume a processor type resource has direct connection to access its program or data memory. It will be too slow to access such memories through packet switched network. However, a large memory shared among many resources will be treated as a separate resource, which is accessed, by other resources using general packet switched network.

At this stage nobody has reported a comparison of these proposal regarding cost or performance.

5. NOSTRUM: A CASE STUDY OF A PROTOCOL STACK FOR NOC

5.1 Protocol Layers

In this section, we discuss some details of the protocol stack proposed for on chip communication network proposed for KTH-VTT [17]. The ideas proposed in Nostrum are directly applicable, or easily adaptable, to other on-chip communication networks. Like the OSI reference model, the communication in Nostrum is layered with different responsibility for different layers. A clear separation of functionality of the layers in the protocol stack allows independent development and reuse of resources, on-chip network and applications.

Nostrum has three compulsory layers, namely, physical, data link and network layers. A transport layer is also defined to establish and maintain communication between source and destination resources. These lower three layers put together provide the service of delivering a packet of data from one resource to any other resource in the network. Nostrum supports both datagram as well as virtual circuit approach of packet delivery. Below we define the functionality of the lower four layers [17] for on chip communication.

Physical Layer
The physical layer models the physical link between the switches and provides the service of moving a word of data from output of one switch to the input of the connected switch. The signal degration, data error are modeled by this layer.

Data Link Layer
The data link layer is responsible for reliable communication of data through a link with the specified synchronization, error detection/correction, and flow control.

Network Layer
The network layer is responsible for delivery of packets from the source resource through the network of switches to the destination resource. Since in a datagram approach, different packets belonging to a message can follow different routes and can reach out of order, the network layer provides possibility of dynamic routing algorithm as well as buffering of packets in the switches. For virtual circuit approach, the network layer sets up and

Qualcomm Exhibit 1006

maintains a virtual circuit between the sender and receiver for the desired period and with desired bandwidth guarantee.

Transport Layer

The transport layer handles establishment of communication path, packetization and de-packetization of messages. Packetization involves breaking the message into small parts and appending identification of the message, address of the destination process (resource) and sequence number of the packet to the actual payload.

5.2 Data Format at various layers

Figure 6 describes suggested formats at various protocol levels in Nostrum.

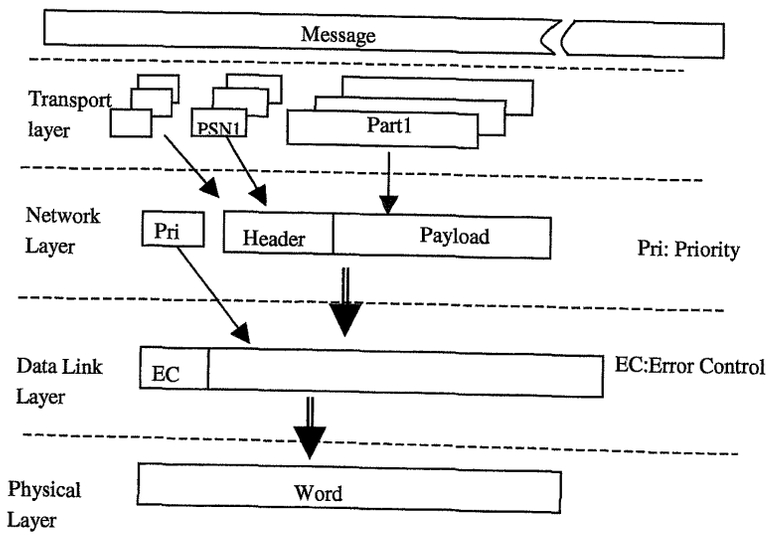


Figure 6: The Nostrum Protocol Stack

A priority index (Pri) is maintained/appended by the network layer to give different priority to different packets when using a datagram approach. One of the possible approaches could be to give a higher priority to packets, which have already spent more time in the network. The other approach could be based on the current distance of the packet from its destination.

One could give higher priority to packets closer/farther from their destination. Some more bits could be appended to the packet by data link layer for error detection and correction.

Physical issues to implement a KTH-VTT NoC has been investigated [22] and it is shown that it is possible to build a 10 X 10 switch network on a chip with a slot of 4 mm^2 for each resource. It is also shown that it is possible to route more than 100 wires between a resource and a switch and between two switches in the mesh network. This implies that it is possible to have a packet with 64-bit payload. Since the wire lengths between switches are going to be of deterministic lengths, signal quality can be ensured and the probability of error will be quite small. To minimize error detection and correction logic, it may be sufficient to provide error detection to only critical bits in the word. For most of the applications, an error in the header part is more dangerous than the error in the payload. Therefore, unequal error protection is recommended at the data link layer.

5.3 Protocol space for on chip communication

For the designer of on-chip networks, there is a large protocol space for selection. The available choice provides a trade-off between performance and flexibility. On one extreme is the possibility of using protocols, which support only circuit switched networks for connecting cores. Such protocols can provide high performance but little flexibility. On the other hand very general protocols used for computer communication can provide a lot of flexibility but very low performance. NoC protocols will provide a trade-off between flexibility and performance. Because of the small physical dimensions of the on-chip network it will be possible to assume smaller error rates and higher communication link bandwidths.

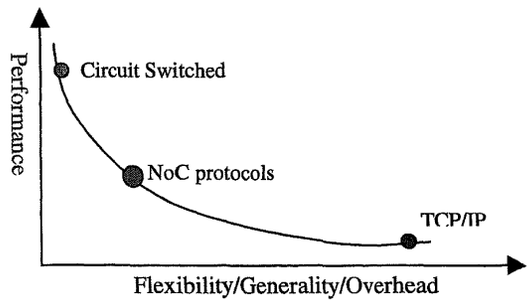


Figure 7: Space of protocols for on chip communication

5.3.1 On Chip communication Vs. Internet Communication

It is important to realize that the geometrical and electrical conditions are very different when dealing with on chip communication as compared to devices connected by cables to Internet. It will be possible to have much higher bandwidth for the on-chip communication as compared to external networks. It is possible to route many parallel wires between resources and the network as compared to serial links for connecting a computer to the Internet. The traffic in the on chip network is also likely to quite different. For example, a good mapping of the application to a NoC will lead to more traffic among resources, which are physically close on the chip. A NoC running a set of applications is also likely to have much more deterministic and periodic traffic than the Internet.

The major difference in the two networks is going to be in the message sizes and variation in the message sizes. It is expected that the message size for on-chip communication is going to vary between 1 byte to a few thousand bytes, with an average being a number much smaller than 100. Therefore, choosing a fixed packet size with a payload of eight will be efficient as compared to a variable packet size.

6. EVALUATION OF ON-CHIP NETWORKS

Designing a network for on-chip communication is similar to the problem of designing a local area computer network with some computing and communication requirements. Therefore, it is necessary to evaluate the performance of these networks to make sure that they provide required performance for intended applications. The purpose of evaluation could be either to decide parameters of the network for an application, or to estimate the performance of the network for a set of applications having certain on-chip communication traffic. In the absence of availability of actual network and real applications which are going to be run on NoC systems, the only choice available for the evaluation is to use models of networks and applications. Network and applications can be modeled using a set of parameters relevant to the evaluation task.

6.1 Evaluation parameters

The parameters could be divided into three categories, namely, architectural parameters, application parameters and performance parameters.

6.1.1 Architectural Parameters

These parameters include topology and size of the network. Packet buffer size used in every switch of the network will be an important parameter. It also includes various parameters used at various layers in the protocol stack used for communication. Important parameters in this category include the routing algorithm, packet priority scheme at network layer level and error control at data link layer level. Packet formats at various levels will also fall in this category.

6.1.2 Application Parameters

For the evaluation of the network, the relevant feature of application can be modeled by the communication behavior of various resources running the application. The communication requirement of various resources will determine the traffic in the network. Therefore, the following parameters are useful for specifying traffic.

- **Rate of packet generation**: The packet could be generated at random intervals or could be generated periodically. The packets generation could be bursty or have uniform time gaps.
- **Destination address of packets**: The destination address of a packet originating from a resource may have a fixed set of destinations, random destination or may have bias towards neighboring resources.

Another important aspect to be modeled for an application, especially real time applications, will be the requirement of bandwidth between the communicating resources. This could be modeled through by an upper limit on packet delay, or average bandwidth between a source and a destination. This quality of service constraints will help the designer to choose between the datagram approach and virtual circuit approach for communication. They also provide the basis for assigning different priorities to different messages.

6.1.3 Performance Parameters

As discussed earlier, the purpose of network evaluation could be for finding the best architecture for an application or a set of applications, or for checking the suitability of a specific on-chip network for an application. Therefore, the chosen performance parameters should help in the above task. The following are a set of generic parameters. There could be other parameters, which could be network specific or application specific.

- **Packet delay**: This refers to the time it takes for a packet to go from source to destination. Depending on the context, average packet delay or maximum packet delay may be important.
- **Packet drop probability**: It defines the probability of a packet being lost in the network due to heavy traffic and limited buffer capacity in the switches. Or this could be due to defects/errors alt various layers in the network.

The packet delay and drop probability will depend on the network load and buffer sizes in the switches. These will also depend on the routing algorithm and priority schemes used.

6.1.4 NoC Simulators

Researchers have borrowed ideas and tools used for evaluation of general computer networks for evaluation of on-chip networks. Yi-Ran[11] has used network simulator **ns-2**, developed at University of Berkeley, for evaluation of KTH-VTT NoC proposal. **ns-2** has been used for research in the area of networks of various types, including wireless networks, fiber optics networks and parallel computers. **ns-2** simulator provides facilities to model network with any topology and any link characteristics. It also provides facilities to define communication protocols and network traffic. It helps to measure useful performance parameters required for the network design [17]. Yi-Ran[11] used **ns-2** to study effect of buffer size and network load on packet delay and packet drop probability in a network with mesh topology of 5 X 5.

Researchers are also building dedicated simulators from scratch for study of NoC architecture and on-chip packet switched communication [18].

7. CONCLUSIONS

Scalability and reusability are the two key features, which are critical for any communication platform for on-chip communication. Packet switched networks proposals are promising to meet these demands. However, the following limitations of packet switched networks must be solved before they will be accepted to be used for implementing useful SoCs.

- Memory Access: Memory accesses will be very inefficient using packet switched network between computing cores and memory cores. The problem can either be reduced by adding local memory to resources. But many data intensive real time systems, especially in the area of image processing, require memory buffers to be shared among many processor cores. We will need hybrid communication infrastructure, which can provide both circuit switched access to such memories and allow packet switched communication for general message passing among cores.
- The second issue is the standardization of protocol stack for on-chip communication. Without standardization it will be difficult to raise the level of reusability of the communication backbone and motivate new users.
- Like any new platform, design and development methodology and tools will be necessary before designers, who are not expert in network concepts and programming, accept the new paradigm.

REFERENCES

1. Luca Benini and Giovanni De Micheli, " Network on Chips: A New SoC Paradigm", IEEE Computer, Jan. 2002, pages 70-78.
2. M. Sgroi et. al., "Addressing the System-on-a-chip Interconnect woes through communication based design", Proc. of the 38th Design Automation Conference, June 2001.
3. D. Wingard, "MicroNetwork based integration for SoCs", Proc. of the 38th Design Automation Conference, June 2001.
4. William J. Dally and Brian Towles, " Route Packets, Not Wires: On-Chip interconnection Networks", Proc. of the 38th Design Automation Conference, June 2001.
5. Edwin Rijpkema, Kees Goossens, and Paul Wielage, " A Router Architecture for Network on Silicon", Proceedings of Progress 2001, 2nd Workshop on Embedded Systems.
6. Pierre Guerrier and Alain Greiner, " A generic architecture for on-chip packet switched interconnections", Proc. of DATE 2000, March 2000.
7. Henry Chang, Larry Cooke, Merrill Hunt, Grant Martin, Andrew Mcnelly and Lee Todd, **Surviving the SOC Revolution: A guide to platform-Based Design**, Kluwer Academic Publishers, 1999.
8. Rochit Rajsuman, **System-on-a Chip: Design and Test**, Artech House, 2002, ISBN 1-5803-107-5.

9. Virtual Socket Interface Alliance, "On-Chip Bus Attributes" and Virtual Component Interface-Draft Specification", http: // www.vsia.com
10. Shashi Kumar et. al. , "A Network on Chip Architecture and Design Methodology", Proc. Of IEEE Annual Symposium on VLSI, 2002, Pittsburgh, USA, pp. 117-124.
11. Yiran Sun, Shashi Kumar and Axel Jantsch, " Simulation and Evaluation for a Network on Chip Architecture using NS-2", Proc. NORCHIP 2002, Copenhagen.
12. James Irvine and David Harle, **Data Communications and Networks: An Engineering Approach**, John Wiley & Sons, 2002, ISBN 0471 80872 5.
13. David E. Culler, Jaswinder Pal Singh (with Anoop Gupta), **Parallel Computer Architecture: A Hardware/Software Approach**, Morgan Kaufmann Publishers, Inc., 1999, ISBN 1-55860-343-3.
14. Kai Hwang and Zhiwei Xu, **Scalable Parallel Computing: Technology, Architectures, Programming**, WCB/ McGraw-Hill Publishers, 1998, ISBN 0-07-031798-4.
15. Ken Mai et. al, "Smart Memories: A modular Reconfigurable Architecture", Proc. of the 27th Annual International Symposium on Computer Architecture, June10-14, 2000 Vancouver, British Columbia, Canada.
16. Mikael Millberg et. al," The Nostrum Backbone: A communication protocol stack for Networks on Chip", Technical Report, LECS/IMIT/KTH, Royal Institute of Technology, Stockholm 2002.
17. The Network Simulator (ns-2) home page: http://www.isi.edu/nsnam/ns/
18. Rikard Thid, " A time driven network on chip simulator implemented in C++", Master thesis, Laboratory of Electronics and Computer Systems, Department of Microelectronics and Information Technology, Royal Institute of Technology, Stockholm, July 2002.
19. Lance Hammond et. al.," A single-chip multiprocessor", IEEE Computer, Sept. 1997.
20. Lance Hammond et. al." The Stanford Hydra Chip Multiprocessor", IEEE MICRO-Magzine, March-April 2002.
21. Erland Nilsson, " Design and Analysis of a hot potato switch in Network on Chip", Master thesis, Laboratory of Electronics and Computer Systems, Department of Microelectronics and Information Technology, Royal Institute of Technology, Stockholm, July 2002.
22. Johnny et. Al, " A feasibility study on the performance and power distribution on two Network on Chip architectures", submitted for publication to IEEE Trans. On VLSI.
23. Dale Liu et. al., " SoCBUS: The solution of high communication bandwidth on chip and short TTM", invited paper in Real Time and Embedded Computing conference, Sept. 2002, Göteborg.

Chapter 6

ENERGY-RELIABILITY TRADE-OFF FOR NoCs

Davide Bertozzi
Luca Benini
Dipartimento Elettronica Informatica Sistemistica
University of Bologna (Italy)
{dbertozzi,lbenini}@deis.unibo.it

Giovanni De Micheli
Computer System Laboratory
Stanford University (USA)
nanni@stanford.edu

Abstract Solutions for combined energy minimization and communication reliability control have to be developed for SoC networks. Redundant encodings and error-resilient protocols create new degrees of freedom for trading off energy against realiability and viceversa. In this chapter, the theoretical framework for energy and realiability analysis is introduced and several error control and recovery strategies are investigated in a realistic SoC setting. Furthermore, the chapter provides guidelines and methods to select the most appropriate error control scheme for a given reliability and/or energy efficiency constraint.

Keywords: communication energy, noise, reliability, error-correcting codes, error-detecting codes

A. Jantsch and H. Tenhunen (eds.), Networks on Chip, 107–129.
© 2003 *Kluwer Academic Publishers. Printed in the Netherlands.*

1. Introduction

As the integration of a large number of IP blocks on the same silicon die is becoming technically feasible, the design paradigm for future Systems-on-Chip (SoC) is shifting from device-centric to interconnect-centric. Performance and energy consumption will be increasingly determined by the communication architecture. Under very high IP integration densities, on-chip realization of interconnection networks (*Networks-on-Chip, NoC*) is emerging as the most efficient solution for communication. NoCs can be viewed as an adaptation of the wide-area network paradigm, well known to the communication community, to the deep sub-micron (DSM) ICs scenario. In this context, micronetworks of interconnects can take advantage of local proximity and of a lower degree of non-determinism, but have to meet new distinctive requirements such as design-time specialization and energy constraints [1].

Energy dissipation is a critical NoC design constraint, particularly in the context of battery-operated devices, and will be the focus of this chapter. The SIA roadmap projects that power consumption can marginally scale up while moving from 100nm to 50nm technology. At the same time, projected clock frequency and number of devices on-chip are increased significantly. These trends translate directly into much tighter power budgets. Voltage scaling, as predicted by the roadmap, is helpful in reducing power. Nevertheless, voltage scaling alone will not suffice, and specific design choices for low-energy consumption will be required.

Computation and storage energy greatly benefits from device scaling (smaller gates, smaller memory cells), but the energy for global communication does not scale down. On the contrary, projections based on current delay optimization techniques for global wires show that on-chip communication will require increasingly higher energy consumption. Hence, communication-related energy minimization will be a growing concern in future technologies, and will create novel challenges that have not yet been addressed by traditional high-performance network designers.

An effective approach to high-speed energy-efficient communication is low swing signaling [2]. Even though it requires the design of receivers with good adaptation to line impedance and high sensitivity (often achieved by means of simplified sense-amplifiers), power savings in the order of 10× have been estimated with reduced interconnect swings of a few hundreds of mV in a 0.18um process [3].

As technology scales toward DSM, the energy efficiency concern cannot be tackled without considering the impact on communication reliability. These two competitive issues are brought up by the scaling scenario and their interaction can be briefly summarized as follows:

- Low swing signaling reduces signal-to-noise ratio, thus making interconnects inherently sensitive to on-chip noise sources such as cross-talk, power supply noise, electromagnetic interferences, soft errors, etc. This sensitivity is increased by the reduction of receiving gates voltage noise margins as an effect of the decreased supply voltages.

- Coupling capacitance between adjacent wires is becoming the dominant component of interconnect capacitance. This has an impact not only on signal integrity, but also on power consumption, as most power consumed by interconnects is associated with switching of coupling capacitances (*coupling power*).

As a consequence, solutions for combined energy minimization and communication reliability control have to be developed for NoCs. The energy-reliability trade-off can be efficiently tackled by means of redundant encoding. Given a predefined reliability constraint for on-chip communication, different coding schemes, with their own error recovery strategies, can be compared in their ability to meet the requirements with the minimum energy cost.

This chapter introduces the theoretical framework for energy and reliability analysis and compares several error control and recovery strategies in a realistic SoC setting. Furthermore, the chapter provides guidelines to select the most appropriate error control scheme for a given reliability and/or energy efficiency constraint.

2. Communication reliability

The critical challenge for NoC design is to provide adequate *quality of service (QoS)* with a limited energy budget under strong technology limitations. QoS requirements include, but are not limited to, communication reliability and performance.

With present technologies, most chips are designed under the assumption that electrical waveforms can always carry correct information on chip. As we move to consider DSM NoCs, communication is likely to become inherently unreliable because of the increased sensitivity of interconnects to on-chip noise sources. The most relevant ones are hereafter briefly described.

1 CROSSTALK

Sidewall coupling capacitance is becoming the dominant component of interconnect capacitance. The relative ratio between cross-coupling capacitance and total wire capacitance already amounts to 70%, but is expected to achieve 80% in a 50 nm process.

Projections show that for semiglobal interconnects the wire critical length at which the peak crosstalk voltage is 10% of the supply voltage decreases almost an order of magnitude by 2014 (50 nm process), which will drastically increase the number of interconnects with significant crosstalk. Local interconnects suffer from the same problem, as they are scaling more in the horizontal dimension rather than in the vertical one [4], thus increasing the sidewall coupling area between adjacent wires.

Crosstalk noise in real buses is usually tackled at the physical layer by means of shielding: a grounded wire is inserted between every signal wire on the bus. The effectiveness of this technique is counterbalanced by the doubling of the wiring area, that might be impractical when routing resources are scarce.

2 POWER SUPPLY NOISE

A large number of "simultaneous" switching events in a circuit within a short period of time can cause a current-resistance (IR) drop on the voltage references [6].

A further worsening of the reference voltages is due to the chip and package inductances, responsible for the Ldi/dt noise contribution, which becomes relevant as an effect of the higher frequencies allowed by the scaled transitor sizes.

SPICE simulations show that a 10-15% voltage drop during a cell transition period can increase cell propagation delay by 20-30% [7]. Interconnects propagation delay can be affected as well because of the temporarily reduced driving capability of wire capacitances.

Common solutions to relieve the power supply noise problem include topology optimization [9], wire sizing [10], on-chip voltage regulation [11] and decoupling capacitance deployment [8, 12].

3 ELECTROMAGNETIC INTERFERENCE (EMI)

The drastically increased number of simultaneously switching transistors per die, combined with faster edges due to increasing clock rates, make EMI a serious concern. External electric and magnetic fields can be coupled into circuit nodes and corrupt signal integrity.

Many design techniques have been developed to meet EMC demands for VLSI circuits, including RC low pass filtering, scalability and temperature compensation for pad drivers [14].

4 INTERSYMBOL INTERFERENCE (ISI)

The smaller cross sections of interconnects in DSM technologies result in a big increase in resistive parasitics. The consequent RC behaviour leads to signal dispersion: if a transmitted pulse across the wire spreads out into other time slots, it causes ISI (the result of memory in the channel which causes consequent symbols to overlap). This represents a fundamental limitation on the bandwidth of DSM interconnects [5].

A code-based approach has been proposed as a workaround for ISI-related limitations, wherein streams are encoded in such a way that isolated bits (i.e. bits taking value b in time slot n while bits in time slots $n-1$ and $n+1$ take the complementary value $\bar{b}$) are not transmitted [15].

5 OTHER NOISE SOURCES

Logic values of chip internal nodes can be flipped as an effect of charge injection due to alpha particles (emitted by packaging materials) or to thermal neutrons from cosmic ray showers. This results in a *soft error*, that used to be relevant only for large DRAMs, but is now conjectured to be a potential hazard for large SoCs as well [13].

Other noise sources are generally handled by means of statistic models: namely, crosstalk between perpendicular wires at adjacent levels (*interlevel*), *thermal noise*, *shot noise* (caused by the quantization of current to individual charge carriers), and *1/f noise* (originated by random variations in components giving rise to a noise that has equal power per decade of frequency). For most cases of practical interest, these noise sources can be modelled as gaussian white noise.

Finally, synchronization problems are worth mentioning. As clock frequencies increase, the time difference between events decreases, and this may lead to synchronization failures. For instance, sampling changing data signals traveling across interconnects has to deal with the sampling uncertainty issue, combined with non-deterministic delays that may affect signal propagation. This might result in sampling errors or *metastability*. For our purposes, synchronization errors can be seen as an additional noise source, to be reckoned with when designing on-chip communication architectures.

3. Traditional approach to fault tolerance

The self-checking property of VLSI circuits can be defined as the ability to automatically verify whether there are faults in logic, without the need for externally applied test stimuli. Whenever this feature is available, on-line error detection is possible, i.e. faults can be detected during the normal operation of a circuit.

Error-detecting codes are widely deployed for the implementation of self-checking circuits (SCCs), mainly because of design cost considerations and because they allow error recovery to be carried out either in hardware or in software. Basically, a functional unit provides an information flow protected by an error detecting code, so that a checker can continuously verify the correctness of the flow and provide an error indication as soon as it occurs. Error correcting codes would on the contrary incur performance penalties related to additional correction circuitry.

Two frequently used codes in SCCs are *parity check code* and *two-rail code*. The former one adds only one parity bit to the information bits and detects all error patterns of an odd number of bits, but cannot detect double errors that can be relevant in a crosstalk-dominated scenario.

The two-rail code represents a signal as a pair of two complementary variables $(x_i, \bar{x}_i)$, thus doubling the number of bus lines. This overhead may not always be acceptable in spite of the high error detection efficiency provided by the code [18].

It has been observed that many faults in VLSI circuits cause unidirectional errors (i.e. 0-1 or 1-0 errors, provided the two kinds of errors do not occur simultaneously). Therefore, coding schemes targeting this kind of errors are well-known to the testing community, such as *m-out-of-n code* and *Berger code*.

In particular, the Berger code is the optimal separable all-unidirectional error detecting code: no other separable code can detect all unidirectional errors with a fewer number of check bits [16]. The check bits are the binary representation of the number of 0s counted in the information bits.

As technology scales toward DSM, traditional schemes used in SCCs may loose their detection effectiveness when applied to on-chip buses, because of the new noise sources that come into play. Unidirectional errors cannot efficiently describe the effects of these noise sources, and the detection capability of multiple bidirectional errors instead of unidirectional errors will be the distinctive feature of error control schemes for DSM NoCs. For instance, crosstalk causes bidirectional errors, when two coupled lines switch in the opposite direction and both transitions are delayed inducing sampling errors.

Many solutions have been proposed to overcome the detection capability limitation of traditional error control schemes with respect to multiple bidirectional errors:

- Acting on the layout in both a code-independent way (i.e. spacing rules, shielding, line crossing, etc.) or code-driven way (e.g. keeping the two complementary bits as far apart as possible in a two rail code). Alternatively, layout information can be exploited to come up with weight-based codes, i.e. extensions of Berger or m-out-of-n codes that are able to deal more efficiently with bidirectional errors [19, 20].

- Acting at the electrical level (e.g. the probability of single errors can be increased with respect to that of bidirectional ones by unbalancing bus lines drivers).

- Using suitable detectors capable to deal with the effects of specific errors (e.g. crosstalk induced errors), but they might not be available is some design styles.

The major drawback of the above mentioned approaches is the need to have layout knowledge or to act at the electrical level. A more general approach could be desirable, wherein the proper course of action against multiple bidirectional errors can be taken early in the design stage, independent of the technology and the final layout, the knowledge of which is not generally available in advance.

Redundant bus enconding remains the most efficient approach for this purpose. Yet, new codes must be used, targeting a more general class of errors than unidirectional ones. Linear codes could be a viable solution, in that they target error multiplicity rather than error direction. Moreover, their codecs can exhibit very leightweigth implementations and can be provided with optional correcting capability. In the following section some details are given about a well-known class of linear codes (Hamming code), whose flexibility and optimality in the number of parities make it suitable for micro-network applications. We also consider cyclic codes, another class of linear codes characterized by highly efficient hardware implementation, and high resilience to a different class of errors.

4. Linear codes

In block coding, the binary information sequence is segmented into message blocks of fixed length; each message block, denoted by u, consists of k information digits. There are a total of 2^k distinct messages.

Qualcomm Exhibit 1006

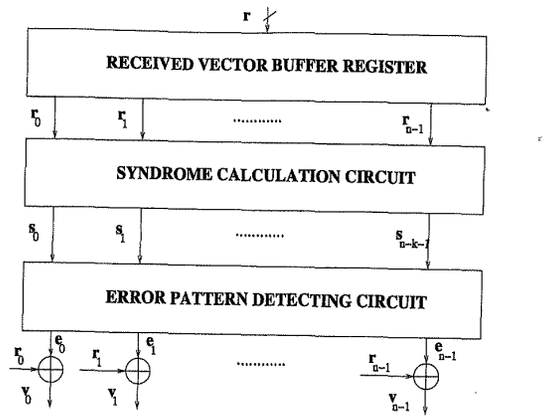


Figure 6.1. Decoder for a Hamming code.

The encoder, according to certain rules, transforms each input message u into a binary n-tuple v with $n > k$. This binary n-tuple v is referred to as the code word of the message u. Therefore, corresponding to the 2^k possible messages, there are 2^k code words. This set of 2^k code words is called a block code. For a block code to be useful, the 2^k code words must be distinct. A binary block code is *linear* if and only if the modulo-2 sum of two code words is also a code word [21].

4.1 Hamming codes

Hamming codes have been the first class of linear codes devised for error correction and have been widely employed for error control in digital communication and data storage systems.

For any positive integer m≥3, there exists a Hamming code [21] with the following parameters:

Code length: $n = 2^m - 1$
Number of information symbols: $k = 2^m - m - 1$
Number of parity-check symbols: $n - k = m$
Error-correcting capability: $t = 1$.

The decoder for a Hamming code is reported in Fig. 6.1. The whole circuit can be implemented as an EXORs tree. Note that the final correction stage is optional, and this makes Hamming code very flexible from an implementation standpoint. According to the codec design, several versions of the Hamming code can be obtained: single error correcting code (SEC), single error correcting and double error detecting code (SECDED) and error detecting Hamming code (ED). The latter uses Hamming code for detection purposes only, and can therefore exploit its full detection capability, which includes not only all single and double errors, but also a large quantity of multiple errors (only those error patterns that are identical to the nonzero code words are undetectable).

Hamming codes are promising for application to on-chip micronetworks because of their implementation flexibility, low codec complexity and multiple bidirectional error detecting capability. Note however that when correction is carried out, the detection capability of the code is reduced, because restrictive assumptions have to be made on the nature of the error. This explains why for a linear code the probability of a decoding error is much higher than the probability of an undetected error [21].

4.2 Cyclic codes

Cyclic codes are a class of linear codes with the property that any code word shifted cyclically (an end-around carry) will also result in a code word. For example, if $c_{n-1}, c_{n-2}, ..., c_1, c_0$ is a code word, then $c_{n-3}, ..., c_0, c_{n-1}, c_{n-2}$ is also a code word.

Cyclic redundancy check (CRC) codes are the most widely used cyclic codes (e.g. in computer networks), and the new DSM scenario could raise the interest for their on-chip implementation, as will be hereafter briefly described.

An (n,k) CRC code is formed using a generator polynomial

$$G(x) = g_m x^m + g_{m-1} x^{n-1} + ... + g_0 x^0$$

of degree $m = n - k$ and having $g_m = g_0 = 1$. If a data message of arbitrary length is expressed as a polynomial M(x), the division of $x^m M(x)$ by $G(x)$ generates a remainder $R(x)$ of degree no more than $(m-1)$. The m coefficients of the remainder are the check bits which are appended to the message. The resulting codeword $T(x)$ is then represented as

$$T(x) = R(x) + x^m M(x) \tag{6.1}$$

The highest degree bits are transmitted first (information bits), check bits last.

A cyclic code with m check bits has two relevant properties [22]:

- All error bursts of length less than or equal to m will be detected. The length of a burst is the span from first to last error, inclusive.

Qualcomm Exhibit 1006

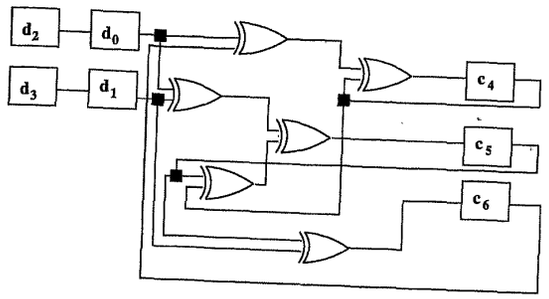


Figure 6.2. Serial-parallel PBCG implementation executing in 2 clock cycles. The transition matrix is raised to the power 2.

- All patterns of 1,2 or 3, or indeed, any odd number of errors will be detected.

Cyclic codes such as CRC codes can play a major role in detecting crosstalk related faults in DSM micronetworks, thanks to the notion of burst error applied to the space domain (i.e. errors on physically contiguous lines). Capacitive coupling is more significant among contiguous lines, therefore crosstalk-induced errors are more likely to occur in neighboring wires. On the contrary, bursts of errors in the time domain regard errors affecting a certain number of contiguous bits transmitted on the same line, but in DSM ICs this scenario is not relevant, except for ISI-related errors.

Hamming code targets error patterns rather than error bursts, and this may represent a useless redundancy: in fact, many error patterns detected by a Hamming code (e.g. multiple errors that are sparse all over the dataword) may occur with an almost null probability in a real on-chip scenario, unlike the kind of errors detected by a cyclic code.

Cyclic redundancy checking could be easily carried out by means of a Linear Feedback Shift Register (LFSR), consisting of $n-k$ flip-flops: their state values are called "syndrome vector". This serial implementation is clearly not suitable for high-performance parallel communication, since it takes k clock cycles to transmit k bits.

The most efficient pareallel CRC implementation makes use of series-parallel sequence generation theory adapted to cyclic codes, and enables the maximum possible speed for a bounded requirement on circuit complexity [24]. The LFSR operation can be described by means of a recurrence equation that relates the syndrome vectors at times i and $i+j$

System		Parity	Area		Power (uW)		Delay (ns)	
Scheme	Error	Lines	Enc.	Dec.	Enc.	Dec.	Enc.	Dec.
UNENCODED	Free	-	-	-	-	-	-	-
SEC	Free	6	5,022	11,034	153	233	1.61	4.56
	Single				153	279		
SECDED	Free	7	6,588	14,238	205	308	2.31	4.85
	Single				205	360		
	Double				254	363		
ED	Free	6	5,022	5,049	153	146	1.61	1.73
	Single				190	144		
	Double				190	148		
	Triple				190	152		
PAR	Free	1	2,538	2,592	61	63	1.40	1.45
	Single				77	62		
	Triple				77	66		
CRC-4	Free	4	2,376	2,637	54	62	0.65	1.02
	Single				68	61		
	Burst				68	65		
CRC-8	Free	8	2,160	2,700	47	58	0.39	0.88
	Single				59	59		
	Burst				59	62		

Table 6.1. Characteristics of synthesized codecs for different Hamming and CRC codes, and their average performance in some cases of interest, wherein communication completes successfully.

through a transition matrix T:

$$s_{i+j} = s_i T^j, j = 1...k \tag{6.2}$$

where k is an integer multiple of j. By raising T to the j-th power we get a corresponding encoding/decoding circuit that executes in k/j clock cycles instead of k (see Fig. 6.2). This approach, called "Parallel Bit Code Generator" (PBCG), can be pushed to the limit by raising T to the power k, obtaining a combinational circuit that executes in one clock cycle at the cost of more circuit complexity. This technique provides a general strategy for identifying architectures with the desired speed-complexity trade-off.

A comparison between synthesized codecs is shown in Table 6.1. Considering a 32 bit encoded link, Hamming codes are compared with two CRC codes detecting all error bursts of length less than 4 and 8, respectively. All of the error control schemes use retransmission as error recovery strategy, except for SEC and SECDED that correct single errors. A 0.25 um synthesis library has been used, with a supply voltage of 2.5 V. Note that CRC codes exhibit the most leightweigth implementations, comparable to that of a single parity check code (PAR).

5. Energy-reliability trade-off

The reliability issue that has been discussed so far is tightly related to another fundamental constraint in NoC design: energy consumption. With reference to on-chip buses, encoding strategies have been successfully exploited by the low-power design community to save energy by reducing the switching activity on long wires (*low power encoding*). As technology scales toward DSM, coupling effects become more significant. This consideration has led to the development of *energy-efficient* and *coupling-driven* bus encoding schemes, that consider coupling power in their power minimization framework. Yet another approach is viable, that consists of employing linear codes to meet predefined requirements on communication reliability, and of minimizing energy consumption by reducing interconnect voltage swing.

5.1 Low power encoding

An early approach to minimize transitions on a multi-bit communication channel was to transfer an inverted word whenever it reduces the Hamming distance between two successive patterns. An additional line is used to indicate whether the word is inverted or not [25]. This technique, called *Bus-Invert Coding*, has been used as the reference point for a number of extensions and changes [26].

Low-power encoding approaches have also been split into a source encoding part and a channel encoding part [29]. The exploitation of source properties has been studied in [27], who proposed *Working-Zone Encoding (WZE)* to exploit locality of memory references. The underlying observation is that applications typically favour a few "working zones" of their address spaces. If both sender and receiver keep a table of base addresses of these working zones, an address can be expressed as an offset along with an index of the working zone based address. Later, in [28], the WZE was extended to multiplexed data buses. A number of other techniques have been proposed, such as *transition pattern encoding* [30], *codebook-based encoding* [31], *probability-based mapping* [32] and *entropy-reducing code* [29].

5.2 Coupling-driven bus encoding for low power

A number of new bus encoding schemes (*coupling-driven signal encoding*) have been developed to alleviate coupling effects for DSM micronetworks or global on-chip interconnects. A simple solution is to minimize the occurrence probability of critical data patterns, wherein adjacent wires switch in the opposite direction.

An extended version of bus-invert coding can be used for this purpose: while the original technique flips the data signal when the number of switching bits is more than half of the number of signal bits, the coupling-driven bus-invert coding inverts the input vector when the coupling effect of the inverted signals is less than that of the original ones. The coupling effect is estimated by categorizing the transitions between adjacent wires and assigning a binary encoding to each of them (e.g. "00" for lines switching in the same direction, "11" for opposite switching directions, "01" for a single line switching, etc..). Then a majority voter recognizes the number of "1"s in the encoded codeword, and if it is more than 50% of the total number of bits, the original data word is inverted due to the excess of critical transitions [35].

Another approach makes use of *dictionary encoding* techniques [33]. The key idea is to exploit a certain amount of correlation between adjacent bits of a word sent across the data bus when using real-world and freely available applications. Since there is not sufficient knowledge on data characteristics to build static dictionaries, adaptive dictionaries are used. An original data word is divided into three parts: non-compressed, index and upper part. The lower parts of the words on data buses typically change quite frequently (i.e. lower correlation), so encoding should not affect the lower part. The encoder uses the index part of the current data to look-up the dictionary. Whenever there is a match (i.e. the upper part of the data word is in the dictionary), it then transmits the index part of the data word and the non-compressed part, so that the decoder is able to recover the upper part from its dictionary. Whenever a dictionary miss occurs, the encoder transmits the unchanged data word.

Using bus encoding to minimize coupling energy, which dominates the total bus energy, has the advantage of being a technology-independent approach that tackles the problem at the architectural level, but it must be observed that the increase in communication reliability comes only as a side-effect: it cannot be precisely quantified and noise sources other than cross-talk tend to be neglected. For DSM micronetworks, where the communication is expected to be inherently unreliable, tighter requirements on reliability will have to be met at a very limited energy cost, and the scenario could not necessarily be crosstalk-dominated.

5.3 Linear codes with low swing signaling

An alternative approach is to apply linear (cyclic) coding to on-chip micronetworks to meet predefined requirements on communication reliability, and to simultaneously minimize energy consumption by using low-swing signaling. Note that the reduction of the voltage swing across

Qualcomm Exhibit 1006

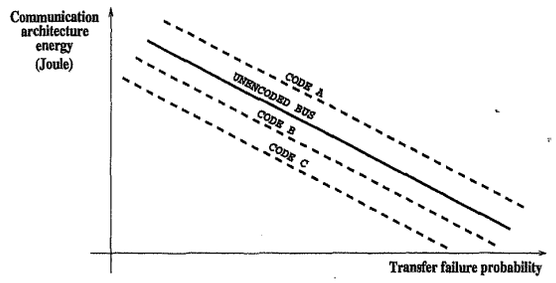


Figure 6.3. Reliability versus energy efficiency for redundant bus encoding schemes making use of low swing signaling.

interconnects determines a decrease of signal-to-noise ratio (SNR), and hence an increased sensitivity to noise sources. Therefore, a trade-off exists between communication reliability and energy efficiency, as depicted in Fig. 6.3. The lower the probability of a codeword being transmitted in error, the higher the energy cost that has to be sustained by the communication architecture: very high detection capabilities have to be ensured by more complex codecs, and wire voltage swings have to be kept high to preserve high SNR values.

The energy efficiency of a code is a measure of its ability to achieve a specified communication reliability level with minimum energy expense. This efficiency depends on a number of parameters, such as code detection capability, number of induced bus transitions, number of redundant lines, encoder and decoder implementation complexity. Therefore, each coding scheme is able to meet the reliability constraints with different energy costs, according to its intrinsic characteristics, and this allows to search for the most efficient code from an energy viewpoint (see Fig. 6.3).

An energy efficiency metric can be defined in order to make a comparison between different coding schemes [36]: the *average energy per useful bit*,

$$\bar{E}_{ub} = \frac{\sum_{i=0}^{m} p_i \bar{E}_i}{\sum_{i=0}^{m} p_i} \tag{6.3}$$

where p_i is the probability of having i errors at the same time affecting the transfer of a codeword. Not all values of i are considered in the metric, but only those ones corresponding to error patterns that are detectable by a certain error control code. E_i is the average energy consumed by the coding scheme implementation in the event described by p_i. All average energies are referred to a single useful bit: therefore the energy overhead associated with redundant parity lines is ascribed

to the information lines, thus considering the impact of coding efficiency on energy efficiency.

As an example, Hamming SEC works properly both in the error free case (it happens with probability p_0 and the average communication and codec-related energy consumption per useful transferred bit is $\bar{E}_0$) and in the single error case (probability p_1 and energy $\bar{E}_1$).

In the above defined metric, the denominator can be thought of as the probability of correct operation of the system, and it has the same value for all of the schemes that have to be compared, as it represents the common predefined communication reliability requirement.

The average energies per useful bit $\bar{E}_i$ can be obtained, for each scheme, as follows:

$$\bar{E}_i = \bar{E}b_i + \bar{E}e_i + \bar{E}d_i \tag{6.4}$$

where $\bar{E}b_i$ is the average energy per useful bit spent for global line transitions, while $\bar{E}e_i$ and $\bar{E}d_i$ express the average energy consumption of encoder and decoder respectively. This figure of merit could be refined to account for coupling effects and their impact on energy consumption.

An important parameter needed to estimate enery efficiency of a code is the voltage swing used across interconnects. Its value is tightly related to the communication reliability, and a simple model has been proposed by Shanbhag [34]. The assumption is that every time a transfer occurs across a wire, it can make an error with a certain probability ϵ. The parameter ϵ depends on the knowledge of different noise sources and their dependence on the voltage swing, and is therefore difficult to estimate. So, for purpose of statistical analysis, the sum of several uncorrelated noise sources affecting each line of a point-to-point link is modelled as a single gaussian noise source, and the value of ϵ depends on the voltage swing V_{sw} and the variance σ_N^2 of the noise voltage V_N:

$$\epsilon = Q(\frac{V_{sw}}{2\sigma_N}) \tag{6.5}$$

where $Q(x)$ is the gaussian pulse

$$Q(x) = \int_x^\infty \frac{1}{\sqrt{2\pi}} e^{-\frac{y^2}{2}} dy \tag{6.6}$$

This model accounts for the decrease of noise margins (and hence for an increase of the line flipping probability ϵ) caused by a decrease of the voltage swing across a line, and allows a simplified investigation of the energy-reliability trade-off for the introduced linear codes.

As a case study, let us now consider the scenario depicted in Fig. 6.4. Redundant encoding has been applied to the communication channel (a

Qualcomm Exhibit 1006

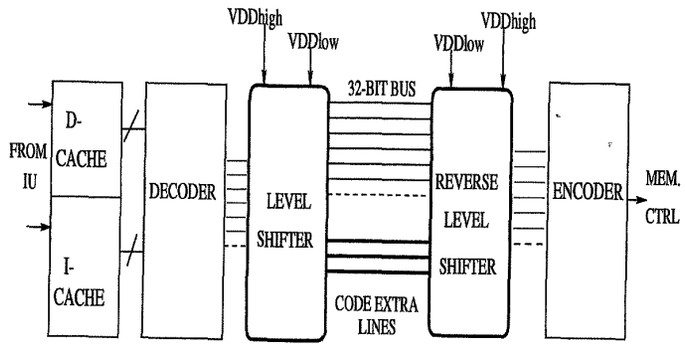


Figure 6.4. Realistic SoC setting where the energy-reliability trade-off is investigated.

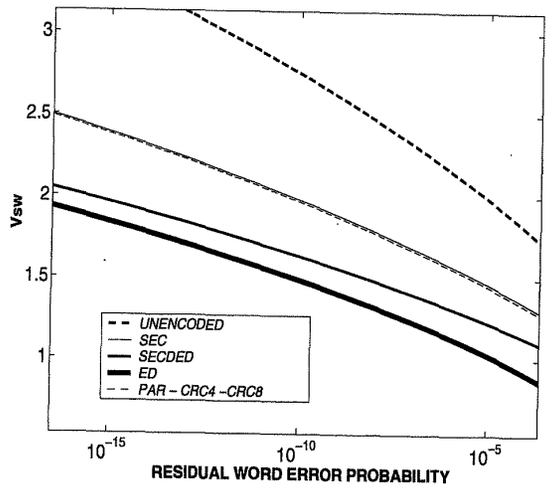


Figure 6.5. Minimum voltage swing needed by each coding scheme to meet a predefined communication reliability requirement.

point-to-point bus) between a master (a SPARC V8 processor) and a slave (an on-chip memory). The link is provided with retransmission capability. Encoder and decoder are powered at standard voltage levels, while voltage level translators allow wires to work at a reduced swing. This setup points out the energy efficiency of the codes under test and provides useful indications for NoC designers because it can be thought of as a point-to-point connection in a NoC, e.g. between the network

interface and a switch, or between two switches. For the purpose of our analysis, the only relevant difference respect to a multi-hop NoC scenario is that this latter makes use of data packetization, and this also affects the way retransmissions are carried out, as will be discussed in section 6.

Given the requirement on communication reliability for this link, the minimum wire voltage swings V_{sw} that have to be used by error control codes to meet the commmon constraint can be derived from the Shanbhag model, and are reported in Fig. 6.5.

The unencoded link has of course to use the highest swing, while the other encoding schemes can rely on their error detection capability, independently of the error recovery action. Note that retransmission-based techniques, however, require lower swings than correction oriented ones, because they do not have to make restrictive assumptions on the nature of errors in order to be able to correct them. CRC codes have the same requirements as SEC and PAR, as they have the common characteristic of detecting single errors (having the highest occurrence probability with the used model) and not all double ones [37].

In general, the larger the detection capability of a coding scheme, the lower the voltage swing that can be used across interconnects: in fact, the lower SNR is counterbalanced by the ability of the decoder to detect a large number of error patterns and to take the proper course of recovery action. The next step is to evaluate whether such a swing reduction is beneficial in terms of energy dissipation: the energy overhead associated with error recovery must not make up for the low-swing related savings.

The impact of error recovery techniques can be assessed by computing the energy efficiency metric for each code. Results in Fig. 6.6 refer to a bit line load capacitance C_L of 5 pF (a wire of about 1 cm in a 0.25 um technology). Such a load capacitance makes the energy cost associated with bus transitions dominant with respect to codec-related energy overhead. Retransmission based strategies (ED, PAR, CRC) perform better than SEC because they can work at lower voltage swings thanks to their higher detection capabilities, and this makes the difference independently of the increased number of transitions on the link lines. Note that SECDED gives satisfactory results, in that it uses a mixed approach: correction is used for single errors, retransmission for double ones. The leftmost part of the graph is the one of interest, because it corresponds to mean times between failures (MTBFs) in the order of years. In the rightmost part, MTBF is hundreds of milliseconds.

Fig. 6.7 shows the same curves plotted for a C_L of 0.5 pF (a few millimeter long wires). Here transitions on the link lines play a minor role, while the contribution of codec complexity becomes relevant. This explains why the gap between SEC, SECDED and the other schemes be-

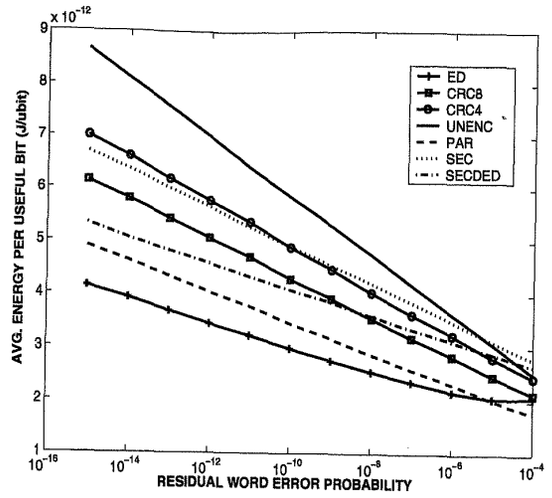


Figure 6.6. Energy efficiency of coding schemes for point-to-point links. Wire lengths are in the the order of centimeters.

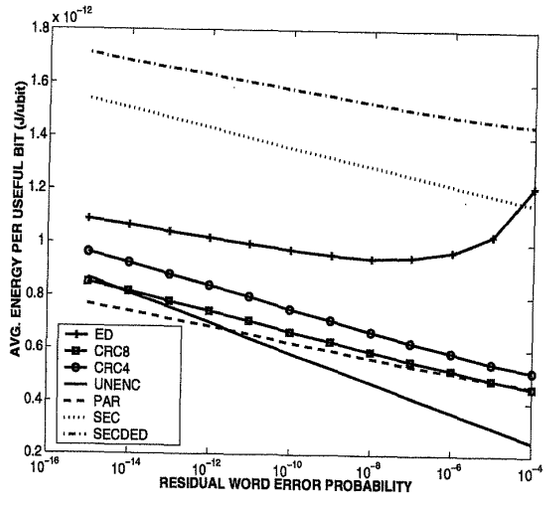


Figure 6.7. Energy efficiency of coding schemes for point-to-point links. Wire lengths are in the order of millimeters.

comes more relevant: correction circuitry at the decoder side makes the difference. Among retransmission-oriented schemes, PAR outperforms ED, and CRC4 and CRC8 become competitive.

The illustrated results point out that the detection capability of a code plays a major role in determining its energy efficiency, because it is directly related to the wire voltage swing. As regards the error recovery technique, error correction is beneficial in terms of recovery delay, but has two main drawbacks: it limits the detection capability of a code and it makes use of high-complexity decoders. On the contrary, when the higher recovery delay (associated with the retransmission time) of retransmission mechanisms can be tolerated, they provide higher energy efficiency, thanks to the lower swings and simpler codecs (pure error detecting circuits) they can use while preserving communication reliability. Mixed approaches such as SECDED could be a trade-off solution.

Error-resilient low-swing signaling can also be ensured by means of an adaptive optimization technique, wherein variable low-swing transmission is used to minimize energy and a linear error detecting code is employed for reliable communication [38]. A controller is the keypoint of this implementation, that tunes the voltage swing and the operating frequency based on the working conditions (actual workload, noise, technology quality). Therefore, the desired energy-reliability trade-off can be dinamically adapted to non-stationary conditions of operation and noise levels.

6. Multi-hop NoCs

In multi-hop NoCs, the efficiency of retransmission also depends on network topology and on the relative distance (expressed as a number of hops) between sender and receiver. There are two possible scenarios:

1 The error recovery strategy can be *distributed* over the network. Each communication switch is equipped with error detecting/correcting circuitry, so that error propagation can be immediately stopped. This is the only way to avoid routing errors: should the header get corrupted, its correct bit configuration can be immediately restored, preventing the packet from being forwarded across the wrong path to the wrong destination. Retransmission-oriented schemes also need buffering resources at each switch, so their advantage is more in terms of higher detection capability rather than circuit complexity, and the results derived throughout this chapter hold.

2 Alternatively, the approach to error recovery can be *concentrated*: only end-nodes are able to perform error detection/correction. In

this case, retransmission may not be convenient at all, especially when source and destination are far apart from each other, and retransmitting corrupted packets would stimulate a large number of transitions, beyond giving rise to large delays. For this scenario, error correction is the most efficient solution, even though proper course of action has to be taken to handle incorrectly routed packets (retransmission time-outs at the source node, deadlock avoidance, etc.).

Another consideration regards the way retransmissions are carried out in a NoC. Traditional shared bus architectures can be modified to perform retransmissions in a "stop and wait" fashion: the master drives the data bus and waits for the slave to carry out sampling on one of the following clock edges. If the slave detects corrupted data, a feedback has to be given to the master, scheduling a retransmission.

In packetized networks, data packets transmitted by the master can be seen as a continuous flow, so the retransmission mechanism must be either "go-back-N" or "selective repeat". In both cases, each packet has to be acknowledged (ACK), and the difference lies in the receiver (switch or network interface) complexity. In a "go-back-N" scheme, the receiver sends a not ACK (NACK) to the sender relative to a certain incorrectly received packet. The sender reacts by retransmitting the corrupted packet as well as all other following packets in the data flow. This alleviates the receiver from the burden to store packets received out of order and to reconstruct the original sequence.

On the contrary, when this capability is available at the receiver side (at the cost of further complexity), retransmissions can be carried out by selectively requiring the corrupted packet without the need to retransmit also successive packets. The trade-off here is between switch (interface) complexity and number of transitions on the link lines.

7. Conclusions

In this chapter, the energy-reliability trade-off has been investigated for the basic building blocks of SoC communication architectures, providing NoC designers with guidelines for the selection of energy efficient error control schemes. In particular it has been showed that:

- Error-control coding can significantly enhance communication reliability and contemporarily reduce energy-per-bit dissipation. The energy overhead introduced by redundant parity lines is counterbalanced by the reduced voltage swings.

- The optimal encoding scheme for a point-to-point link is not unique but depends on loading conditions. For lines that are a few millimeters long, CRC codes are the most efficient solution because they target error bursts (better modeling the effects of crosstalk at a high abstraction level) with very leightweigth implementations.
- With state of the art technology, retransmission turns out to be more efficient than correction from an energy viewpoint. However, as an effect of the IC scaling scenario, communication energy is likely to largely overcome computational energy, and for very DSM technologies error correction might bridge the gap.

Interesting research and development opportunities lie ahead: future NoCs could implement error correction and error detection at various levels of the communication stack, using dedicated hardware, even coupled with application-level software support. In general, communication reliability is likely to become a top-level design constraint and its relationships with energy efficiency will become one of the most promising exploration directions in future designs.

References

[1] Benini L., and De Micheli G."Networks on chips: a new SoC paradigm" Computer, Vol.35, January 2002, pp.70-78.

[2] Hui Z., George V., and Rabaey J.M. "Low-swing on-chip signaling techniques: effectiveness and robustness" IEEE Transactions on VLSI Systems, Vol.8, NO.3, June 2000, pp.264-272.

[3] Svensson C."Optimum voltage swing on on-chip and off-chip interconnect" IEEE Journal of Solid-State Circuits, Vol.36, NO.7, July 2001, pp.1108-1112.

[4] Sylvester D., and Hu C."Analytical modeling and characterization of deep-submicron interconnect"Proceedings of the IEEE, May 2001, pp.634-664.

[5] Dally, W.J. and Poulton, J.W. (1998). *Digital Systems Engineering.* New York: Cambridge University Press.

[6] Bakoglu, H.B. (1990) *Circuits, Interconnects and Packaging for VLSI.* MA: Addison-Wesley, 1990

[7] Jiang Y.M., and Cheng K.T. "Analysis of performance impact caused by power supply noise in deep submicron devices" Proceedings of Design Automation Conference, June 1999, pp.760-765.

[8] Chen H.H., Ling D.D. "Power supply noise analysis methodology for deep-submicron VLSI chip design" Proceedings of Design Automation Conference, June 1997, pp.638-643.

[9] Erhard K.H., Johannes F.M., and Dachauer R. "Topology optimization techniques for power/ground networks in VLSI" Proceedings of Design Automation Conference in Europe, March 1992, pp.362-367.

[10] Dutta R., and Sadowska M.M. "Automatic sizing of power/ground networks in VLSI" Proceedings of Design Automation Conference, June 1989

[11] Ang M., Salem R, and Taylor A "An on-chip voltage regulator using switched decoupling capacitors" Proceedings of int. Solid-State Circuits Conference Dig.Tech.Papers, February 2000, pp.438-439

[12] Zhao S., Roy K. and Koh C.K. "Decoupling capacitance allocation and its application to power-supply noise-aware floorplanning" IEEE Transactions on CAD of Integrated Circuits and Systems, Vol.21, NO.1, January 2002, pp.81-92.

[13] Prince B. "Report on Cosmic Radiation Induced SER in SRAMs and DRAMs in Electronic Systems, May 2000.

[14] Steinecke T. "Design-in EMC on CMOS large-scale integrated circuits" 2001 Int. Symposium on electromagnetic compatibility, Vol.2, August 2001, pp.910-915.

[15] Bogliolo A. "Encoding for high-performance energy-efficient signaling" ISLPED, August 2001, pp.170-175.

[16] Pradhan, D.K. (1986). *Fault-Tolerant Computing: Theory and Techniques.* Englewood Cliffs, NJ: Prentice Hall.

[17] Metra C., Favalli M., and Ricco' B. "On-line detection of bridging and delay faults in functional blocks of CMOS self-checking circuits" IEEE Transactions on CAD, Vol.5, NO.4, 1997, pp.770-776.

[18] Favalli M., and Metra C. "Bus crosstalk fault-detection capabilities of error-detecting codes for on-line testing" IEEE Transactions on VLSI Systems, Vol.7, NO.3, September 1999, pp.392-396

[19] Das D., and Touba N. "Weight-based codes and their application to concurrent error detection of multilevel circuits" Proceedings of IEEE VLSI Test Symposium, 1999, pp.370-376.

[20] Favalli M., and Metra C. "Optimization of error detecting codes for the detection of crosstalk originated errors" Proceedings of DATE 2001, March 2001, pp.290-296.

[21] Lin, S. and Costello, D.J. (1983) *Error control coding: fundamentals and applications.* Englewood Cliffs, NJ: Prentice Hall.

[22] Mazo J.E., and Saltzberg B.R. "Error-burst detection with tandem CRC's" IEEE Transactions on Communications, Vol.39, NO.8, August 1991, pp.1175-1178.

[23] Sobski A., and Albicki A. "Partitioned and parallel cyclic redundancy checking" Proceedings of the Midwest Symposium on Circuit and Systems, Vol.1, August 1993, pp.538-541.

[24] Popplewell A., O'Reilly J.J, Williams S. "Architecture for fast encoding and error detection of cyclic codes" IEE Proceedings on Communications, Speech and Vision, Vol.139, NO.3, June 1992, pp.340-348.

[25] Stan M.R., and Burleson W.P. "Bus-invert coding for low-power I/O" IEEE Transactions on VLSI Systems, Vol.3, NO.1, 1995, pp.49-58

[26] Kretzschmar C., Siegmund R., and Mueller D. "Adaptive bus encoding technique for switching activity reduced data transfer over wide system buses" Int. Workshop - Power and Timing Modeling, Optimization and Simulation, September 2000.

[27] Musoll E., Lang T., and Cortadella J. "Working-zone encoding for reducing the energy in microprocessor address buses" IEEE Transactions on VLSI Systems, Vol.6, NO.4, December 1998, pp.568-572.

[28] Lang T., Musoll E., and Cortadella J. "Extension of the working-zone-encoding method to reduce the energy on the microprocessor data bus" Int. Conference on Computer Design, October 1998.

[29] Ramprasad S., Shanbhag N.R., and Hajj I.N. "A coding framework for low-power address and data busses" IEEE Transactions on VLSI Systems, Vol.7, NO.2, June 1999, pp.212-221.

[30] Sotiriadis P.P., and Chandrakasan A. "Bus energy minimization by transition pattern coding (TCP) in deep sub-micron technologies" IEEE/ACM Int. Conference on CAD, 2000, pp.322-327.

[31] Komatsu S., Ikeda M., and Asada K. "Low power chip interface based on bus data encoding with adaptive codebook method" Great Lakes Symposium on VLSI, 1999, pp.368-371.

[32] Benini L., Macii A., Macii E., Poncino M., and Scarsi R. "Architectures and synthesis algorithm for power efficient bus interfaces" IEEE Transactions on CAD ICs and Systems, Vol.19, NO.9, 2000, pp.969-980.

[33] Lv T., Henkel J., Lekatsas H., and Wolf W. "An adaptive dictionary encoding scheme for SOC data buses" Proceedings of DATE 2002, March 2002, pp.1059-1064.

[34] Hegde R., and Shanbhag N.R. "Toward achieving energy efficiency in presence of deep submicron noise" IEEE Transactions on VLSI Systems, Vol.8, NO.4, August 2000, pp.379-391.

[35] Kim K.W., Baek K.H., Shanbhag N., Liu C.L., and Kang S.M. "Coupling-driven signal encoding scheme for low-power interface design" IEEE/ACM ICCAD, November 2000, pp.318-321.

[36] Bertozzi D., Benini L., and De Micheli G. "Low power error resilient encoding for on-chip data buses" Proceedings of DATE 2002, March 2002, pp.102-109.

[37] Bertozzi D., Benini L., and Ricco' B. "Energy-efficient and reliable low-swing signaling for on-chip buses based on redundant encoding" Proceedings of ISCAS 2002, May 2002, Vol.I, pp.93-96.

[38] F. Worm, P. Ienne, P. Thiran and G. De Micheli, " An Adaptive Low-power Transmission Scheme for On-chip Networks," ISSS, Proceedings of the International Symposium on System Synthesis, Kyoto, October 2002, pp. 92-100.

Qualcomm Exhibit 1006

Chapter 7

Testing Strategies for Networks on Chip

Raimund Ubar, Jaan Raik
Tallinn Technical University, Estonia

Key words:

Abstract: The complexity of Networks-on-Chip (NoC) makes the application of traditional test methods obsolete. For NoC, a combination of methods known from the System-on-Chip, memory and FPGA test areas should be used. That includes functional test, scan test, logic BIST, RAM BIST and testing of interconnect switches and wires. The increasing complexities of systems based on deep-submicron technologies cause two contrary trends: low-level defect-orientation to reach the high reliability of testing and high-level behavioral modeling to reach the efficiency of test generation. Hierarchical approaches seem to be the solution. Built-In Self-Test (BIST) is the main concept for testing the cores in systems on chip. Hybrid BIST containing both hardware and software components is probably the most promising approach to test the nodes of NoC. In densely packaged NoC with embedded memories and reusable cores scan-based approaches, P1500 standard for core test, test access mechanisms, test control and isolation issues are prospective methods. Testing the NoC interconnect switches and wires is also an important issue.

1. INTRODUCTION

The reliability of electronic systems is no longer merely a topic of limited critical applications e.g. in military, aerospace and nuclear industries, where failures may have catastrophic consequences. Electronic systems are becoming ubiquitous and their reliability issues are present in all types of consumer applications. Adequate testing of electronic products is a must. The complexity of systems and new failure models accompanied by modern technologies causes the necessity for developing more efficient test methods.

A. Jantsch and H. Tenhunen (eds.), Networks on Chip, 131–152.
© 2003 *Kluwer Academic Publishers. Printed in the Netherlands.*

Qualcomm Exhibit 1006

In the middle of 1990s, the embedded core based System-on-Chip (SoC) concept evolved. This brought along new strategies and standards dedicated to SoC test. Since circuits are growing in size, the bus-based interconnection scheme used in SoCs is becoming a bottleneck in design. In the beginning of the 21-st century the design methdology is moving towards the NoC approach, where package-based communication infrastructure is used, thus avoiding problems with scalability and reuse of SoC interconnects. Much of the knowledge inherited from core and SoC testing will be used also in the NoC era. However, the presence of the regular communication structure requires new dedicated methods to test it. The goal of this Chapter is to discuss the range of strategies applicable in testing NoCs.

2. GENERAL CONCEPTS OF DIGITAL TEST

2.1 Testing and the test quality

The term *testing* does not refer to checking the correctness of the function of the implemented circuit (i.e. *functional verification*). By testing we understand checking for manufacturing correctness. In other words, we check whether the design implemented in silicon is free of manufacturing defects. Thus, according to their definition, testing and verification are different tasks with different goals.

The quality of the test influences the level of reliability of the manufactured electronic circuits in the following way. Let *yield Y* be the fraction of the circuits that are manufactured without defects in a given production process. Knowing the yield Y and the quality of the test stimuli T, it is possible to calculate the fraction of the bad chips that pass the test. This fraction is called *defect level,* and it can be calculated as $DL = 1 - Y^{1-T}$ [1].

As it can be seen from the formula, there are only two ways to minimize the fraction of defective devices passing the test. One possibility is to maximize the yield Y, the other way is to maximize the test quality T. Yield is very much dependent of the maturity of the manufacturing process. The newer the process, the lower the yield. During the recent years, the manufacturing processes have been renewed in an increasingly frequent rate, resulting in low process yields which then in turn cause problems.

Test quality T is expressed as the *defect coverage* of the test. However, it is not really practical to measure the exact defect coverage of a test. Mathematical mechanisms, called *fault models*, are needed instead in order to model the actual physical defects. The fraction of modeled faults covered by a test is called *fault coverage*. In practice, fault coverage is used as an approximation of the test quality T.

2.2 Errors, defects and fault models

An incorrect operation of the system being tested is referred to as an *error*. The causes of errors may be faults, i.e. *design errors* or *physical defects*. The types of physical defects depend on the technology. The most common types of defects during manufacturing are shorts, opens, cracks, missing transistors etc. In general, the defects do not allow a direct mathematical treatment of testing and diagnosis. The solution is to deal with *fault models*.

Representing defects by fault models is useful on different reasons: the complexity of simulation reduces (many defects may be modeled by the same fault model), one fault model can be used for many technologies, and the tests generated for a fault model may be used for defects whose effect is not completely understood.

Fault models can be categorized according to the level of abstraction. The choice of the level means a trade-off between the fault model's ability to accurately represent a physical defect and the speed of fault simulation. For example, behavior level fault simulation is the fastest but least accurate. On the other hand, layout level defect analysis is most accurate but requires too much time. Logic level is the most common for fault modeling, and the most common fault model is *stuck-at fault* (SAF), which means that a line is stuck at a fixed logic value 0 or 1.

In case of functional level fault modeling the impact of physical defects are mapped to the behavior of functional blocks. Functional fault models are used for example for RAM [2] and for microprocessor [3] testing.

3. TEST GENERATION

3.1 Functional modeling of faults

The efficiency of test generation (test quality, test generation speed) is highly dependent on the system and fault models. Traditional low-level methods and tools have lost their importance, functional, behavioral, or hierarchical methods are gaining more and more popularity. The advantage of an hierarchical approach lies in the possibility of constructing test plans at higher functional levels, and modeling faults at lower levels.

The traditional SAF model cannot guarantee high quality tests because it does not represent adequately the majority of real defects in submicron digital circuits. These facts have been well known, but usually ignored in engineering practice. In earlier works on layout-based test generation [4,5] the whole circuits were analyzed as single blocks. Such an approach is

Qualcomm Exhibit 1006

computationally expensive. In [6], a defect-oriented hierarchical method was proposed based on defect preanalysis of components, and on using the results of preanalysis in higher level fault simulation or test generation.

Consider a Boolean function $y = f(x_1,x_2,\ldots,x_n)$ representing a component in a circuit. Introduce a Boolean variable d for modeling a given defect in the component or in its neighborhood layout, which may affect the value y by converting the function f into another function $y = f^d(x_1,x_2,\ldots,x_n,x_{n+1},\ldots x_p)$. Here, the variables $x_{n+1},\ldots,x_p$ belong to the neighborhood of the component, which may influence the function y in the presence of defect d. Introduce now a generalized function with parameter d

$$y^* = f^*(x_1, x_2, \ldots, x_n, x_{n+1}, \ldots x_p, d) = (\neg d \wedge f) \vee (d \wedge f^d),$$

to describe the behavior of the component simultaneously for the two possible cases: the normal case ($d = 0$), and the faulty case ($d = 1$). The faulty function f^d can be found either by logical reasoning, by carrying out defect simulation, or by carrying out experimental work to learn the real physical behaviour of the defect. The solutions of the Boolean differential equation

$$W^d = \partial y^* / \partial d = 1 \qquad (1)$$

describe the conditions, which allow to observe the defect d on the line y.

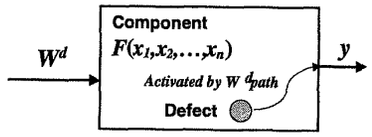


Figure 1. Functional fault model for a physical defect

The method of modeling defects d by constraints W^d can be regarded first, as a method of mapping arbitrary physical defects onto the logic level, and second, as a universal method of fault modeling. The couple (W^d, y), meaning that the defect d in case of $W^d = 1$ will change the value of y, is called *functional fault model* of the defect d (Fig. 1).

The conditions W^d for activating defects d can be used as constraints at higher (logical or register transfer) levels either for fault simulation or for test generation without paying attention to the physical details of defects.

3.2 Hierarchical approach to test

The method of defining faults by conditions W^d allows to unify the diagnostic modeling of components (or modules) of a circuit (or system)

without going into structural details of components and into the diagnostic simulation of communication network of components. In both cases, W^d describes how a lower level fault d (either a defect in a component or a defect in a network) should be activated at a higher level to a given node in a circuit (or system).

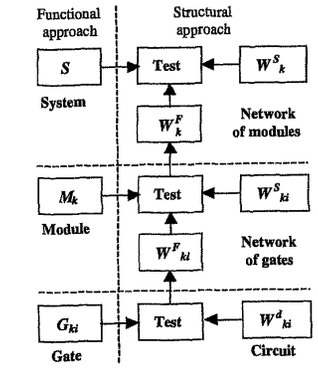


Figure 2. Hierarchical test approach for digital systems

In Fig.2, a hierarchical test concept based on the functional fault model for a 3-level system is illustrated. Functional and structural approaches are possible. In the first case, only the information about the functional behaviour is used. In the structural approach, tests are targeted to detect the faults in the networked components and in the network communication.

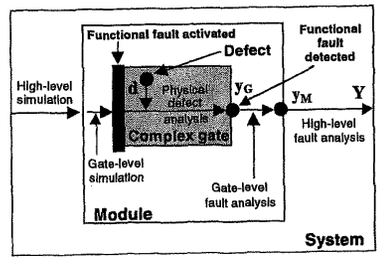


Figure 3. Hierarchical fault simulation

Consider a node k represented by a variable x_k as the output of a module M_k. Associate with k a set of faults $R_k = R^F_k \cup R^S_k$ where R^F_k is the subset of faults in M_k, and R^S_k is a subset of structural faults (defects) in the "network neighborhood" of M_k. Denote by W^F_k (W^S_k) the set of conditions W^d for the faults $d \in R^F_k$ ($d \in R^S_k$). The set W^F_k can be found by low level test

generation for the structural faults in M_k described by W^F_{ki} and W^S_{ki} for gates G_{ki} in M_k. Test generated at the lower level for M_k can be considered as a set of constraints W^F_k (functional fault model) for test generation in the next higher level.

Consider a task of defect oriented fault simulation in a system which is represented on three levels: register transfer, gate and defect levels (Fig. 3). Formally, if Y is the system variable representing an observable point (a register) of the system, y_M is an output variable of a logic level module and y_G is the output of a logic gate with a physical defect d, then the condition to detect the defect d on the observable test point Y of the system is

$$W = \partial Y/\partial y_M \wedge \partial y_M/\partial y_G \wedge W^d = 1,$$

where $\partial Y/\partial y_M$ means the fault propagation condition calculated by high-level modeling, $\partial y_M/\partial y_G$ is the fault propagation condition (Boolean derivative) calculated by gate-level modeling, and W^d is the functional fault constraint calculated from (1) by the gate preanalysis.

3.3 Hierarchical test generation for sequential circuits

Due to the fact that traditional gate-level test generation is not satisfying, hierarchical methods are gaining in popularity. These methods [7] take advantage of high level information, while generating tests for gate level faults or even for physical defects. In the hierarchical approach, top-down and bottom-up strategies can be distinguished.

In the *bottom-up* approach, pre-calculated tests for system components generated on low-level will be assembled at a higher level. This approach fits well to the hierarchical approach described above, which covers in a uniform way both component and network testing. However, bottom-up algorithms ignore the incompleteness problem: the constraints imposed by the network structure may prevent the local test solutions (the constraints W^F_k and W^S_k for a module M_k under test) from being assembled into a global test. This approach would work well only if the the needed testability demands were fulfilled to avoid the incompleteness problems. The *top-down* approach has been proposed to solve the test generation problem by deriving environmental constraints for low-level solutions. This method is more flexible since it does not narrow the search for the global test solution to pregenerated patterns for the system modules.

In [8], a hierarchical test generation algorithm DECIDER was developed and implemented as a top-down approach. The basic principle of the method is to activate control sequences that would propagate the fault effect from the output of the Module Under Test (MUT) to the primary outputs of the design and from the inputs of MUT to primary inputs. The sequences are generated

by deterministic search. In order to establish the global test solution in terms of activated test paths, conditions in the control and data flow graph of the circuit have to be satisfied. We refer to this type of constraints as *sequence activation constraints*. Another type of constraints are also considered that reflect the value changes along the paths from the primary inputs of the circuit to the inputs of the MUT. These constraints are called *value transformation constraints*. Both types of constraints can be represented by similar data structures and manipulated by common operations.

After the constraints are created they have to be justified. In the traditional case this has been done by implementing the transparency modes. DECIDER neglects this concept and does not attempt to perform any value assignments to the encountered module inputs at this stage. Instead it includes the entire module into the constraint and continues justification of its inputs until primary inputs are reached. This allows the algorithm to avoid the solution loss accompanied with using transparency modes and generally speeds up the search process. Subsequent to justification, the set of constraints have to be solved. However, the extracted constraints are not always satisfiable. They may be inconsistent or too complex to solve for the constraint satisfaction algorithm. In this case, a backtrack occurs and the high-level test generation algorithm attempts to activate an alternative global test solution. In general, a single global test solution may not be enough to reach 100 % fault coverage for the MUT, therefore, test for a module can consist of patterns generated during different activated path solutions.

In Table 1, main characteristics of the benchmark circuits used in the experiments and the comparison of ATPG tools are presented. The other tools in comparison were GATEST [9], a genetic algorithm based test generator and HITEC [10], a deterministic gate-level ATPG, respectively. The test generation times differ greatly. For example, DECIDER spends 26 - 615 times less CPU time for test generation than GATEST. The experiments were run on a 300 MHz SUN UltraSPARC 10 workstation.

Table 1. Comparison of test pattern generation tools

circuit	# gates	# faults	DECIDER		GATEST		HITEC	
			cov., %	time, s	cov., %	time, s	cov., %	time, s
gcd	227	844	**92.2**	**3.4**	**92.2**	89.8	89.3	195.6
mult8x8	1058	3915	**79.4**	**13.6**	77.3	1585	63.5	1793
diffeq	4195	15836	**96.0**	**15.8**	**96.0**	9720	95.1	> 8 h

4. BUILT-IN SELF-TEST

Built-in self-test (BIST) is a testing technique in which the components of a system are used to test the system itself. The motivation to use BIST has

Qualcomm Exhibit 1006

arisen particularly from the cost of test pattern generation and from the volume of the test data which have a trend to increase with circuit size, and from the long testing time. When using BIST, it will be possible to test the system at working speed. This aspect is especially important for present technology, where the interconnection effects are causing significant delays.

BIST techniques can be classified as: *on-line* BIST, which occurs during normal functional operation, and *off-line* BIST, which is used when the system is not in its normal working mode [11]. In *concurrent on-line* BIST, coding techniques or duplication and comparison are usually used. In *nonconcurrent on-line* BIST, testing is carried out while a system is in an idle state. This is often accomplished by executing diagnostic software or firmware routines. Off-line BIST is based on using on-chip test generators and output response analyzers or microdiagnostic routines.

4.1 Generic architecture of BIST

The main functions of a BIST are: test generation, test application, and response verification. Different approaches can be used for these functions depending on the type of circuit under test and on the test environment. For example, the methodology used for memory testing (*memory* BIST) is different than that used with random logic circuits (*logic* BIST or LBIST).

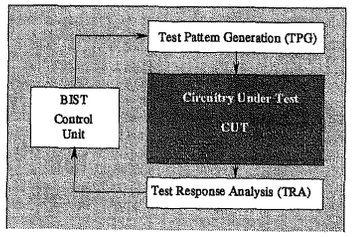


Figure 4. A generic architecture of BIST

A generic architecture of LBIST with three main blocks – Test Pattern Generator (TPG), Test Response Analyzer (TRA) and BIST controller - is represented in Fig. 4. Usually TPG and TRA are based on some forms of Linear Feedback Shift Register (LFSR) [12]. They can be connected directly to the CUT or indirectly via scan path registers.

4.2 Test pattern generation

Test pattern generation approaches can be divided into the following categories: exhaustive, pseudo-exhaustive, pseudorandom, weighted

pseudorandom, and deterministic testing. More advanced forms of on-line testing are hybrid BIST, and functional BIST.

In the *exhaustive testing*, all possible input patterns are applied to the CUT. For on-line test generation, counters or LFSR can be used. For an n-input combinational circuit, all possible 2^n patterns are applied. The advantage of this approach is that all irredundant faults which don't have sequential character can be detected. The drawback of this approach is that when n is large, the test application time becomes prohibitive.

Pseudoexhaustive testing [13] retains the advantages of exhaustive testing while significally reducing the number of test patterns. The basic idea is to partition the CUT into subcircuits such that each subcircuit has few enough inputs for exhaustive testing. For example, a n-bit ripple carry adder can be tested using only 8 patterns, independently of the value of n.

Pseudorandom testing is based on using an LFSR [12]. An example of LFSR with 4 flip-flops is shown in Fig. 5. The outputs of the flip-flops form the test pattern. The number of possible test patterns is equal to the number of states of the circuit which is determined by the feedback structure. The maximum number of possible patterns generated by an n-bit LFSR is $2^n - 1$, since the all-zeros pattern cannot be generated. In random pattern generation, a pattern may be repeated several times in the process. An LFSR produces patterns without repetition. The length of the pseudorandom test depends on the seed of the LFSR. The fault coverage reachable by pseudorandom testing depends highly on the functionality of the CUT and the logic depth. A lot of the faults in the CUT may be random pattern resistant, meaning that not all signal transitions can be controlled and observed by the generated patterns. One way to solve the pattern resistance problem is to improve the testability of the CUT. Another way is to spend more effort on the pattern generation by using a *weighted pseudorandom* pattern generator [14].

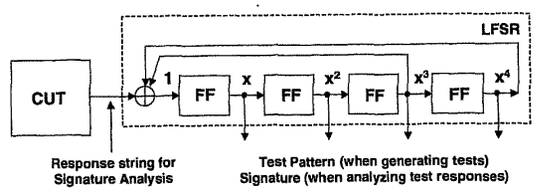


Figure 5. Linear Feedback Shift Register (LFSR)

In *deterministic testing*, traditional test generation techniques are used to generate test patterns to be applied to the CUT when it is in the BIST mode. The test is normally stored in a ROM. This approach is rarely used in a pure form because of the high overhead. Examples of deterministic testing are diagnostic microroutines run by microprocessors. The deterministic BIST

approach is also used in memory BIST, where different algorithms like March, GALPAT, Walking 0s and 1s can be implemented in hardware [2].

In many cases, the LFSR-based approach to on-chip test generation does not guarantee a sufficiently high fault coverage (especially in the case of large and complex designs) and demands very long test application times in addition to high area overheads. Therefore, several proposals have been made to combine pseudorandom test patterns generated by LFSR with deterministic patterns [15-19].

Controlling the LFSR is another approach to cope with random pattern resistant faults. Re-seeding the LFSR or using multiple polynomial LFSR are alternative approaches to control the LFSR during the testing process [15]. Different mixed-mode approaches based on pseudorandom patterns and deterministic information have been developed. It is possible to pregenerate deterministic patterns for detecting random pattern resistant faults. BIST execution then combines pseudorandom and deterministic patterns according to certain protocol. Deterministic information can be stored in a ROM or be implemented as a hardware circuit that is activated at testing time. Bit fixing is one of these techniques. The output bits of LFSR, at a certain position in the sequence, are changed to generate a needed deterministic pattern [16]. An alternative approach to bit fixing is bit flipping [17].

Functional BIST is a new emerging trend. Traditional BIST solutions use special hardware for test pattern generation (TPG) on chip, but this may introduce significant area overhead and performance degradation. To overcome these problems, recently new methods have been proposed which exploit specific functional units such as arithmetic units or processor cores for on-chip test pattern generation and test response evaluation [18,19]. In particular, it has been shown that adders can be used as TPGs for pseudorandom patterns. But up to now there is no general method how to use arbitrary functional units for built-in TPG.

4.3 Output response analysis

In BIST, there is a need to reduce the enormous number of circuit responses to a manageable size that can be stored on the chip. The methods of reducing this information are response compaction and response compression. In *compression* the number of bits in response will be reduced with no loss of information. In *compaction* some information is lost. The loss of information is called *aliasing*.

Signature analysis [20] is a process of compacting the test responses into a very small bit length number, representing a statistical circuit property, for economical on-chip comparison of the behavior of a possible defective CUT with a good CUT. *Signatures* can be created from the data stream by feeding

it into the EXOR gate at the input of an n-bit LFSR (Fig. 5). After the data stream has been clocked through, a residue of the serial data is left in the shift register. This residue is unique to the data stream and represents its signature. To form the signature, the LFSR is first initialized to a known state.

An n-bit signature register can generate 2^n signatures. However, many input sequences can create the same signature. If the length of the data stream is m then 2^m sequences can map into 2^n signatures. Only one out of 2^m possible response streams is error-free and produces the correct signature. However, any of the remaining $2^{m-n} - 1$ sequences may also map into the correct signature. This mapping causes *aliasing*, i.e., masking of the fault present in the CUT. The probability that two input sequences will have the same signature is $P = (2^{m-n} - 1)/ 2^m - 1$ [20].

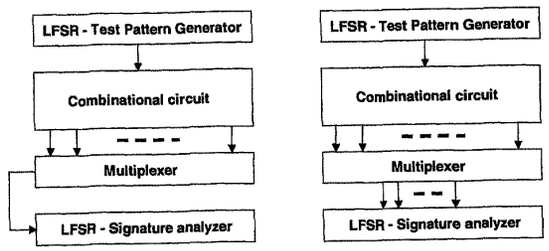


Figure 6. Single input and multiple input signature registers

In general, a CUT will have more outputs, and it produces more response sequences as the result of testing. Different possibilities are possible for space compaction. In Fig. 6a, responses from one output of the CUT at a time are compacted through the signature analyzer using a multiplexer. The procedure is slow, but the probability of aliasing is $1/2^n$. In Fig. 6b a *multiple input signature register* (MISR) is used. The inputs are feeded into EXOR gates (each input per gate) between the flip-flops of the LFSR. A k-bit MISR can compact an m ($m >> k$) – bit output sequence in m/k cycles. Thus, a MISR can be considered as a *parallel signature analyzer*.

4.4 Specific BIST architectures

Over the years several testing schemes which make use of pseudorandom test generators and signature analyzers (MISR) have been developed. Typically LBIST approaches combine these components with other DFT constructs like scan-path and Boundary-Scan. Some control circuitry is required to configure the circuit for normal operation and testing modes. The signature of MISR is checked automatically vs. the good CUT response,

which is stored in a ROM on the chip. The effectiveness of testing depends on the choice of LFSRs, their length, and configuration. It usually yields high fault coverage for combinational circuits [21].

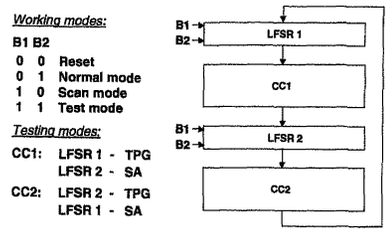


Figure 7. BILBO architecture

In the *Built-in Logic Block Observer* (BILBO), the scan-path technique is combined with BIST. It uses a multipurpose module, called BILBO, that can be configured to function as a test generator or a signature analyzer. Fig.7 shows the BILBO architecture for two cascaded combinational circuits CC1 and CC2. Control signals B1 and B2 are used for choosing the working modes for BILBO registers LFSR 1 and LFSR 2. For example, when testing CC1 LFSR 1 is used as test generator, and LFSR 2 is used as signature analyzer. For testing CC2, the LFSR registers will work in the opposite way.

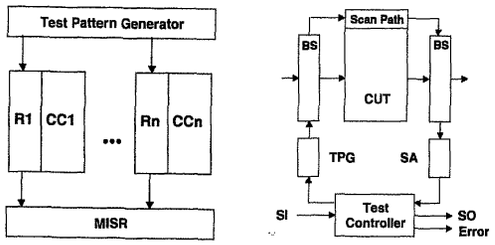


Figure 8. BIST architectures

Fig. 8a presents a BIST architecture called STUMPS (Self-Testing Using an MISR and Parallel Shift Register Sequence Generator). STUMPS uses multiple serial scan paths that are fed by a pseudo-random test generator [12]. Each scan corresponds to a part of the CUT. Once data has been loaded into the scan paths, the system clock is activated. The test results are loaded into the scan paths and then shifted into the MISR.

In Fig. 8b a BIST architecture called LOCST (LSSD On-Chip Self Test) is presented [22], where LSSD stands for Level-Sensitive Scan Design [11].

LOCST combines pseudorandom testing with LSSD-based scan designs. The inputs are applied and the outputs are obtained via boundary scan cells. The boundary scan cells and memory elements in the CUT form a scan path. Some of the memory elements at the beginning of the scan path are configured into an LFSR for generating pseudo-random patterns, and some of the memory elements at the end of the scan path are configured into another LFSR, which operates as a signature analyzer. The test process starts by serially loading the scan path with pseudorandom pattern generated by the LFSR. The pattern is applied to the combinational part of CUT, and the resulting output pattern is loaded in parallel into the scan path. These output bits are then shifted into the signature analyzer. After finishing the test process, the resulting signature is compared with the reference signature.

Circular BIST (Fig.9) is intended for register-based architectures [23]. The LFSR plays simultaneously the roles of the pseudo-random test generator and signature analyzer. The test process requires three phases: initialization of the flip-flops in the LFSR, testing of the CUT (during this phase the self-test path of connected flip-flops operates as both test generator and a MISR), and response evaluation where the content of LFSR is scanned out and compared with a precompared fault-free signature.

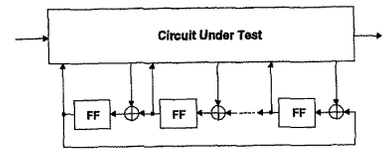


Figure 9. Circular Self-Test

To reduce the hardware overhead cost in the BIST applications, the LFSR can be implemented in software, which is especially attractive to test NoCs, because of the availability of computing resources directly in the system.

4.5 Hybrid BIST

In many cases the LFSR-based approach to on-chip test generation does not guarantee a sufficiently high fault coverage (especially in the case of large and complex designs) and demands very long test application times in addition to high area overheads. Therefore, several proposals have been made to combine on-line generated pseudorandom test patterns with prestored deterministic patterns [15-17, 24] to form a hybrid BIST solution.

In [25] an approach is proposed to use a hybrid test set, which contains a limited number of pseudorandom and deterministic vectors. The on-line

pseudorandom test can be generated either by hardware or by software. The test will be improved later by a prestored deterministic test set which is specially designed to shorten the pseudorandom test cycle and to target the random resistant faults. To reduce the hardware overhead cost the LFSR can be implemented in software, which is especially attractive to test NoCs, because of the availability of computing resources directly in the system.

Hardware-based hybrid BIST architecture is depicted in Fig.10. Pseudorandom pattern generator (PRPG) and MISR are implemented inside the core under test. Pregenerated deterministic patterns are stored in ROM.

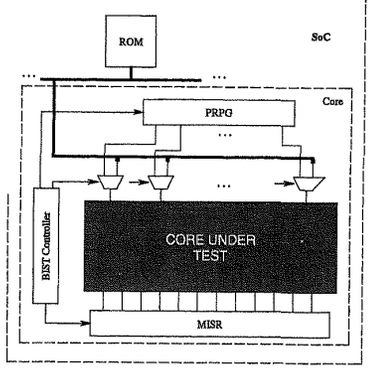


Figure 10. Hardware-based hybrid BIST architecture

In case of software-based solution, the test program, together with test data (LFSR polynomials, initial states, test length, deterministic test patterns, signatures) are kept in a ROM. In test mode the test program is executed in the processor core. The test program proceeds in two successive stages. In the first stage the pseudorandom test pattern generator, which emulates the LFSR, is executed. In the second stage the test program will apply precomputed deterministic test vectors to the core under test (CUT).

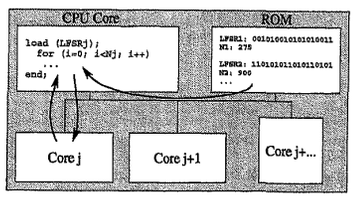


Figure 11. Software-based hybrid BIST architecture

The pseudorandom TPG software is the same for all cores in the system and is stored as one single copy. All characteristics of the LFSR needed for

emulation, are specific to each core and are stored in the ROM. They will be loaded upon request. Such an approach is very effective in the case of multiple cores, because for each additional core, only the BIST characteristics for this core have to be stored. The general concept of the software based pseudorandom TPG is depicted in Fig. 11.

The quality of the pseudorandom test is of great importance. It is supposed that for the hybrid BIST the best pseudorandom sequence (the shortest possible one with highest fault coverage) will be chosen. In case of hybrid BIST, we can dramatically reduce the length of the initial pseudorandom sequence by complementing it with deterministic stored test patterns, and achieve the 100% fault coverage.

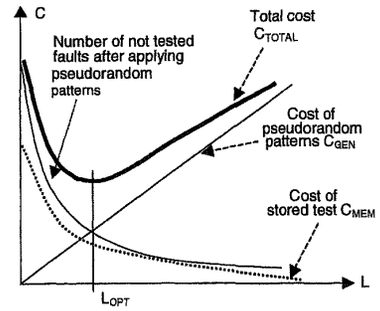


Figure 12. Cost calculation for hybrid BIST

The total test cost of the hybrid BIST C_{TOTAL} can be calculated as the sum of two costs:

$$C_{TOTAL} = C_{GEN} + C_{MEM} = \alpha L + \beta S$$

where C_{GEN} is the cost related to the time for generating L pseudorandom test patterns (number of clock cycles), C_{MEM} is related to the memory cost for storing S precomputed test patterns to improve the pseudorandom test set, and α, β are constants to map the test length and memory space to the costs of the two parts of the test to be mixed. To find the optimum total cost of the hybrid BIST, several algorithms have been developed [25].

4.6 Memory BIST

RAM memories are just one possibility of the types of resources in the nodes of a NoC. Due to their regular structure, efficient algorithms and self-test architectures have been developed for RAMs. Naturally, the BIST means additional silicon overhead. Kraus et al. [26] estimate that the on-chip BIST area overhead is about 1% for a 4 Mb DRAM. According to [27], a 2%

overhead could be expected. However, this area is relatively small, especially if we consider that the MBIST allows us to test RAM resources 2 to 3 orders of magnitude faster than conventional off-chip methods [2].

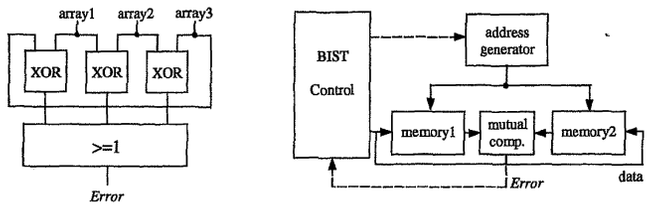


Figure 13. Mutual comparator and its application in MBIST

Several types of memory BIST can be distinguished according to the test strategies used. *Concurrent BIST* is a mechanism where the memory can be tested concurrently during normal operation. In *non-concurrent BIST* the normal operation has to be interrupted and, in general case, the original contents of RAM are lost. In *Transparent* RAM BIST the operation is also interupted but the original memory contents are preserved.

Important components of a memory BIST architecture are the *address* generator or stepper and the *data generator*. For the stepper, usually an LFSR is implemented since it uses substantially less area than the binary counter [2] and it can be made self-testable [28]. Moreover, LFSRs can be adjusted to generate the all-zero pattern and the forward and exact reverse sequences [2,12]. This type of generator satisfies all the ordering conditions for detecting address decoder faults using march tests.

In addition, a structure called *mutual comparator* [2] is useful in MBIST when the memory has multiple arrays. It is possible to test multiple arrays simultaneously by applying the same data and addresses to all of them. The comparator asserts the error signal when one of the array outputs disagrees with others. Thus, there is no need to generate the good machine response as it is assumed that only a minority of the memory array outputs are incorrect at any given time. Fig.13a presents a mutual comparator for a system consisting of 3 arrays and in Fig.13b a memory BIST scheme with 2 memories is given.

Different algorithms should be used in testing different types of RAM. SRAM memories can be tested by the march tests. However, for testing DRAMs neighborhood pattern sensitive fault model is more appropriate [2]. The latter is not capable of detecting address decoder faults while the march test is [27]. Thus, both test algorithms should be implemented in BIST hardware in case of testing DRAMs.

5. TESTING THE NOC ARCHITECTURE

5.1 Test reuse and test application issues

One of the most important concepts in SoC design is the reuse of cores. This is going to be the case also for NoC, where reusability of interconnections and software form additional requirements for a system of cores. Naturally, when we are talking of core reuse it should also include reusability of core test. The latter is a very difficult issue, both for the core provider and for the SoC test integrator. First, the core provider should support all the test tools and strategies that could possibly be used by a prospective core integrator. Second, the core test integrator is responsible for generating test for the system of cores. In the case of soft cores this task can be managed assuming that certain testability guidelines are followed by the core providers. However, in the case when the system contains hard cores or protected Intellectual Property (IP) it may be even impossible to apply conventional test tools to the cores in the system. In the case of hard cores the logic-level description may be unavailable. The protected IP cores are provided in an encrypted form and their implementation details are not accessible by the core integrator. These cores come with a pre-generated set of test patterns supplied by the core provider and the test integrator has the task then to apply the given tests at the inputs of the core in the system and observe its outputs. Thus, *test access mechanisms*, *isolation* and *test control* issues are crucial in testing systems composed of embedded cores.

Test Access Mechanisms (TAM) are the means for applying desired test stimuli to the inputs of a core and obtaining the response from the outputs of that core. TAMs are on-chip structures that are either normal interconnection architectures of the system or dedicated mechanisms for transporting test data to and from the core under test. The *source* of the test patterns can be on-chip or off-chip. The same applies to the destination of the output responses of the core under test, which is called the *sink*.

Test control, in turn, refers to switching between the normal and test modes of the core operation and activation and deactivation of the core testing functions. For example, mechanisms to activate/deactivate BIST and/or scan chains are referred to as test control. Embedded test processor cores are a more advanced example of test control mechanisms.

In some cases, isolation of cores may be required. By isolation we mean disconnecting the core inputs and outputs from other cores during the test in order to avoid damage to the surrounding cores and the user-defined logic. In the following we will explain the concepts of TAM, test control and isolation on the basis of the IEEE P1500 standard, which has been established in order to facilitate test access to embedded cores and core test knowledge transfer.

In addition to testing the cores in the resources of a NoC, the communication infrastructure, i.e. switches and wires, have to be tested. This topic is discussed in Subsection 5.3.

5.2 IEEE P1500 standard for core test

IEEE P1500 effort started in 1995 as a Technical Activity Committee (TAC) of the TTTC (Test Technology Technical Council) of the IEEE Computer Society [29]. The two most important components of the P1500 standard are *the core test language* (CTL) and *the scalable core test architecture*. In order to understand the concept of the P1500 standard, some basic definitions have to be introduced first. In the following, three components are listed that are generally required to test embedded cores.

1. A *source* for application of test stimuli and a *sink* for observing the responces.
2. *Test Access Mechanisms* (TAM) to move the test data from the source to the core inputs and from the core outputs to the sink.
3. A *wrapper* around the embedded core.

The wrapper allows switching between the normal functional mode and test modes of the core. Both, internal test and testing of the external interconnects of the core may be performed. Note, that the wrapper is specified in the P1500 standard, while TAM, source and sink are not. Fig. 14 explains the basic architecture for embedded core tests.

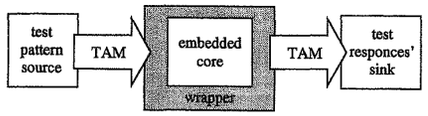


Figure 14. Test application scheme for embedded cores in P1500

The first part of the standard specifies the CTL, a language, which is an extension of IEEE Std. 1450.0 Standard test Interface Language (STIL) [29]. Its purpose is to standardize the core test knowledge transfer. A CTL file of a core must be supplied by the core provider. This file contains information on how to instanciate a wrapper, map core ports to wrapper ports, and reuse core test data.

The second part of the standard specifies the scalable core test architecture. It standardizes only the wrapper and the interface between the wrapper and TAM, called Wrapper Interface Port or (WIP). In fact, the P1500 TAM interface and wrapper can be viewed as an extension to IEEE Std. 1149.1, since the 1149.1 TAP controller is a P1500-compliant TAM interface, and the boundary-scan register is a P1500-compliant wrapper.

The P1500 wrapper must contain an instruction register (WIR), a wrapper boundary register consisting of wrapper cells, a bypass register and some additional logic. The wrapper must allow normal functional operation of the core plus it has to include a 1-bit serial TAM. In addition to the serial test access, parallel TAMs may be used. The motivation for providing both serial and parallel test access to the core, is because for the same core some system integrators might prefer serial access while some might need parallel access. The P1500 wrapper supports internal and external test functions.

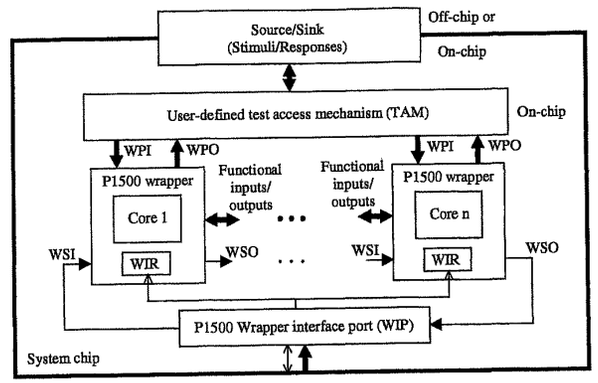


Figure 15. Scalable core test architecture

Fig. 15 presents the P1500 scalable core test architecture consisting of n cores. The test architecture is controlled by the Wrapper Interface Port (WIP). The wrapper interface controls and clocks the wrapper instruction (WIR) and bypass registers. The serial test data and wrapper instructions are shifted in and out via the Wrapper Serial In (WSI) and Wrapper Serial Out (WSO) ports, respectively. Parallel test access is allowed through the WPI and WPO busses. During normal operation of cores, the core functional inputs and outputs are used.

5.3 Testing the communication infrastructure

In addition to testing the cores, or resources of the NoC, the communication infrastructure should be dealed with as well. In this chapter, we consider a mesh-like topology of NoC consisting of switches (routers), wire connections between them and slots for SoC resources, also referred to as tiles. In fact, many other types of topological architectures, e.g. honeycomb and torus may be implemented and their choice depends on the constraints for low-power, area, speed, testability etc. [30].

Qualcomm Exhibit 1006

The resource can be a processor, memory, ASIC core etc. The network switch contains buffers, or queues, for the incoming data and the selection logic to determine the output direction, where the data is passed. In the example of a mesh-like network, the five possible directions are the upward, downward, leftward and rightward neighbours plus the resource direction. In the current Subsection we do not discuss the topic of testing the switch interconnection with the resource but assume that this is covered by the test of each resource.

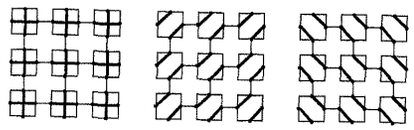


Figure 16. Three configurations to test an interconnect switch

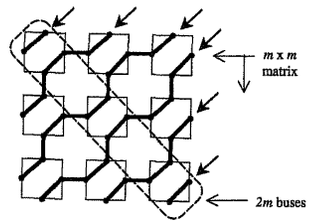


Figure 17. The concatenated bus concept

A lot of useful knowledge for the test of NoC network structures can be obtained from the interconnect testing of other regular topological structures. The test of wires and switches is to some extent analogous to testing the interconnects of an FPGA. In [31], it was shown that a switch in a mesh-like communication structure can be tested by using only three different configurations (See Fig. 16). It is also known that arbitrary short and open in an n-bit bus can be tested by $\log_2(n)$ test patterns [32]. When testing the NoC interconnects we can regard different paths through the interconnect structures as one single concatenated bus. Assuming that we have a NoC, whose mesh consists of m x m switches, we can view the test paths through the matrix as a wide bus of $2mn$ wires. Figure 17 explains the concatenated bus concept on a diagonal configuration. Note, that in the current approach stuck-at-0 and stuck-at-1 faults are modeled as shorts to Vdd and ground. Thus we need two extra wires, which makes the total bitwidth of the bus $2mn + 2$ wires. Finally, from the above facts we can find that $3[\log_2(2mn+2)]$ test patterns are needed in order to test the switches and the wiring in the

NoC [31]. Thus, due to the regularity of the wiring in NoC, the testing of its interconnections can be easily managed.

6. CONCLUSIONS

This Chapter provided an overview of test methods for NoC designs. First, the basic concepts of digital test were explained. We presented a general functional fault model to be used in different levels of the design hierarchy. Defect-oriented and hierarchical test generation approaches as promising trends in the deep-submicron era were discussed. We also proposed a new efficient method for hierarchical test generation of sequential circuits.

Due to the fact that BIST allows at-speed testing and simplifies test access to embedded cores, it has become a popular technique for testing the cores in SoC. Various architectures of logic BIST that can be applied in testing the NoC resources were discussed. Since RAM memories serve as one type of resources, memory BIST basics has also been explained.

The test of embedded cores in the NoC resources has been discussed. Important issues in core testing, including test reuse, isolation, test control and Test Access Mechanisms (TAM) were covered. The basics of P1500 standard for embedded core test were explained, including the Core Test Language (CTL) and the scalable core test architecture. The P1500 core wrapper, its functionality and interface to TAM was considered. Finally, testing of NoC interconnect switches and wires was discussed. We have shown that there is a strong correspondence between this task and the problem of testing the RAM-based FPGA interconnects. Thus, methods known from the field of FPGA testing can be applicable in the context of NoC interconnection test.

REFERENCES

[1] T. W. Williams and N. C. Brown, "Defect Level as a Function of Fault Coverage", *IEEE Trans. on Computers*, Vol. C-30, No. 12, pp. 987- 988, 1981.
[2] A.J. van de Goor. Testing Semiconductor Memories. J.Wiley & Sons, 1991, 512 p.
[3] S.M.Thatte, J.A.Abraham. Test Generation for Microprocessors. IEEE Trans. On Computers, 1980, Vol. C-29, No. 6, pp.429-441.
[4] P.Nigh,W.Maly. Layout-Driven Test Generation. Proc.1989 ICCAD, pp. 154-157, 1989.
[5] M.Jacomet, W.Guggenbuhl. Layout-Dependent Fault Analysis and Test Synthesis for CMOS Circuits. IEEE Trans. on CAD, vol. 12, pp. 888-899, 1993.
[6] R.Ubar, W.Kuzmicz, W.Pleskacz, J.Raik. Defect-Oriented Fault Simulation and Test Generation in Digital Circuits. Proc. ISQED, San Jose, March 26-28, 2001, pp.365-371.

[7] S.R.Rao, B.Y.Pan, J.R.Armstrong. Hierarchical Test Generation for VHDL Behavioral Models. EDAC, Feb. 1993, pp. 175-183.
[8] J.Raik, R.Ubar. Fast Test Pattern Generation for Sequential Circuits Using Decision Diagram Representations. J. of Electronic Testing: Theory and Applications. Kluwer Academic Publishers. Vol. 16, No. 3, pp. 213-226, 2000.
[9] E. M. Rudnick, J. H. Patel, G. S. Greenstein, T. M. Niermann, "Sequential-Circuit Test Generation in a Genetic Algorithm framework," Proc. DAC, pp. 698-704, 1994.
[10] T.M. Niermann, J.H. Patel, "HITEC: A test generation package for sequential circuits", *Proc. of EDAC*, 1991, pp.214-218.
[11] M. Abramovici et. al. Digital Systems Testing & Testable Designs. Computer Science Press, 1995, 653 p.
[12] P.Bardell, et. al. Built-In Test for VLSI Pseudorandom Techniques. Wiley & Sons, 1987.
[13] E.J. McCluskey. Logic Design Principles: With Emphasis on Testable Semicustom Circuits. *Prentice Hall*, 1986.
[14] E.B. Eichelberger, E.Lindbloom, J.A.Waicucauski, T.W. Williams. Structured Logic Testing. Prentice Hall, 1991, 183 p.
[15] S. Hellebrand, S.Tarnick, J.Rajski, B.Courtois. Generation of Vector Patterns through Receeding a Multiple-Polynomial LFSR. Proc. ITC, Baltimore, 1992, pp.120-129.
[16] N.A. Touba, E.J. McCluskey. Altering a Pseudorandom Bit Sequence for Scan-Based BIST. Proc. IEEE Int. Test Conference, 1996, pp.167-175.
[17] H.J. Wunderlich, G.Kiefer. Scan-Based BIST with Complete Fault Coverage and Low Hardware Overhead. Proc. IEEE European Test Workshop, pp.60-64.
[18] J. Rajski, J. Tyszer. Arithmetic BIST For Embedded Systems, Prentice-Hall, 1998.
[19] R. Dorsch, H.-J. Wunderlich. Accumulator Based Deterministic BIST, Proceedings IEEE International Test Conference, Washington, DC, October 1998, 412-421.
[20] R.A. Frohwerk. Signature Analysis: A New Digital Field Service Method. Hewlett-Packard Journal, 1977, Vol.28, No. 9, pp.2-8.
[21] S.Mourad, Y.Zorian. Principles of Testing Electronic Systems. J.Wiley & Sons, Inc. New York, 2000, 420 p.
[22] J.J. LeBlanc. LOCST: A Built-In Self-Test Technique. IEEE Design and Test of Computers, November, 1984, pp.42-52.
[23] A. Krasniewski, S.Pilarski. Circular self-test path: a low cost BIST technique for VLSI circuits. IEEE Trans. 1989, CAD, Vol. CAD-8, No. 1, pp.46-55.
[24] M. Chatterjee, D.K. Pradhan. A novel pattern generator for near-perfect fault coverage. VLSI Test Symposium, 1995, pp.417-425.
[25] G.Jervan, H.Kruus, Z.Peng, R.Ubar. About Cost Optimization of Hybrid BIST in Digital Systems. Proc. ISQED, San Jose, March 18-20, 2002, pp.273-279.
[26] R. Kraus, O. Kowarik, K. Hoffmann, D. Oberle, "Design for Test of Mbit RAMs", Proc. Of the International Test Conference, Aug. 1989, pp. 316-321.
[27] M.L.Bushnell, V.D.Agrawal. Essentials of Electronic testing. Kluwer Acad. Publishers, 2000, 690 p.
[28] M. Nicolaidis, "An Efficient Built-In Self-Test Scheme for Functional Test of Embedded RAMs", Proc. Of the IEEE Fault Tolerant Computer Systems Conf., 1985, pp. 118-123.
[29] http://grouper.ieee.org/groups/1500
[30] W. J. Dally, B. Towles, "Route packets, not wires: On-chip interconnection networks", Proceedings of the 38th Design Automation Conference, June 2001.
[31] M. Renovell, J.M. Portal, J. Figueras, Y. Zorian, "Testing the Interconnect of RAM-based FPGAs", IEEE Design & Test, January-March 1998, pp. 45-50.
[32] W.H. Kautz, "Testing for Faults in Wiring Networks", IEEE Trans. Computers, Vol. C-23, No. 4, 1974, pp. 358-363.

Chapter 8

Clocking Strategies for Networks-on-Chip

Johnny Öberg
Royal Institute of Technology, Sweden
johnny@imit.kth.se

Keywords: Network-on-Chip (NoC), Clock-distribution Networks, clock trees, clock skew, H-trees, Quasi-synchronous clocking, Mesochronous clocking, synchronization.

Abstract: One of the main problems when designing today's ASICs, is to distribute a skew-free synchronous clock over the whole chip. A large part of the power is also consumed in the clock-tree, in some cases as much as 50% of the total power consumption of the chip, because the large wires in the clock-tree are switched often. To attack these two problems, several methods have been discussed in the research literature over the years, from the more obvious solution of using asynchronous communication between locally clocked regions (Globally Asynchronous Locally Synchronous - GALS) to more fancy methods like distributing a standing wave on the clock-wires across the whole chip. In this chapter, we go through different clocking methods that has been proposed over the years and are suitable for the NoC scheme as well as presenting a new and clever way of distributing a Quasi-synchronous, i.e., a perfectly synchronous, but not skew-less, clock across an entire NoC-chip.

1. INTRODUCTION

In today's ASICs (Application-Specific Integrated Circuits), a large part of the power is consumed in the clock-tree, a typical figure being 25-30% of the total power consumption of today's chips [1]. The high fraction is caused by the fact that every synchronous digital system needs to have some kind of clock connected to every flip-flop and latch in the system to function properly. As the feature size of the transistors decrease, the clock frequency can be increased. However, the wire lengths do not shrink proportionally,

A. Jantsch and H. Tenhunen (eds.), Networks on Chip, 153–172.
© 2003 *Kluwer Academic Publishers. Printed in the Netherlands.*

Qualcomm Exhibit 1006

since we still have to communicate from one end of the chip to the other. As the digital systems grows bigger and bigger, it becomes harder and harder to distribute a synchronous clock over an entire chip.

Several methods for distributing a clock has been discussed in the research literature over the years [1], from the more obvious solution of using asynchronous communication between locally clocked regions (Globally Asynchronous Locally Synchronous - GALS) [2] to more fancy methods like distributing a standing wave on the clock-wires across the whole chip [3].

However, most of today's research is targeted towards reducing the clock-skew and jitter by improving current clock distribution methods, together with new and better de-skew circuits and improved noise filtering in order to reduce the jitter. The distance from the clock insertion point on the chip to where it is consumed at the latches causes a phase difference that must be compensated for in order to have synchronous communication with the external world. The clock distribution trees are so large and carry so much capacitance that buffers need to be inserted just to be able to drive the clock-tree in order to have a reasonable clock waveform. Process variations cause variances in the delay of the buffers. The capacitive load at the end of the clock tree is not constant but varies slightly. Variations in the length to the closest power/ground pad insertion point, cause a slight power supply mismatch. And, finally, differences in the operating temperature in different parts of the chip cause a variation in the device delays. These effects are cumulative and make it hard to design a clocking network in a proper way. Clock skew accounts for about 5% of the clock cycle time and is trending higher as the frequency increases [4]. In addition to these factors, different sources of noise cause the clock signals to jitter. The jitter is typically half the clock-skew [1].

For Network-on-Chips, on the other hand, other possibilities exists. Communication between resources is supposed to take place over the intercommunication network. Special resources, I/O nodes, are utilized to communicate with the external world. Computational resources need only to be synchronous with its network switch. Also, the regular structure of the NoC's equidistance switches and resources, makes it possible to exploit the clock-skew instead of fighting it, much in the same way it was exploited by Hataminian and Cash in the design of regular multiplier arrays in the middle of the 80's [5, 6]. This Quasi-synchronous method is similar to the Globally Updated Mesochronous (GUM) Design Style, as proposed in [16], with the difference that the phase difference between the synchronous blocks is (almost) constant across the chip.

The rest of this chapter is organized as follows. In Section 2, an overview and classification of different signal clock synchronization methods is given.

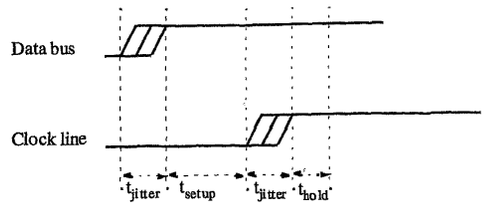


Figure 8.1. Latest allowed safe arrival for data signals w.r.t the clock.

In Section 3, an overview over the most common clock distribution networks is given. Clock-distribution networks for NoCs are discussed and two possible non-synchronous clock distribution methods are introduced in Section 4.The Quasi-synchronous clocking used for the NoC and the differences with its relative Mesochronous clocking is described in detail in Sections 5 and 6, respectively. In Section 7, the physical properties of the Quasi-synchronous clocking is investigated, and especially how it improves the on-chip power consumption by reducing the peak-current. Finally, in Section 8, a summary of this chapter is given and some conclusions are drawn.

2. SIGNAL-CLOCK SYNCHRONIZATION METHODS

Signal-clock synchronization methods can be divided into five classes [7], see Table 1 below: Synchronous, Mesochronous, Plesiochronous, Periodic, and Asynchronous signal-clock synchronization.

In the Synchronous case, a global clock can be distributed to synchronize the whole chip. The clock signal arrives simultaneously at the local flip-flops used to latch data all-over the chip. However, making the clock arrive simultaneously all-over the chip is not possible in practice. Process variations together with small differences in the length and load of single clock-wires causes a small spread in clock arrival, called the clock skew. In the synchronous case, special care is taken in order to make the clock-skew as close to zero as possible, or at least as low as to guarantee a safe transition of the signal. Ideally, the data arrives just in time to meet the setup-requirement for the local flip-flop, see Figure 8.1.

In the Mesochronous case, no special care is taken to minimize the clock skew. All signal data is synchronized with the local clock. The local clock is derived from a global clock that has been distributed all-over the chip, paying no heed to the signal-clock arrival times at different parts of the chip.

Table 1: Classification of Signal-Clock Synchronization [7]

Classification	Periodic	$\Delta\phi$	Δf	Description
Synchronous	Yes	0	0	A signal is synchronous w.r.t. the clock if it has the same frequency as the clock and is in phase with the clock. It is safe to sample the signal directly with the clock.
Mesochronous	Yes	ϕ_C	0	A signal is mesochronous wrt to the clock if the signal has the same frequency as the clock but it is potentially out of phase, with an arbitrary difference, ϕ_C. It is safe to sample the signal if the clock or signal is delayed by a constant amount.
Plesiochronous	Yes	Varies	$f_d<\varepsilon$	A signal is Plesiochronous w.r.t. the clock if it has a frequency that is nearly at the same frequency as the clock. It is safe to sample the signal if the clock or signal is delayed by a variable amount. The difference in frequency can lead to dropped or duplicated data.
Periodic	Yes	Periodic	$f_d>\varepsilon$	A signal is Periodic w.r.t. the clock is it is periodic at an arbitrary frequency. The periodic nature of the signal can be exploited to predict in advance which events may occur during an unsafe portion of the clock.
Asynchronous	No			A signal is Asynchronous w.r.t. a clock if the signal events may occur at arbitrary times. A full synchronizer is required.

All that is necessary is to adjust the sampling, by delaying either the local clock or the arriving signal, to keep signal transitions from unsafe regions of the local clock.

In the Plesiochronous case, clock signals are produced locally. The local clock is almost at the same frequency as clocks produced elsewhere, causing a small frequency drift. Special care has to be taken to avoid losing or inserting data when the cumulative drift comes close to a full clock period. Local PLLs and DLLs can be used to compensate for the frequency drift to ensure a safe transition of incoming signals.

In the Periodic case, the signal itself is periodic, just as the clock, because of coding or properties of the data. The difference in phase between the signal and the clock then also becomes periodic, with the difference in frequency $f_\phi = |f_s - f_c|$. The periodicity of this phase difference can then be

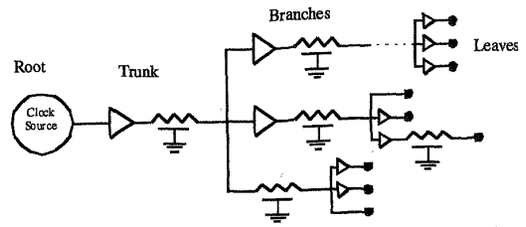


Figure 8.2. Tree structure of a general Clock Distribution Network

used to predict if the signal transition takes place during a safe or an unsafe portion of the clock.

In the Asynchronous case, there is no relation between the sending an receiving clock. In fact, in the fully asynchronous case, a clock doesn't have to be present at all. Signals arrive arbitrary. Asynchronous communication protocols, for example two- or four-phase handshaking, can be used to synchronize between two regions with different clock speed. However, care must be taken to avoid metastability in the sampling latches.

3. CLOCK-DISTRIBUTION NETWORKS

Skew-minimization is the target of almost every clock-distribution network. A general clock-distribution network, also called a clock-tree because of its similarity with a tree, is shown in Figure 8.2. Clock buffers are placed strategically along the clock wires, in the trunk near the root, near the trunk in the branches, and near the branch in the leaf nodes, in order to boost the signals and improve the wire-delay. The clock wires are modelled as distributed RC-lines - or RLC lines at GHz frequencies. Even if the distance from the source to each latch or flip-flop in the design would be made equal, some skew caused by load, temperature, and device mismatch would remain. To get around this, de-skewing techniques can be utilized. The common denominator in these de-skewing techniques is a phase-comparator that compares a reference phase with the actual phase of the clock signal and drives a voltage controlled delay line, more or less a clock-buffer with tunable delay, to adjust for the phase difference.

3.1 Local clock-skew tailoring

In some cases, achieving a zero-skew would actually lead to a sub-optimal solution. In these cases, speed can be improved by inserting a local delay in order to retime the signal-clock skew. Such an example is shown in

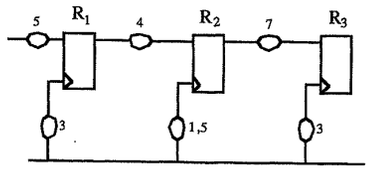


Figure 8.3. Inserting local delay to retime signal-clock skew in data paths.

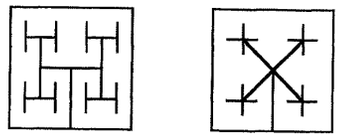


Figure 8.4. Symmetric distribution networks. a) H-tree. b) X-tree.

Figure 8.3. If the delay of the register is 2 ns, and the propagation delay through the logic between register R1 to R2 is 4 ns, and the propagation delay through the logic between register R2 and R2 is 7 ns, the critical path between the two pairs of registers is 6 ns and 9 ns, respectively. A zero-skew circuit would then have to be clock with a clock-period of 9 ns. However, if we borrow 1.5 ns by inserting a negative clock skew of 1.5 ns, i.e., let the clock arrive 1.5 ns early, for register R2, a faster circuit with a 7.5 ns clock period can be achieved. The first path would then just meet the clock constraint with its critical path of 6 ns, since 1.5 ns has to be deducted from the clock period of that path, and the second path would have an additional 1.5 ns to complete is execution, just meeting the critical path of 9 ns for the second path.

3.2 Symmetric distribution networks

A fully symmetric clock distribution network has, before load effects and device and power mismatches has been taken account, equal distance from the clock source in the root to the latches in the leaves of the clock tree, and hence a zero clock-skew. The two most common symmetric clock trees are the H-tree and the X-tree, both shown in Figure 8.4.

To improve the driving capability and improve on the timing, clock buffers can be inserted into the clock-trees. Also, to minimize reflections on branching points in the tree, the clock line impedances can be matched to improve on the timing. One example of this is the Tapered H-tree, shown in Figure 8.5 a). Another way to improve the timing properties is to connect the clock lines from different branches together. This Mesh, shown in Figure 8.5 b), evens out the statistical spread among the branches of the tree since and

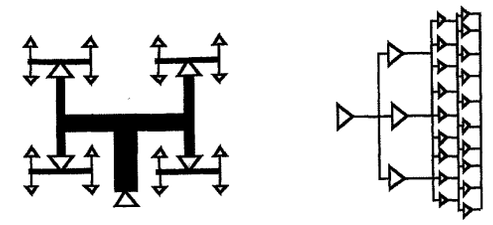


Figure 8.5. Common buffered trees. a) Tapered H-tree. b) Mesh

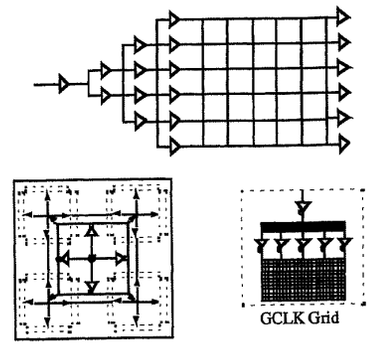


Figure 8.6. DEC Alpha Clock distribution. a) 21064 b) 21264

also improves on the delay, since the output impedances are connected in parallel.

3.3 Example Clock networks

In the optimization of the clock networks in most modern computers, a well-balanced fully synchronized clock-tree with as little skew as possible, after local skew-retiming, is the target. This can be achieved in several ways, as can be seen in the following examples.

In the DEC Alpha 21064, a centrally placed 200 Mhz clock is distributed through five levels of buffering, see Figure 8.6. After the fourth level, a metal mesh is placed in order to minimize any skew arisen from the first four levels in the tree. The fifth layer is used to clock the local registers [8, 9, 10]. In the DEC Alpha 21164, the single bank of clock drivers was partitioned into two halves in order to reduce the thermal effects across the chip. This method was exploited further in the DEC Alpha 21264, where a two-phase gridded global clock (GCLK) was distributed across the chip. In the centre

Qualcomm Exhibit 1006

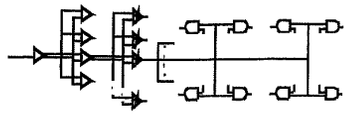


Figure 8.7. Itanium 2 Processor Clock Distribution.

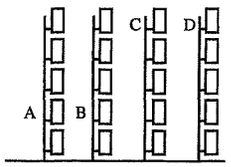


Figure 8.8. Hataminian's and Cash's Clock distribution network of a multiplier array. The difference in skew between A and B and between C and D is almost the same.

trunk of the tree, an on-chip PLL was used to generate the clock. The clock was then distributed using an X-tree to 16 local clock drivers, which in turn are connected to the GCLK grid [11, 12].

In the Itanium 2 processor clock distribution, see *Figure 8.7*, the primary driver is connected to five repeaters, through a pseudo-differential, impedance matched balanced H-tree. Each repeater in the first stage is connected to 33 adjustable delay buffers, configured as a balanced H-tree, where the widths and lengths has been tuned. Each second-level buffer is connected to roughly 70 tap points consisting of 8 gaters each, where each clock gater has a load-tuned drive-strength [13].

An example of an older configuration, is the one presented by Hataminian and Cash in [5, 6], where they present a clocking scheme for a highly regular structure, namely Multiplier Array Networks. The structure is very interesting from a Network-on-Chip point of view, since the Hataminian-Cash structure exhibits the same regular structure as a Network-on-Chip does.This will be covered in detail in the ensuing sections.

In Hataminian's and Cash's network, data is flowing from the left to the right, see Figure 8.8. They noted that for very regular structures, the skew can be divided into one horizontal and one vertical component. If the vertical clock lines are placed equidistant from each other, the horizontal difference in skew between two neighbouring vertical lines becomes close to constant. Furthermore, the horizontal skew between two neighbouring nodes on different vertical heights also becomes close to a constant. This means that the difference in skew between the nodes A and B and the skew between the nodes C and D in Figure 8.8., is almost the same.

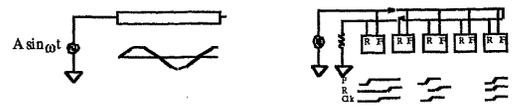


Figure 8.9. Travelling wave clocking a) Salphasic clocking. b) Round-trip clocking.

3.4 Future networks

In [3], Chi presented an approach named Salphasic clocking, where a standing wave is distributed across the chip, see Figure 8.9. If the clock lines are not too lossy, the forward and the backward travelling wave will match each other in amplitude, creating a standing wave, with the same frequency as the clock signal and without any phase errors. Local clock signals can then be connected anywhere along the transmission line, except of course at null points. The only thing required is to amplify the signal so that it can drive the local clock buffers. However, if the amplitudes do not fully match, a small, predictable, phase error will occur.

Round-trip clock distribution is another way of distributing a clock signal, which in principle is similar to the Salphasic clocking. A forward travelling clock is sent through a transmission line. When it is reflected at the end point, it is sent back travelling along the same set of modules as it passed on its way forward, although this time, the modules are passed in reverse order. At each point along the wire, the average time of arrival of the forward and reverse clock edges correspond to a clock signal without phase error. As long as the amplitudes of the two waves are identical, then a phase free clock can be derived simply by summing the two signals together. The clock signal can then be fed to the modules after buffering.

In [14], Wood et al. presented a transmission line based, self-generating rotary clock generation. The method is similar to the round-trip clocking, with the addition that the transmission line is closed in upon itself, through an inverter, similar to the way a ring-oscillator is built. The inverter is placed at a cross-point, where another inverter is running a second round-trip transmission line in the other direction. The cross-point connects the input of one inverter to the output of the other and vice versa. At all points along the two lines, a clock signal will be present. The clock's frequency will then be a property of the length of the transmission line, while the phase is determined by the tapping points place along the line in relation to the place of the inverters.

Another possibility would be use an optical clock distribution method, as the one presented by [15].The advantage of this method would be that

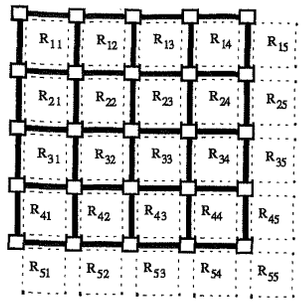


Figure 8.10. An example Network-on-Chip, with 5x5 resources connected together via a 5x5 network of switches.

optical transmission is immune to process variations, power-grid noise and temperature.

4. CLOCK DISTRIBUTION FOR NOCS

In a Network-on-Chip, the Resources performing the functionality, and the Switches, performing the inter-resource communication, are placed at equidistant spacing across the entire chip. The basic idea is to ease the design process by providing designers with a standardized interface and a standardized protocol for inter-resource communication. An example of a 5x5 NoC is shown in *Figure 8.10.*

Distributing a clock for such a Network-on-Chip can be done in many different ways. All of the clock distribution methods present in the previous section can be adapted to provide chip-wide synchronization also on a NoC. However, because of the regular structure, and the fact that only the special I/O resources utilized to communicate with the external world need to be in-sync/in-phase with the external world, suggest that an adaption of the method proposed by Hataminian and Cash [5, 6] to the NoC concept is the right way to go.

The starting point of such a scheme would be the structure proposed by Hataminian and Cash themselves, as shown in Figure 8.11, where the clock is either generated or entered onto the chip in the upper left corner of the Noc. However, there will be one significant difference between the two structures. In the Hataminian-Cash distribution, data was only flowing in one direction. In the NoC case, data will be flowing in four directions, North, South, West and East.

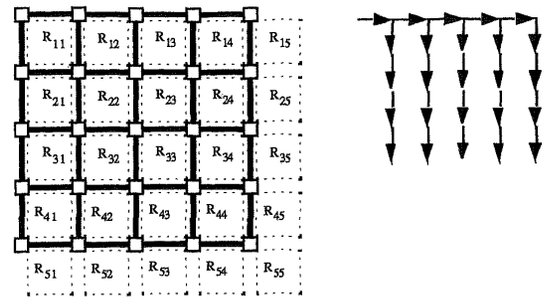


Figure 8.11. The waterfall distribution method with clock entry/source in the upper left corner, with a horizontally distributed waterfall.

In the next section, Section 5, a novel method of distributing a Quasi-synchronous clock, i.e., a synchronous clock with the same frequency but with a constant phase difference, across the entire Network-on-Chip is presented. The basic idea is to divide the chip into clock-regions, where the difference in arrival time of the clock signal (the skew) between any two neighbouring clock regions can be controlled and/or calculated beforehand. Just as with the Hataminian-Cash distribution, the clock is then running horizontally like a stream and at equidistance spacing, where a switch is placed, falling down the vertical lines like a waterfall falling of a cliff (although their structure was "falling" upwards).

The Quasi-synchronous distribution should not be confused with a Mesochronous distribution, i.e., a synchronous clock with the same frequency but with a random, static, phase difference, or with a fully synchronous distribution, where the clock has the same frequency and zero phase difference. The basic idea with Mesochronous clocking is to just to distribute the clock to each clock region, without any regard to the clock phase. However, special flip-flops with delay lines, either on the clock or the data lines, has to be used to ensure that data is clocked on a safe portion of the clock in order to avoid metastability problems. Mesochronous clock distribution for NoCs is covered in Section 6.

5. QUASI-SYNCHRONOUS CLOCK DISTRIBUTION

In Quasi-Synchronous Clocking, the clock signal is buffered and distributed to the nearest neighbours directly from the point where the clock enters the chip. Upon reaching the nearest neighbour, the clock is buffered

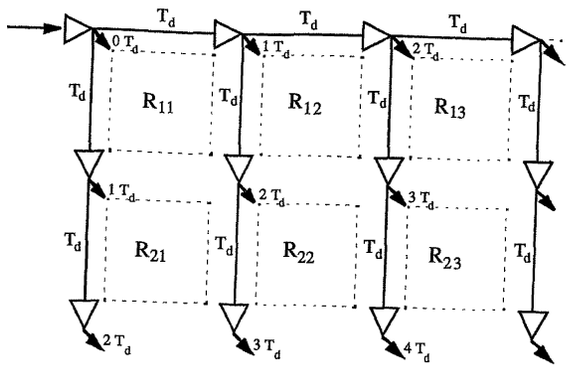


Figure 8.12. The difference in skew between any two neighbouring regions become 1 T_d.

again and fed to the local clock region. In one direction across the chip (horizontally in the *Figure 8.11*) the signal is buffered and split again, in the other (vertically in the figure), the clock signal is buffered and forwarded in the incoming direction. Hence, the clock is running horizontally like a stream and at equidistance spacing, where a switch is placed, the clock is falling down the vertical lines like a waterfall falling of a cliff. The procedure is repeated until all clock regions on the chip are connected. Each clock-region can then be further divided into sub-regions, and so on, until an appropriate terminating leaf-tree (depending on the application) is reached and be terminated with a local H-tree or X-tree.

Example. A Network-on-Chip is divided into 25 clock regions, as shown in *Figure 8.11*. The clock enters the chip in the upper left corner, and is distributed according to the waterfall distribution model. At each junction, a buffer is inserted. That buffer clocks the local switch, and the clock region R11. The clock is then distributed to the switches in the neighbouring regions R12 and R21. The delay between R11 and R12 and R21 respectively will then be 1 Td (which equal the buffer delays along the clock line plus time of flight). The clock coming into R12 is then buffered again, divided and distributed to R22 and R13, while the incoming clock to R21 is only buffered and then distributed to R31. The delay between any two neighbouring clock regions will then always be 1 Td, see Figure 8.12.

5.1 A relativity theory for clocking on a NoC

The difference in the clock-arrival time between any two neighbouring clock-regions must be taken into account when a signal should be sent from

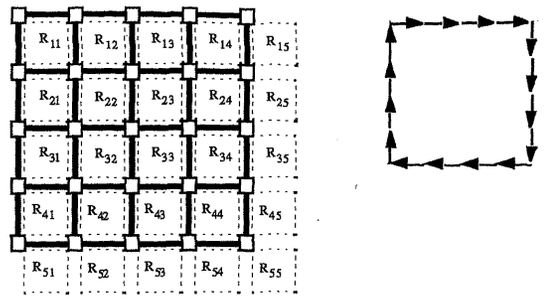


Figure 8.13. Accumulation of clock phase along the NoC. Data is adding a relative phase of + 1 Td travelling down-stream and subtracting - 1 Td travelling up-stream.

one clock-region to another. A signal travelling down the clock-stream will get a little extra time, +1 Td/clock region difference, since it is travelling along with the clock signal, before it has to be latched/stored again. A signal travelling against the stream will get a little less time, -1 Td/clock difference, since it is travelling against the direction of the clock signal.

Example. A datagram is travelling from the upper left corner (NW) of the switch to the lower right (SE) along the path outlined in Figure 8.13. For each step in the Switch Network, the datagram will accumulate a phase difference of + 1 Td, since it is always travelling down-stream. Upon reaching the destination, the datagram will be at a relative clocking phase of the origin of + 8 Td. Another datagram is sent as an answer by the receiving resource to the first sender. For each step in the Switch Network, the datagram will accumulate a phase difference of - 1 Td, since it is always travelling up-stream. Upon reaching the origin, the datagram will have accumulated a total clocking phase of (+8-8=0) Td. Thus, the received datagram will be back in phase with the origin.

5.2 Deriving Communication Timing Constraints

Data travelling downstream will be time-delayed with + 1 Td for every switch. This difference in arrival time must be carefully tuned with the delays on the interconnect buses. Since down-stream travelling is equivalent to wave-pipelining of the databus, one important constraint for down-stream travelling is that a signal travelling along the clock wave must not arrive before the clock signal does. In fact, data must not arrive before a safe transfer of the previous data on the bus has been latched into the down-stream switch, see Figure 8.14.

Qualcomm Exhibit 1006

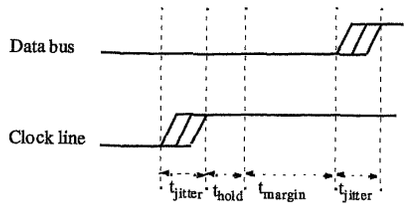


Figure 8.14. Earliest allowed arrival for down-stream travelling data.

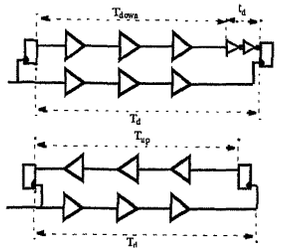


Figure 8.15. Up-stream and down-stream travelling of data.

Data travelling down-stream on the buses must be delayed relative to the clock. This is not a serious constraint, since data travelling down-stream will get an extra +1 Td to arrive at the next down-stream switch, see *Figure 8.15.* Data buses for up-stream travelling are easier to design, since up-stream data is latched on the receiving side first. Thus, data will always be transferred on the receiving side in a safe way. However, data must also arrive - 1 Td faster than the clock-period. One of the advantages with the NoC-structure is that data is latched at both the sending an receiving side of the switch. Thus, the whole clock-period can be used for transferring a signal from one switch to another. The timing constraint that has to be obeyed are summarized by the following equations:

$$t_d > t_{hold} + 2t_{jitter} \quad (1)$$

$$T_{period} + T_d > T_{down} + t_d \quad (2)$$

$$T_{period} - T_d > T_{up} \quad (3)$$

where t_{jitter} is the time during which a signal is jittering because of noise, and t_{hold} is the time needed for data to be stable after the clock edge in order to ensure a safe sampling.

From these equations it is clear that it is the up-stream data link that will limit the maximum frequency on the data buses. If the clock line is designed as a normal data bus line and up- and down-stream data lines are designed in the same manner, one additional constraint can be added:

$$T_{down} = T_{up} = T_d + t_{clk2q} + t_{hold} + 2t_{jitter} \quad (4)$$

where t_{clk2q} is the time it takes for data to appear on the output of the latches driving the data bus lines. Then, equation (3) can be rewritten into

$$T_{period} > 2T_d + t_{clk2q} + t_{hold} + 2t_{jitter} \quad (5)$$

from which the maximum achievable speed on the data buses can be derived. As long as the clock period is slow enough and the extra delay on the down-stream data lines is long enough, this method is not sensitive to process variations.

6. FULLY MESOCHRONOUS CLOCK DISTRIBUTION

Because of small variations in the manufacturing process, there will be a small spread in the exact delay of each clock buffer. This spread can be countered by introducing phase-compensation circuitry on the clock delay lines. All in all, this compensation circuitry should be as effective to fight skew as within a normal clock tree, i.e. the maximum skew can be held within 5% of the clock period.

Instead of fighting the delay spread, a fully mesochronous clocking distribution can be used [16]. The advantage with this approach would be to eliminate the skew-dependence totally. A fully mesochronous clocking scheme is insensitive to phase-variances along the clock line and can therefore be scaled indefinitely. However, special measures has to be taken that the latching flip-flops on the receiver side do not enter a metastable state. This is done by adding a small tunable delay on the clock lines between the network switches. The delay is set during a reset or initialization sequence in order to avoid data being latched in the failure-zone, i.e., a setup or hold-time violation, of the latching flip-flops [17].

Although a fully mesochronous approach sounds appealing at first thought, it introduces some problems of its own on system level. One of the objectives on the system-level is to provide the user with a predictable communication channel [18], where the delay on a channel is known beforehand. Introducing a fully Mesochronous would result in a Time

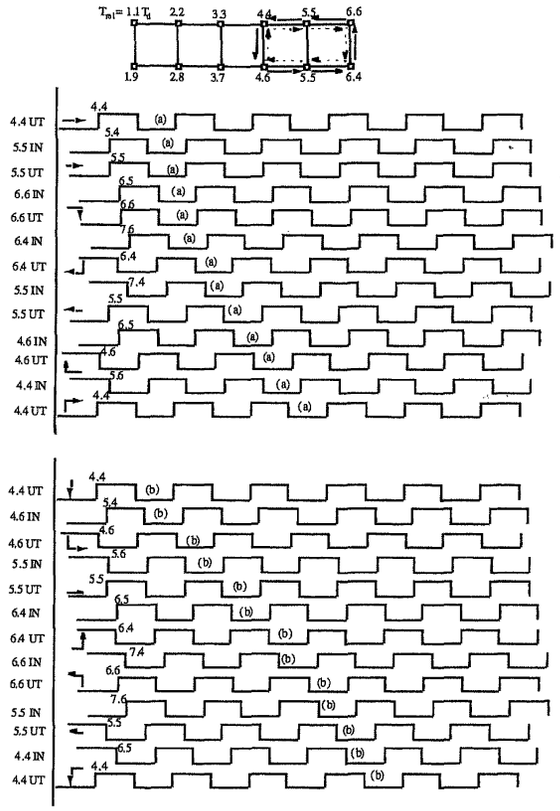


Figure 8.16. Simulation of Time dilation/whirl effects using a Fully Mesochronous clock distribution. a) Clock-wise going message. b) Counterclockwise going message.

dilation or Whirl-stream effect on the channels, i.e., a message travelling clock-wise or counterclockwise down-stream or up-stream would arrive at different points in time.

This effect is seen in *Figure 8.16*, where a simulation of two messages going around in a circle is shown. Message (a) is sent counterclockwise and message (b) is sent clockwise through a mesochronously clocked network. The delay for data travelling between the nodes is assumed to be 1 Td. The values shown is the nodes relative phase compared to the clock source/origin. It is assumed that the clock arrives at the top row with a +1.1 Td between the network nodes and that the clock arrives at the lower row

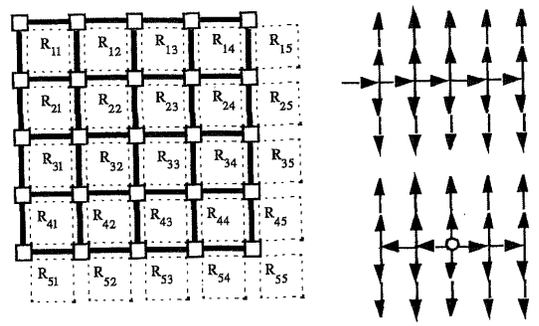


Figure 8.17. The waterfall method with clock entry/source placed as a) on the middle of one of the chip sides, b) in the centre of the chip.

with +0.9 delay between the network nodes. Also, the internal delay of one clock cycle that a message spends inside each switch has been removed to enhance readability of the figure.

Another action that can be taken to improve the Quasi-synchronous approach is to move the point where the clock is entered onto the chip. If the insertion point of the clock is moved to centre of one chip side, twice as many Switches can be used along that side before any compensation circuitry is needed. Then the waterfall is flowing in two directions, both upwards and downwards along the horizontal clock streamline. By moving the clock source/entry point to the middle of the chip, and let four waterfalls branch off, one in each direction (North, South, East and West), four times the number of Switches can be used before any compensation circuitry is needed. In this case, the clocking wave spreads like rings on the water across the chip. See Figure 8.17.

7. PHYSICAL PROPERTIES OF THE WATERFALL DISTRIBUTION

The power consumption of a wire on a CMOS-process is $P=0.5C_wV^2_{DD}f$, where C_w is the capacitance of the wire, V_{DD} is the driving voltage, and f the frequency at which the capacitance is switching). The NoC interconnection switch network is being clocked using a waterfall clocking distribution, where the clock lines are distributed with the normal data bus. This means that the C_w of the clock line is low, since a normal bus line can be used to transfer the clock. Thus, a waterfall distribution tree would consume less

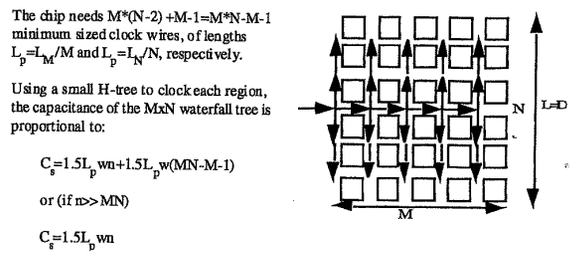


Figure 8.18. A chip divided into MxN clock regions.

power than any other distribution method that does not distribute the clock using a normal bus line.

Since the resources should be run synchronously with the network interface, the clock-trees inside a resource are best implemented using a symmetrical H- or X-tree.

If a chip is divided into MxN clock regions, as shown in Figure 8.18, it needs *M*N-M-1* minimum sized clock wires. If *M=N*, and small H-trees are used for clock distribution inside the resources, the waterfall model can be compared with a well-balanced H-tree. The H-tree will have a length of *1.5D*, where *D* is the side of the chip, which means that its capacitance is proportional as *1.5Dwn*, where *w* is the width of a minimum sized wire and *n* is the number of end nodes on the chip. The same chip using a waterfall-tree will have a capacitance proportional to $1.5L_pwn$ where L_p is *D/N*. The power consumption of the waterfall model can then be written as a function of the power consumption of the H-tree: $P_W=P_H/N$.

The waterfall distribution scheme also reduces the peak current of the chip. Not all gates in a logic module are switching simultaneously, rather, a wave of current consumption is swept across the logic module as some of the CMOS gates change value due the change on the inputs of the module. A good approximation of this current profile is to assume a triangular pattern distributed over several stages [7].

In a fully synchronous NoC, all modules in all resources would switch simultaneously. Hence, the peak current consumption would be proportional to the peak current consumption inside a module multiplied with the number of modules per resource times the number of resources inside the NoC. In an NxN Noc, this means a power consumption of $I_{peak}=I_{module}*N_{modules}*N*N$.

In the Quasi-synchronous NoC, the modules inside the resources that has the same relative time delay switch simultaneously. Then, the maximum number of simultaneously switching resources is *2(N-1)* instead of N^2, if the clock is inserted in the centre of the chip. The peak current consumption then reduces to $I_{peak}=I_{module}*N_{modules}*2(N-1) \approx 2I_{peaksync}/N$. If the clock would be

entered in the upper left corner, or on the middle of one chip side, the maximum number of simultaneously switching resources would be N, which would give a power consumption of $I_{peaksync}/N$. A fully Mesochronous NoC, the power consumption could be spread evened out further, but also go up, depending on how many of the synchronously clocked resource would end up having the same relative phase.

Another advantage with these clocking schemes is that all the gates that are not switching at the same time, contributes with half their load capacitance to the decoupling capacitance. Thus, less bypass capacitance is needed inside the resources, releasing area that can be used for more logic.

8. SUMMARY

In this Chapter, an overview over common existing clock distribution methods has been presented together with a new and novel clock distribution method, well suited for a Network-on-Chip. A Quasi-synchronous clock is distributed across the Intercommunication network together with the data lines. The distribution resembles a waterfall, where two horizontal clock-streams, generated at the centre of the chip, are branching off in the vertical (North and South) directions at each switch junction. The difference in clock phase between two neighbouring then becomes close to a constant, and data can safely be sampled. Data travelling down-stream accumulates a positive relative phase and data travelling up-stream accumulates a negative relative phase.

The clock distribution scheme also reduces the power consumption, since the clock lines are distributed using normal, thinner, bus wires along with the data buses. The only requirement is that data travelling downstream is delayed slightly in order to avoid meta-stability, i.e., to avoid that the clock and the data signal arrives simultaneously to the data-register. Furthermore, the Quasi-synchronous clocking scheme also reduces the peak current consumption, since the gate switching in the module is spread out over a larger period of the clock cycle.

An alternative to Quasi-synchronous clocking would be to implement a fully Mesochronous clocking. Mesochronous clocking is insensitive to skew and can therefore be scaled indefinitely. However, messages sent clock-wise or counterclockwise through the NoC could end up having different arrival times. This property is not so good from a system-design point of view, since one of the objectives on system-level is to provide the user with a predictable communication channel, where the delay on a channel is known beforehand.

REFERENCES

[1] E. G. Friedman, "Clock Distribution Networks in Synchronous Digital Integrated Circuits", In Proceedings of the IEEE, Vol. 89, No. 5, May 2001.

[2] A. Hemani, T. Meincke, S. Kumar, A. Postula, T. Olsson, P. Nilsson, J. Öberg, P. Ellervee, D. Lindqvist, "Lowering Power Consumption in Clock by Using Globally Asynchronous, Locally Synchronous Design Style", In the Proc. of DAC'99 (36th DAC), pp. 873-878, New Orleans, LA, USA, June 21-25, 1999.

[3] V. Chi, "Salphasic Distribution of Clock Signals for Synchronous Systems", IEEE Trans. on Computers, Vol. 43, No. 5, May 1994, pp. 597-602.

[4] S. Rusu, "Trends and Challanges in Clock Generation and Distribution", Esscirc Workshop Tutorial presentation, Esscirc 2002, Florence, Italy.

[5] M. Hataminian, and G. Cash, "A 70-MHz 8 bit x 8 bit parallel pipelined multiplier in 2.5 mm CMOS", IEEE Journal on Solid-State Circuits, Vol. SC-21, pp. 505-513, Aug. 1986.

[6] M. Hataminian, and G. Cash, "High speed signal processing, pipelining, and VLSI", in Proc. of IEEE International Conference on Acoustics, Speech, and Signal Processing, pp. 1173-1176, April 1986.

[7] W. J. Dally, J. Poulton, "Digital Systems Engineering", Cambridge University Press, 1998, ISBN 0-521-59292-5.

[8] M. Horowitz, "Clocking strategies in high performance processors", In Proc. of IEEE Symposium of VLSI Circuits, pp. 50-53, June 1992.

[9] D. W. Dobberpuhl et al., "A 200-MHz 64-b dual-issue CMOS microprocessor", IEEE Journal on Solid-State Circuits, vol. 27., pp. 1555-1565, Nov. 1992.

[10] J. Montanaro et al., "A 160-MHz, 32-b, 0.5-W CMOS RISC microprocessor", Digital Tech. Journal, Vol. 9, No. 1, pp. 49-62, 1997.

[11] P.E. Gronowski et al., "High-performance microprocessor design", IEEE Journal on Solid-State Circuits, Vol. 33., pp. 676-686, May 1998.

[12] D.W. Bailey and B.J. Benschneider, "Clocking design and analysis for a 600-MHz alpha microprocessor", IEEE Journal on Solid-State Circuits, Vol. 3, pp. 1627-1633, Nov. 1998.

[13] F. E. Anderson, et al., "The Core Clock System on the Next Generation Itanium Microprocessor", ISSCC Technical Digest of papers, 2002.

[14] J. Wood, et al., "Multi-gigahertz low-power low-skew rotary clock scheme", ISSCC Digest of Technical papers, pp. 400-401, 2001.

[15] A.V. Mule, S. M. Schultz, T. K. Gaylord, J. D. Meindl, "An Optical Clock Distribution Network for Gigascale Integration" In Proc. of IEEE Interconnect Technology Conference, pp. 6-8, 2000.

[16] I. Söderquist, "Globally-Updated Mesochronous Design Style (GUM-design-style)", In Proc. of Esscirc-2002, pp. 603-606, 2002.

[17] F. Mu, C. Svensson, "Self-Tested Self-Synchronization Circuit for Mesochronous Clocking", In IEEE Trans. on Circuits and Systems - II: Analog and Digital Signal Processing, pp. 129-140, Vol. 48, No. 2, Feb. 2001.

[18] M. Millberg, "The Nostrum Protocol Stack and Suggested Services Provided by the Nostrum Backbone", Technical Report, Royal Institute of Technology, TRITA-IMIT-LECS-R02:01, 2002.

Chapter 9

A PARALLEL COMPUTER AS A NOC REGION

Martti Forsell
VTT Electronics

Abstract: A network on chip (NOC) scheme relying on reuse of existing intellectual property blocks and a unified communication solution has been proposed for solving architectural and design productivity problems of future systems on chips. A NOC consists of a set of heterogeneous computing and storage resources that are connected to each other via a standardized communication network. A heterogeneous structure is suitable for application specific computing, but not for high-speed general purpose computing that is increasingly used in devices that will be powered by NOCs. General purpose functionality can, however, be provided by dedicating the whole chip or a special area on a heterogeneous NOC, called region, for a homogeneous general purpose computing engine. Unfortunately the architectures proposed for NOCs and on-chip parallel computers feature poor performance and portability, or are difficult to program in general purpose computing due to limited communication bandwidth, inability to eliminate delays caused by the latency of the network, high communication overheads, and poor models of parallel computing. In this chapter we will discuss the problems and solutions of implementing an efficient single chip general purpose parallel computing engine. We will also describe our ECLIPSE architecture that can be used either as a truly scalable, high-speed, single chip parallel computer or as a NOC region responsible of general-purpose computation.

Key words: General purpose computing, Parallel computing, NOC, region, architecture

1. INTRODUCTION

There is a strong trend towards general-purpose operation in design of future hand held devices like multimedia phones, communicators and PDAs. Users of such devices are not happy with just calendars, phone catalogs and

A. Jantsch and H. Tenhunen (eds.), Networks on Chip, 173–192.

© 2003 *Kluwer Academic Publishers. Printed in the Netherlands.*

Qualcomm Exhibit 1006

wap browsers, but they also demand at least limited access to connectivity tools like web browsers with full multimedia capabilities, various remote office utilities for text, graphics, spreadsheet and database processing, amusement software including games, virtual reality and musical instruments, and application development. This sets hard requirements for the architectures of chips powering them, because current architectures tuned for signal processing are not efficient in general purpose computing.

A silicon chip can be considered as a parallel computing device, consisting of a high number of interconnected computing elements and memory cells. These elements should be organized so that the computational tasks that are aimed for the chip will be efficiently executed. The main alternatives in organizing the elements are dedicated logic, reconfigurable logic and processors. Dedicated logic is the most efficient in solving a single computational problem, but each problem requires a different dedicated logic. Reconfigurable logic lies in between dedicated logic and processors featuring connectivity limitations as well as performance and area overheads in respect to dedicated logic while giving better performance than processors in certain applications requiring e.g. wide buses. Reconfigurable logic is, however, not as nearly as general purpose as processors, because altering a configuration is slow in respect to processor instruction fetches. Processors provide the largest variability and in theory they are able to solve virtually any kind of a computational problem. Although this happens at the cost of performance, organizing computational elements as one or more processors is the only real alternative for general purpose processing.

There exist a wide spectrum of architectural alternatives how processors on a high-capacity chip should be designed and how intercommunication should be organized so that general purpose operation, easy programmability, high performance, and portability of applications could be achieved. The main alternatives are a single processor with a low number of wide functional units, a single processor with a high number of ordinary functional units, and multiple processors that are connected via a communication network. Single processor configurations are, however, not efficient in general purpose computing, because they are not able to exploit control parallelism. Furthermore, a large amount of bit-level and instruction-level parallelism is difficult extract from typical application programs, and implementations of such architectures will inevitably have some long wires decreasing the maximum clock frequency remarkably [1, 2]. Multiprocessor configurations can—at least in theory—be used for general purpose computing, but then one must deal with one more level of parallelism, processor intercommunication, synchronization, and data and program partitioning issues [3]. Unfortunately, most architectures proposed for parallel computing so far are limited to narrow application areas only,

feature poor performance and portability, or are difficult to program [4, 5]. This applies especially to the very few existing single chip parallel solutions, e.g. PACT XPU128.

A *network on chip* (NOC) scheme relying on reuse of existing intellectual property blocks and a unified communication solution has been proposed for solving architectural and design productivity problems of future *systems on chips* (SOC) [6, 7, 8]. A NOC consists of a set of heterogeneous computing and storage resources that are connected to each other via a standardized communication network. A heterogeneous structure is suitable for application specific computing, but not for high-speed general purpose computing that is increasingly used in devices that will be powered by NOCs. General purpose functionality can, however, be provided by dedicating the whole chip or a special area on a heterogeneous NOC, called region, for a homogeneous general purpose computing engine. Unfortunately the architectures proposed for NOCs feature similar problems in general purpose computing as parallel architectures discussed earlier.

In this chapter we will discuss the problems and solutions of implementing an efficient single chip general purpose parallel computing engine. We will also describe our ECLIPSE architecture that can be used either as a truly scalable, high-speed, single chip parallel computer or as a NOC region responsible of general-purpose computation.

The rest of this chapter is organized as follows: In section 2 we will give a short introduction to parallel computing. In section 3 we will discuss about the problems and solutions of implementing an efficient single chip general purpose parallel computing engine. Our proposal for such an engine is introduced in section 4. Finally, in section 5 we will give our conclusions.

2. PARALLEL COMPUTING

Parallel computing is a natural way to solve computational problems. In fact, it is even more natural way than sequential computing, because most computational problems are parallel by their nature and human effort is needed to sequentialize them for current sequential programming languages and sequential machines. Furthermore, by doing that a programmer inserts artificial dependencies that are difficult to eliminate afterwards [9].

Parallel computing has been a target of scientific research for last 40 years [10, 11, 12, 3, 4, 5, 13, 14]. Unfortunately these efforts have not yet yielded into commercially successful parallel computers that would be real alternatives to our sequential computers.

Qualcomm Exhibit 1006

2.1 Classes of parallelism

Different types of parallel computing can be classified into four main categories according to granularity of parallelism expressed in programs—bit-level parallelism, instruction-level parallelism, thread-level parallelism, and program-level parallelism.

The most obvious and easiest way to benefit from parallelism is to apply a logical operation to multiple bits simultaneously. This is called *bit-level parallelism* (BLP). It is used in most logic blocks, e.g. in arithmetic and logic units (ALU), in which operations are applied to, e.g. 32 bits in parallel.

It is possible to speed up processors by executing two or more independent instructions simultaneously in two or more functional units. This is called *instruction-level parallelism* (ILP). The two main forms of ILP are pipelined [10] and superscalar [11] execution, which both are used in current processors. Communication between functional units happens via an internal network.

The power of multiple interconnected processors can be exploited in solving a single computational problem by writing own subprogram or thread, which solves a part of the problem, for each processor and by letting the subprograms communicate with each other so that the entire problem is solved. If these subprograms can be executed in parallel we are exploiting *thread-level parallelism* (TLP). In TLP systems there are two principal schemes expressing how processor intercommunication is implemented, by passing messages and by using shared memory [12, 3]. If the subprograms are executed in separate computers connected together via a network, we are speaking about *program-level parallelism* (PLP) or even *distributed computing* if the distance between computers is high.

Another classification of parallel computing is division to categories according to type of operations executed in parallel. In *data parallelism*, an application contains multiple data operations that are executed in parallel. The two main types of data parallelism are homogeneous data parallelism used in e.g. vector computers and heterogeneous data parallelism used in e.g. VLIW processors. In *control parallelism* there are multiple control operations that are executed in parallel. Note that it is not possible to exploit control parallelism in single processor systems because by definition a processor can have only one functional unit executing control instructions.

2.2 Models

A *model of parallel computing* is a formal, abstract definition of a parallel computer. It describes the basic components, their properties and available operations, and operation granularity and synchronization.

Researchers can use models to analyze an algorithm's intrinsic execution time or memory space while ignoring many implementation issues.

There are two primary parallel computing models—the *parallel random access machine* (PRAM) model and the *message-passing* (MP) model. The main difference between these models is in how interaction between processors is organized. A PRAM is a fine-grained lock-step-synchronous model of parallel computation [3]. It consists of an unbounded set of processors connected to the same clock and shared memory. All operations, including parallel memory accesses, execute in unit time. Each processor has an ID register with a unique value. A number of memory access variants exist—for example, exclusive-write, exclusive-read (EREW) and concurrent read, concurrent write (CRCW). CRCW lets multiple processors access a memory location simultaneously while EREW does not. The MP model is widely used in parallel computing. A machine using the MP model consists of a set of processors attached to an interconnection network. Processors communicate by sending messages to each other via a network. Each processor has its own local memory, and implements synchronization through message passing.

2.3 Programming

High-level programs written for the PRAM model express processor intercommunication, synchronization, and data partitioning differently from programs written for the MP model. In PRAM, a programmer can simply place data requiring cooperation into the shared memory. In MP, data partitioning and movement must be handled by explicitly inserting partitioning directives as well as send and receive calls to ensure that the right data is in the right place at the right time. Similarly, a PRAM programmer can rely on the model's synchronicity, while an MP programmer must insert explicit synchronization primitives where synchronicity is required. In addition, PRAM algorithmic theory is well known and there exists a large base of ready-to-use PRAM algorithms [15, 14] while MP algorithms are more difficult to write and their efficiency depends heavily on the underlying MP architecture.

Automatic techniques for parallelization of sequential programs have been developed to overcome the problem writing programs in parallel, but the results have been modest even if the computational problems have been clearly parallel. This is caused by the artificial dependencies generated by solving a parallel problem with a sequential algorithm and the overall complexity of the algorithm theory [9].

Qualcomm Exhibit 1006

2.4 Current single chip solutions

The area of single chip parallel computers and NOCs is currently very actively investigated. Among the most important proposals are:

- *Scalable, Programmable, Integrated Network* (SPIN) is a communication architecture for possibly heterogeneous single chip parallel computers [16]. The topology of the network is a fat tree, in which the degree of a node is eight.
- *Networks on Chips* (NOC1) is a work describing on-chip micronetworks designed with a layered methodology aiming for functionally correct, reliable operation of interacting SOC components [6].
- *Network on silicon* architecture (NOS) is a hardware architecture to implement communication between IP cores in future technologies [8]. The work gives also a software model in the form of a protocol stack to structure the programming of NOSs.
- *Network on chip* (NOC2) is an architecture and design methodology for future SOCs [7]. The architecture is an m x n mesh of switches and heterogeneous resources that are placed on the slots formed by switches providing physical-architectural-level integration. The work also describes a protocol stack for NOC resource intercommunication.
- *Smart memories* (SME) is a modular reconfigurable architecture targeted at computing needs in 0.1 um technology generation [17]. A SME chip is made up of a homogeneous mesh of processing tiles, each containing local memory, local interconnect, and a processor core with caches. Tile intercommunication is handled with nine global busses.
- The *RAW microprocessor* is a computational fabric for software circuits and general purpose programs [18]. A RAW processor design divides the usable silicon area into 16 tiles consisting of routers, pipelined RISC processor, FPU, and caches. Tile interconnection is handled with four mesh networks operating in parallel.

These proposals rely on the MP model in resource intercommunication making them easier to implement with the cost of programmability, portability, and performance in general purpose applications. NOC2 introduces, however, a region concept that allows dedicating an area on a NOC for a logic that has different internal topology and communication mechanism. If suitable region architecture is found, NOC2 may provide a framework allowing embedding general purpose operation into a NOC.

3. PROBLEMS AND SOLUTIONS

In this section we discuss the problems of realizing general purpose parallel computing and solutions that are proposed to solve these problems. These problems are related to the bandwidth, latency, speed difference, synchronization, and physical feasibility as a single silicon chip. We assume here that the machine at hands uses a physically distributed memory because implementing a large uniform shared memory directly as real multiport memory seems impossible due to complexity issues [19].

3.1 Bandwidth

Executing a parallel program in a parallel machine requires always some resource intercommunication. The amount of this communication is heavily dependent on the algorithm used in the program ranging from virtually non-communicating coarse-grained processes to fine-grained threads that communicate with each other every clock cycle. In high performance general purpose systems the execution machinery should be designed so that even the most communication intensive fine-grained algorithms can be executed efficiently meaning that the bandwidth of the intercommunication network must be proportional to the number of processors. Unfortunately, this much bandwidth may be required by a small set of processors or even by a single processor, because communication patterns, that are produced by typical parallel programs, are likely to cause congestion of messages in the network if typical data partitioning schemes, like stacking and interleaving, are used (see Figure 9-1).

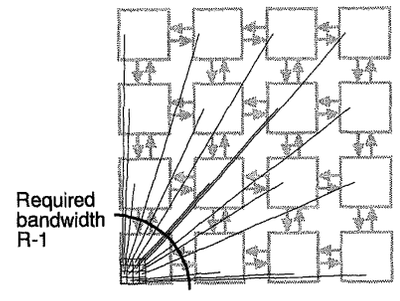


Figure 9-1. Congestion of messages in a machine using simple interleaving occur, if e.g. the size of records equals the number of resources R and the records are accessed in parallel.

The maximum bandwidth requirement for a machine using the shared memory scheme is even higher because memory modules may send read replies back to processors while processors are sending read messages to the

memory modules. In a NOC also the resources not participating in general purpose computing generate traffic that may interfere with the traffic generated by the general purpose machinery. This is far too much for any topology except the fully connected topology. The lack of bandwidth and congestion of messages cause arbitrary long delays, which ruin the performance and real-time operability of the system. One may be tempted to think that if communication could be made local even high amount of communication could be tolerated. Unfortunately, there exists no method to guarantee the locality of communication in general purpose computing.

Congestion can effectively be avoided by randomizing data partitioning, but this technique works for the shared memory scheme only. In the message passing scheme also algorithm functionality should be randomly partitioned, which is not possible. In shared memory machines data partitioning can be implemented by randomly selecting a polynomial hashing function from a family of hashing functions, which map memory locations over the modules [20, 21]. This kind of randomization decreases the bandwidth that is needed for efficient communication dramatically. With high probability, the number of processors divided by two is enough for bisection and one is enough for a single resource [5]. Unfortunately, even this is too much for topologies used in most parallel computers including a 2-d mesh used in NOC2, fixed number of parallel buses used in SME, and fixed number of global buses used in SM (see Figure 9-2). There exist, however, various scalable topologies that meet these bandwidth requirements, e.g. sparse mesh, coated mesh, hypercube, and butterfly [5, 22, 14, 23].

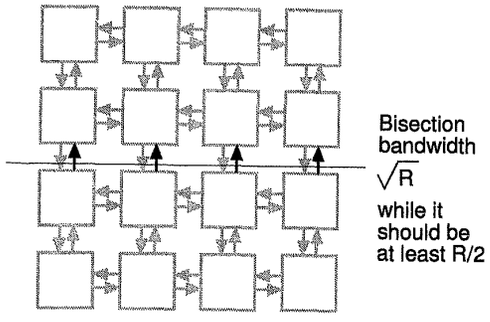


Figure 9-2. The bisection bandwidth of a 2D mesh used in NOC2 is not enough for randomized communication (R=number of processor resources).

In a NOC, it is possible to prevent the traffic generated by the heterogeneous part of the chip from interfering the communication of the

general purpose machinery by isolating the general purpose machinery from the rest of the machine e.g. by using the region concept introduced in NOC2 [7]. A *region* is an area inside the NOC, which is insulated from the network and which may have different internal topology and communication mechanisms (see Figure 9-3). Regions are connected to the rest of the NOC by special communication arrangements called wrappers, which route packets so that regions are insulated from external traffic. Wrappers are also responsible for converting the messages into appropriate format.

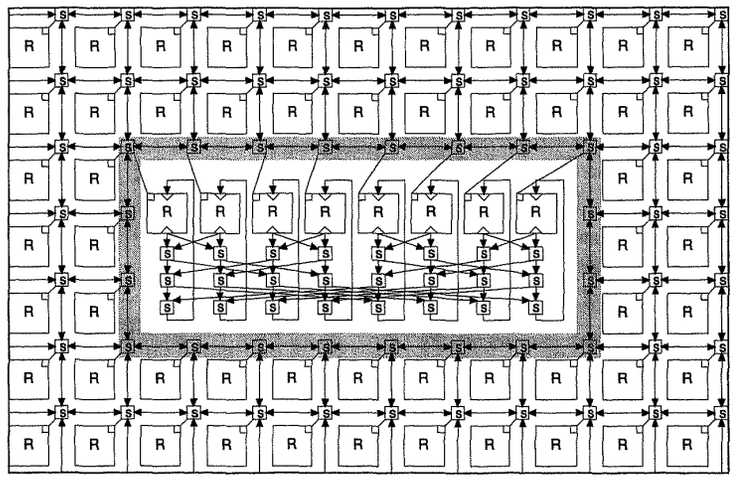


Figure 9-3. A region inside a NOC rounded by the wrapper (S=switch, R=resource).

3.2 Latency and speed difference

Currently and in the foreseeable future, processors operate tens or even hundreds of times faster than high capacity memory blocks [24]. This is due to physical constraints of DRAM technologies. Sequential computer designers have solved this *speed difference* problem by placing one or more levels of cache memories between processors and main memories. In parallel computers using the synchronous shared memory model, e.g. PRAM, we can not use this technique, because full cache coherency among multiple processors would be very expensive to maintain. In the worst case, all P processors could alter the contents of the main memory simultaneously. To keep caches coherent, one would need to implement P-ported caches. According to investigations [19] implementing the basic block of P-ported cache, true multiport memory, is P^2 times more expensive than

implementing the single port cache of the same size. Another problem in parallel computing is the high *latency* of intercommunication, because in general purpose parallel computing we cannot assume that data required for computation is available locally. This is due to functionality of the parallel program is distributed to processors producing data that is often needed by other processors. In the worst case the needed data is in the opposite edge of the machine, meaning that the latency of a read would be two times the latency of the intercommunication network plus the latency of the memory module. This causes long delays in execution and decreases the utilization of the functional units dramatically.

A possible solution to the speed difference problem is to use banking so that the fast stream of references generated by a processor is divided into multiple slow streams that can be handled efficiently by slow memories (see Figure 9-4). A congestion of memory references can occur in a bank or a small set of banks just like in network-level communication. This problem can be solved by randomly hashing of memory locations over the banks of a module like in the case of network-level communication. According to investigations this kind of a memory system organization efficiently eliminates the delays cause by the speed difference between processors and memories [25, 26].

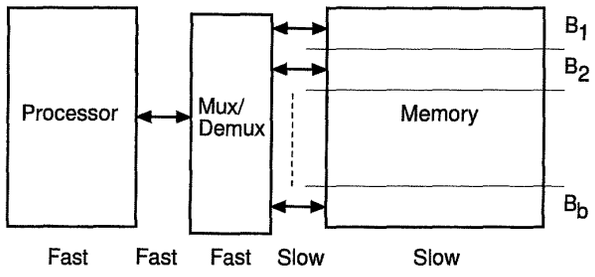


Figure 9-4. A fast stream of references in is divided almost evenly into multiple slow streams that can be handled efficiently by slow memory banks.

In sequential computing there is no solution to the communication latency problem, but in parallel computing one can use so called *parallel slackness* to avoid delays in execution. The idea is that if multiple threads are assigned to a single physical processor it is possible to execute other threads while a thread is accessing memory or committing communication (see Figure 9-5). Special multithreaded processors and a pipelined memory system are needed to implement this efficiently [27, 14].

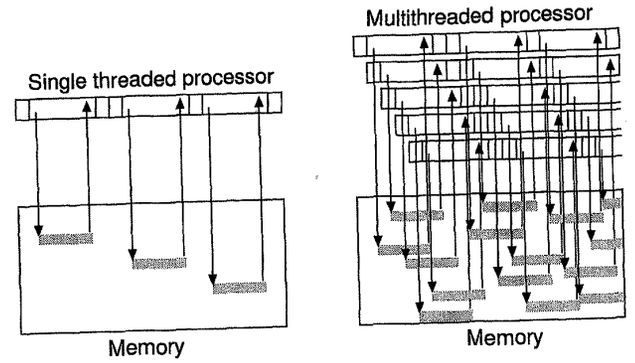


Figure 9-5. Normal execution (left) and execution exploiting parallel slackness (right).

3.3 Synchronization

Advanced programming models assume at least some degree of synchronicity in execution and communication. In PRAM synchronization is completely transparent to the programmer [3]. Consider a hardware committing synchronization after each machine step. In the naive technique synchronization is committed after all threads have completed their memory references. It is not efficient, however, because the variation of latencies of references initiated by different threads may be large leading to decreased utilization of functional units in processors (See Figure 9-6).

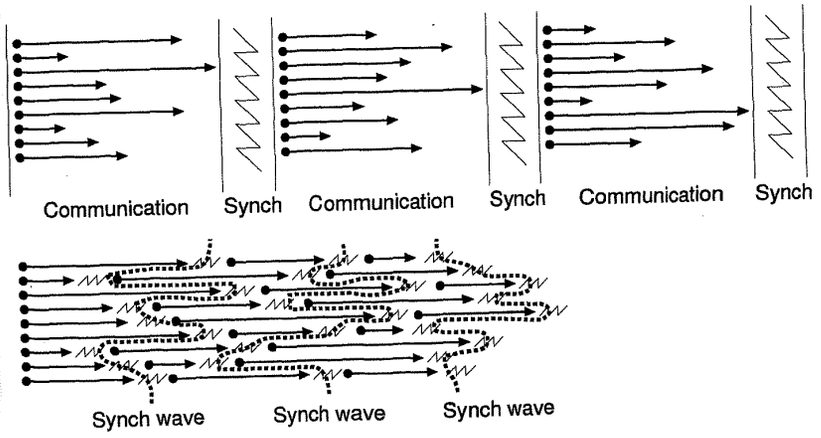


Figure 9-6. Three consecutive synchronizations using naive technique (top) and synchronization wave technique (bottom).

There exist more efficient synchronization techniques, e.g. synchronization wave, that virtually eliminate the synchronization cost caused by the variance of communication latencies. It is a technique for acyclic networks in which special synchronization packets are sent by the processors to the memory modules and vice versa [28]. The idea is that when a processor has sent all its packets on their way, it sends a synchronization packet. Synchronization packets from various sources push on the actual packets, and spread to all possible paths, where the actual packets could go. When a node receives a synchronization packet from one of its inputs, it waits, until it has received a synchronization packet from all of its inputs, then it forwards the synchronization wave to all of its outputs. The synchronization wave may not bypass any actual packets and vice versa. When a synchronization wave sweeps over a network, all nodes and processors receive exactly one synchronization packet via each input link and send exactly one via each output link (see Figure 9-6).

3.4 Physical feasibility

Logarithmic diameter networks, e.g. hypercube, butterfly and fat tree, provide theoretical scalability, low diameter and enough bandwidth to solve the routing problem for randomized communication required to implement efficient general purpose parallel computing. Unfortunately, it is not possible to map these topologies into two dimensions provided by a silicon chip, without increasing the length some interconnection wires proportionally to the number of processors (see Figure 9-7). This will decrease the clock frequency dramatically and sabotage the performance.

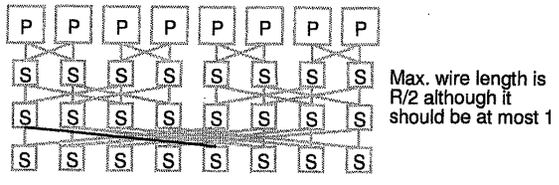


Figure 9-7. The maximum wire length in a butterfly network (P=processor, S=switch, R=number of processing resources).

The only method to overcome the maximum wire length problem is to limit the length of wires so that the wire delay plus logic delay does not exceed the clock cycle time of fast systems. There exist network topologies that are scalable, meet the bandwidth requirements set by efficient general purpose algorithms, and feature fixed wire length, e.g. two-dimensional sparse meshes and coated meshes [5, 22, 23]. They are like ordinary meshes,

but processors and memories are placed sparsely in the mesh or lie in the edges of the mesh (see Figure 9-8)

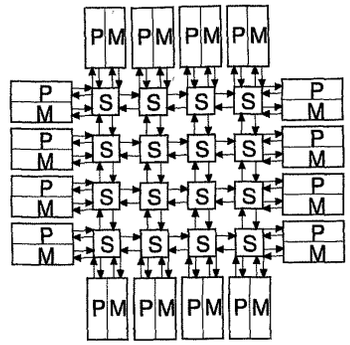


Figure 9-8. A two-dimensional coated mesh (P=processor, M=memory module, S=switch).

4. ECLIPSE ARCHITECTURE

Embedded Chip-Level Integrated Parallel SupErcomputer (ECLIPSE) is a scalable high-performance computing architecture for NOCs [23]. An ECLIPSE is composed of MTAC multithreaded processors with dedicated instruction memory modules, highly interleaved data memory modules, and a high-capacity sparse mesh interconnection network (see Figure 9-9). ECLIPSE features a homogeneous and totally cacheless architecture for simplifying design and avoiding cache coherency problems.

4.1 Programming model

ECLIPSE provides an easy-to-program EREW PRAM-style programming model with a large number of physical threads. The model provides a uniform shared data memory with single superstep memory access time and machine instruction level synchronous execution. Parallel access to shared memory is limited, allowing at most one memory reference per memory location per superstep. The programming model allows for using an ECLIPSE as a single SIMD machine, as a single MIMD machine, or as multiple SIMD and MIMD machines. Execution in ECLIPSE happens in supersteps, which are transparent to the user. During a superstep each thread of each processor executes an instruction, which may include at most one shared memory reference subinstruction.

Qualcomm Exhibit 1006

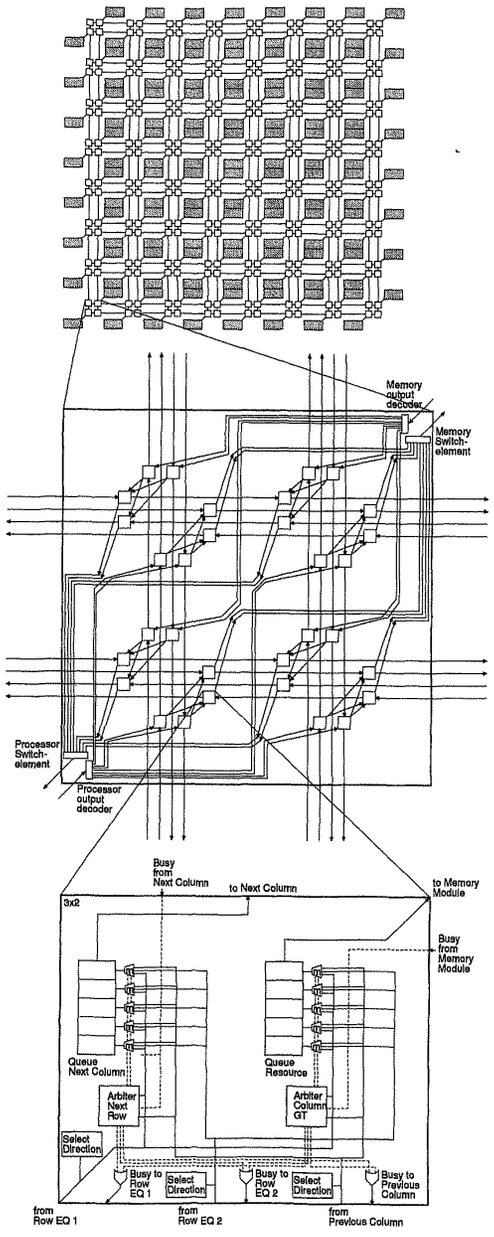


Figure 9-9. A block diagram of an ECLIPSE communication network (top), superswitch (middle), and switch element (bottom).

4.2 MTAC processor

Multithreaded architecture with chaining (MTAC) [27] is a VLIW processor architecture especially designed for realizing the PRAM model on a physically distributed memory architecture. An MTAC processor consists of *a* ALUs, *m* memory units, a hash address calculation unit, a compare unit, a sequencer, and a distributed register file of *r* registers (see Figure 9-10). MTAC has a VLIW-style instruction set with the fixed execution ordering of subinstructions reflecting the chain-like organization of functional units, tools for using subinstruction results as operands of the following subinstructions in the chain, and hardware assisted barrier synchronization mechanism. MTAC supports overlapped execution of a variable number of threads. Multithreading is implemented as a deep, cyclic, hazard-free interthread superpipeline for hiding the latency of the memory system, maximizing the overlapping of the execution, and minimizing the register access delay. Switching between threads happens in zero time, because threads proceed in the pipeline only during the forward time.

The organization of functional units in MTAC is targeted for exploiting ILP during supersteps of parallel execution. Therefore functional units in MTAC are connected as a chain, so that a unit is able to use the results of its predecessors in the chain. The ordering of functional units in the chain is selected according to the average ordering of instructions in a basic block: Two thirds of the ALUs form the beginning of the chain. They are followed by the memory units and the rest of the ALUs. The compare unit and the sequencer are located in the end of the chain, because comparing and branching happen always in the end of basic blocks.

4.3 Memory modules

ECLIPSE has two types of memory modules, data memory modules and instruction memory modules, which are isolated from each other to guarantee uninterrupted data and instruction streams to MTAC processors. Both types of modules use deep interleaving to eliminate performance loss due to constantly increasing speed difference between processors and DRAM-based memory banks.

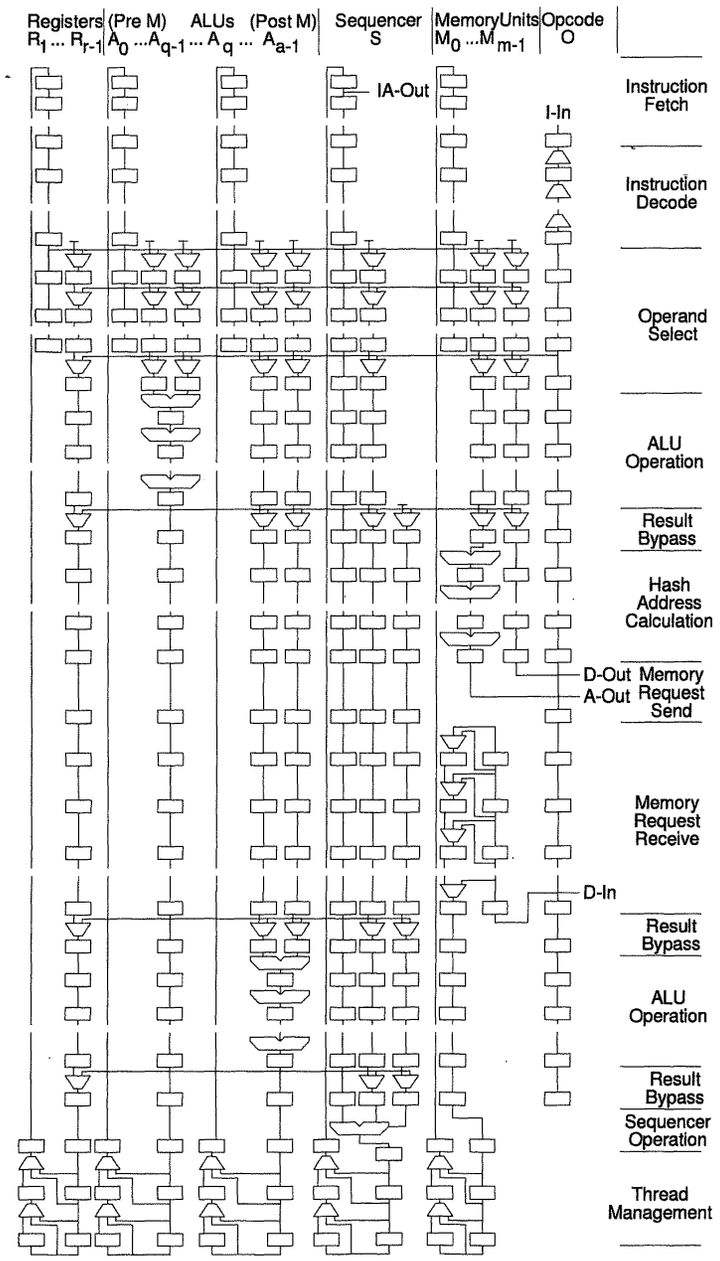


Figure 9-10. A block diagram of an MTAC processor.

A data memory module is composed of B memory banks attached to Q-slot access queues for incoming memory references and a common queue for outgoing memory replies [25]. For optimum throughput, B should be at least the memory bank cycle time divided by the processor cycle time. Unlike ordinary interleaved memories, ECLIPSE modules use a randomly chosen linear hashing function to map memory locations over the banks of a module to minimize congestion of messages within the module.

There exist two alternative designs for instruction memory modules [26]. The interleaved alternative is similar to data memory modules except it features combining of references targeted into the same location. It requires also that a MTAC connected to it is armed with a pipelined fetch unit, which has a similar structure as a data memory unit of MTAC [26]. The trivial alternative has a bank for each thread of MTAC and works without a pipelined fetch unit. It is very efficient in MIMD-style processing, but wastes a lot of memory in SIMD-style processing, because multiple copies of the program are needed. The interleaved alternative avoids this problem, because it keeps only a single copy of each instruction.

4.4 Interconnection network

The ECLIPSE network is a high-bandwidth acyclic variant of a two-dimensional sparse mesh (see Figure 9-9) featuring separate lines for messages going from processors to memories and for messages returning from memories to processors, two-level organization of switches, simple routing, efficient synchronization mechanism, and randomized hashing of memory locations over the memory modules.

We use a variant of two-dimensional sparse mesh in which the number of switches is at least the square of the number of processing resources divided by four, with separate lines for going and returning messages, as ECLIPSE communication architecture, because it provides true scalability, enough bandwidth for heavy random communication, and the degree of nodes as well as the length of interconnection lines are fixed independently on the number of processing resources. Moreover, routing in a mesh is easy yielding to potentially small switches. To exploit locality, the switches related to each resource pair are grouped as superswitches (see Figure 9-9). This kind of a two-level structure allows for sending a message from a resource to any of the switches belonging to a superswitch in a single clock cycle as well as pipelining the operation of switches in a natural way.

ECLIPSE uses slackness of parallel execution for hiding the network and memory module access latency. All this works without cache memories and coherency problems. Memory locations are distributed around the modules

Qualcomm Exhibit 1006

by a randomly chosen linear hashing function for avoiding congestion of messages and hot spots.

In ECLIPSE, the shared memory of the EREW PRAM model is emulated by sending memory requests (reads and writes) and synchronization messages from the processors to the memory modules and vice versa. Messages are routed by using the simple greedy algorithm with two intermediate targets: A message is first sent to a first intermediate target, which is a randomly chosen switch in a superswitch related to the sending resource. Then the message is routed greedily (go to the right row and then go to the right column) to the second intermediate target, which is a randomly chosen switch in the superswitch related to the target resource. Finally the message is routed from the second intermediate target to the target resource. Deadlocks are not possible during communication because the network is acyclic.

ECLIPSE uses the synchronization wave technique to keep messages belonging to successive supersteps separated. Synchronization wave is also used for implementing multiple simultaneous barrier synchronizations: A MTAC processor determines the number of threads participating into each barrier synchronization during each superstep. These numbers are assigned to appropriate fields of the synchronization wave message that is sent out at the end of the superstep. As the synchronization messages in the network are proceeding the numbers in these fields are selectively summed together and the sums are sent forward. By observing the numbers of the barrier fields in synchronization waves returning to processors, the processor can determine the moment when all the threads participating in a barrier have arrived to synchronization point and allow them to continue execution.

4.5 Performance

We compared analytically ECLIPSE with a homogeneous baseline NOC consisting of an ordinary 2-dimensional mesh network of popular embedded ARM9 processors with data and instruction caches, and local embedded memories assuming that the architectures occupy roughly the same silicon area [23]. In execution of a parametric benchmark program ECLIPSE provided up to two decades higher performance than the baseline NOC. ECLIPSE seemed also relatively independent on memory speed and switch delay. Increasing the level of superpipelining or the number of processors increase the performance of ECLIPSE respectively while only the latter was true for the baseline NOC.

We compared ECLIPSE also with an ideal PRAM machine having the same instruction set and similar configuration with simulations [23]. With a

suite of simple parallel benchmarks ECLIPSE was only about 20% slower than the PRAM machine.

5. CONCLUSIONS

We have discussed about the problems and solutions for implementing general purpose computing engine on a heterogeneous NOC. Due to various architectural and efficiency reasons such an engine has to be implemented as an isolated region having differently organized internal communication and processing resources optimized for parallel computing. We have also described our ECLIPSE architecture that is a scalable high-performance computing solution that can be used as a standalone single chip computer or as a region of larger NOC responsible of general purpose computing. ECLIPSE relies on a sophisticated model of parallel computing and high capacity communication network to making programming easy and the cost of memory accesses low.

Our future work includes more thorough performance evaluations and area estimations, finding out various practical implementation issues, building a prototype and a compiler for ECLIPSE.

REFERENCES

[1] N. Jouppi and D. Wall, Available Instruction-Level Parallelism for Superscalar and Superpipelined Machines, Proceedings of the Third International Conference on Architectural Support for Programming Languages and Operating Systems, April 1989, 272-282.

[2] M. Forsell, Architectural differences of efficient sequential and parallel computers, Journal of Systems Architecture 47, 13 (July 2002), 1017-1041.

[3] S. Fortune and J. Wyllie, Parallelism in Random Access Machines, Proceedings of 10th ACM STOC, Association for Computing Machinery, New York, 1978, 114-118.

[4] G. Almasi and A. Gottlieb, Highly Parallel Computing, Benjamin/Cummings, Redwood City, 1994.

[5] V. Leppänen, Studies on the realization of PRAM, Dissertation 3, Turku Centre for Computer Science, University of Turku, Turku, 1996.

[6] L. Benini and G. De Micheli, Networks on Chips: A New SoC Paradigm, IEEE Computer 35, 1 (Jan. 2002), 70-78.

[7] S. Kumar, A. Jantsch, J. Soininen, M. Forsell, M. Millberg, J. Öberg, K. Tiensyrjä and A. Hemani, A Network on Chip Architecture and Design Methodology, In the Proceedings of the ISVLSI'02, April 25-26, 2002, Pittsburgh, Pennsylvania, 117-124.

[8] K. Goossens, E. Rijpkema, P. Wielage, A. Peeters and J. van Meerbergen, Networks on Silicon: The Next Design Paradigm for Systems on Silicon, Design, Automation and Test in Europe Conference, March 4-8, 2002, Paris, France, 423-425.

[9] M. Forsell, V. Leppänen and M. Penttonen, Primitives of Sequential and Parallel Computation, Report 1998/A/3, Department of Computer Science and Applied Mathematics, University of Kuopio, Kuopio, 1998.
[10] E. Bloch, The engineering design of the Stretch computer, Proceedings of the Fall Joint Computer Conference, 1959, 48-59.
[11] J. Thornton, Parallel operation in the Control Data 6600, Proceedings of the Fall Joint Computer Conference 26, 1964, 33-40.
[12] J. T. Schwarz, Large Parallel Computers, Journal of the ACM 13, 1 (1966), 25-32.
[13] D. Culler and J. Singh, Parallel Computer Architecture—A Hardware/ Software Approach, Morgan Kaufmann Publishers Inc., San Fransisco, 1999.
[14] J. Keller, C. Keßler, and J. Träff, Practical PRAM Programming, Wiley, New York, 2001.
[15] J. Jaja, Introduction to Parallel Algorithms, Addison-Wesley, Reading, 1992.
[16] P. Guerrier, A. Greinier, A Generic Architecture for On-Chip Packet-Switched Interconnections, Proceedings of DATE 2000, March 27 -30, 2000, Paris, France, 250 - 256.
[17] K. Mai, T. Paaske, N. Jayasena, R. Ho, W. Dally and M. Horowitz, Smart Memories: A Modular Reconfigurable Architecture, In the Proceedings of the 27th International Symposium on Computer Architecture, Vancouver, Canada.
[18] M. Taylor, et. al., The Raw Microprocessor: A Computational Fabric for Software Circuits and General Purpose Programs, IEEE Micro 22, 2 (March-April 2002), 25-35.
[19] M. Forsell, Are Multiport Memories Physically Feasible?, Computer Architecture News 22, 4 (September 1994), 47-54.
[20] A. Karlin and E. Upfal, Parallel Hashing—an Efficient Implementation of Shared Memory, Journal of the ACM 35, 4 (1988), 876-892.
[21] M. Dietzfelbinger, A. Karlin, K. Mehlhorn, F. Mayer auf der Heide, H. Rohnert and R. Tarjan, Dynamic Perfect Hashing: Upper and Lower Bounds, SIAM Journal on Computing 23, (August 1994), 738-761.
[22] J. Sibeyn, Solving Fundamental Problems on Sparse-Meshes, IEEE Transactions on Parallel and Distributed Systems 11, 12 (December 2000), 1324-1332.
[23] M. Forsell, A Scalable High-Performance Computing Solution for Network-on-Chips, IEEE Micro 22, 5 (September-October 2002), 46-55.
[24] A. Allan, et al., 2001 Technology Roadmap for Semiconductors, Computer 35, 1 (2002), 42-53.
[25] M. Forsell and V. Leppänen, Memory Module Structures for Shared Memory Simulation, In the Proceedings of the International Conference on Advances in Infrastructure for e-Business, e-Education, e-Science, and e-Medicine on the Internet, January 21-27, 2002, L'Aquila, Italy.
[26] M. Forsell, Cacheless Instruction Fetch Mechanism for Multithreaded Processors, WSEAS Transactions on Communications 1, 1 (2002), 150-155.
[27] M. Forsell, MTAC—A Multithreaded VLIW Architecture for PRAM Simulation, Journal of Universal Computer Science 3, 9 (1997), 1037-1055.
[28] A. Ranade, How to Emulate Shared Memory, Journal of Computer and System Sciences 42, (1991), 307-326.

Chapter 10

AN IP-BASED ON-CHIP PACKET-SWITCHED NETWORK

Ilkka Saastamoinen, David Sigüenza-Tortosa and Jari Nurmi

1. Introduction

This chapter gives an overview of our current work in the field of Network-on-Chip design at the Tampere University of Technology. We are developing a NoC architecture that we call *Proteo*, the name of an ancient Greek god who could change his form at will. He also had the gift of prophecy, although he was not very inclined to give oracles to mortals... With this name we want to express the idea of flexibility: based on a small library of predefined, parameterized components, we are able to implement a range of different topologies, protocols and configurations.

Our approach gets its motivation from IP (Intellectual Property) based design flow and the basic principles behind it. The main benefits of IP-based design are structural system implementation - which is constructed using pre-designed hardware and software IP modules - and abstraction of design. These features enable faster design cycle since a design space can be divided to separated modules which are implemented independently. For example, modularity of design enables distributing the tasks in the design project to those designers who are experts in certain fields. As a result, time-to-market of the product is reduced and costs are decreased. Another advantage is that we can use already available IP blocks implementing the design modules. If interfaces of the IP block are designed properly, it can be connected to the target system with little effort and cost. More benefits are gained when modularity is used to hide implementation complexity. Detailed and complex decisions about low level implementation can be delayed and in the beginning of the design process the resources are spent for finding the best algorithms and architectures for the application at a higher abstraction level. At

A. Jantsch and H. Tenhunen (eds.), Networks on Chip, 193–213.
© 2003 *Kluwer Academic Publishers. Printed in the Netherlands.*

Qualcomm Exhibit 1006

this level the lower level blocks are seen as a library of building blocks where each block has a predefined interface and functionality.

The Virtual Socket Interface Alliance (VSIA) is an organization whose goal is to promote IP block reuse and integration solutions. One of their first efforts has been the definition of a standard interface for IP blocks. The goal of this interface recommendation is to ease the interconnection of IP blocks from different suppliers, using a basic set of signals, protocols and communication semantics[1][2]. Although VSIA's interface is not very common in commercial products, it is a good, neutral selection in design cases where choosing a proprietary solution would be a design constraint. Another similar interface recommendation is Open Core Protocol Specification from the OCP-IP Association [3].

Dividing the design flow into several layers and mapping the upper level design description to the more detailed lower level description are typical strategies of a platform-based design methodology [4][5]. Platform-based design follows meet-in-the-middle flow where the goal is to map the application to a pre-designed hardware architecture - a platform instance - through a more abstract architecture platform. In this way the platform-based methodology extends the reuse from single blocks to large architectures. The reuse of platform instances for several applications leads to reduced design costs. However, a general-purpose implementation platform requires much re-configurability or programmability. Software programmability gives usually more freedom and possibilities to alter the functionality of the platform. While with hardware re-configuration it is possible to achieve a significantly better implementation in terms of area, performance and power. Optimal platform architecture may contain configurable blocks both in the software and hardware domains.

Traditionally only the functional parts of a system are included in the IP based design methodology. However, platform-based design and the use of more abstract descriptions is changing the situation. When different applications are mapped to the same platform, the on-chip communication requirements change accordingly. Therefore also the communication mechanisms must be designed in a structured and flexible way. Methodologies for the separated design of system functionality and communication have been proposed[6][7]. It is feasible now to design a general-purpose communication architecture independently and incorporate it into unrelated products. The problem is to design an interconnection mechanism for systems built from heterogeneous blocks, providing an adequate level of performance for a given application. The solution will consist of a network of some type, a set of protocols and a standard interface for accessing the network, along with the implementation of a set of software tools for the automation of the network integration process.

Various architectures for inter-block communication have already been developed and commercially used in SoC designs [8][9]. All of them implement a memory-mapped architecture and are equivalent to buses. An overview of these bus-based systems can be found in [10]. Several researchers have proposed packet switching networks with more general topologies as a solution for the interconnection problem[11][12][13] and stated the possibilities of such a design. In [14] network platforms consisting of processing and storage elements and communication channels are presented. Channels are the architecture resources in the design library and are identified by bandwidth, delay, area, power consumption and error rate. In [15] our structural and flexible communication scheme that is based on a library of pre-designed communication blocks is presented. In this approach the computational parts of the design are called functional IPs. Similarly, the building blocks for the communication network are interconnect IPs.

2. Framework

Our target domain is that of heterogeneous systems, with many different types of IPs coworking in the same chip. One of our most important goals is to design a highly scalable network, both in terms of number of nodes (interconnect IPs) and in performance. Latency and bandwidth goals are set to less than 1 ms and up to 2 Gb/s, respectively. We do not want to focus (initially) on any specific application and our desire is to create a general purpose NoC. This requires scalability and programmability.

As we want our NoC to have an interface with the unlimited number of available cores, the Virtual Component Interface (VCI) recommendation from VSIA was adopted as exemplifying the current bus-oriented standards for interface design. This imposes a series of constraints in our design, the principal being the adoption of a memory-mapped communication model, instead of a more natural node-to-node approach. Effectively, the Proteo network is seen as a bus from the point of view of the functional blocks connecting through it.

While the use of a custom interconnection scheme for a specific very high performance application may be a better solution, there will still be many situations in which a trade-off between performance and the time-to-market reduction achieved by using a predesigned network may be determinant. It is presented that in systems greater than 8 cores the NoC outperforms the bus [16].

2.1 System Design Methodology and Proteo

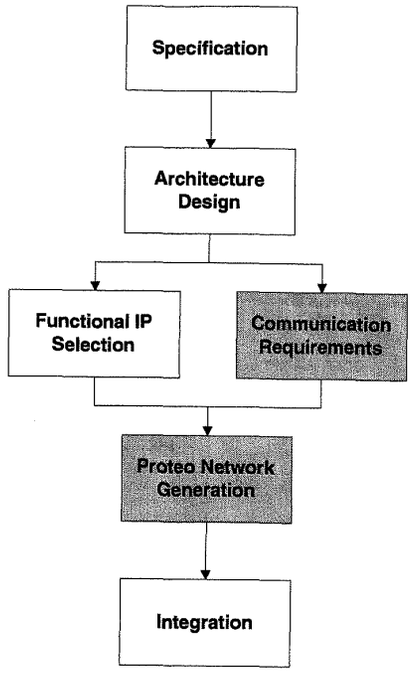


Figure 10.1. Design cycle in our approach.

In figure 10.1, we present a very simple diagram of what a design methodology incorporating a customizable NoC may look like. It is based on the availability of a software environment and a library of network components. The main difference with a more traditional design cycle is the explicit statement of communication requirements, in terms of bandwidth, Quality of Service (QoS) and maximum delay for every block. These requirements are used by a software tool to select instances of the component library and an adequate topology. How the software tool generates the correct network is the subject of another project that is bound to start next year in our group, temporarily called *OIL*. Its goal is to define a series of steps in the design of SoC and NoC, and specify the tool support needed for implementing them. This includes development of description methods and languages for NoCs. In this automated design scheme, Proteo will be the low level component library used for the final synthesis of the resulting system.

The Proteo library consists of two kinds of components for synthesis: nodes and links. Nodes are highly configurable blocks which implement the data layer and the transport layer of the OSI reference model [17]. The links are asynchronous elements implementing the physical layer. They are being designed by another team and fall outside of the scope of this chapter. For validation and simulation purposes, several other components will be available, e.g., abstract models of nodes and links together with bus functional models (BFM) of components to be attached to the network.

The high level design tool will have freedom for customizing several aspects for each of the blocks. Nodes can be customized by sizing their internal buffers, enabling/disabling protocol features, etc., depending on the features of the host interface and the desired network properties. Links will be customized, e.g., in their width, signaling levels, fault tolerant characteristics, and power consumption.

At the moment we use a high level model of the network programmed in VHDL for the design exploration. Similarly, the synthesizable blocks are described using VHDL, so it is possible to easily co-simulate both VHDL models and also reuse of test-benches for validation. However, it seems that for better integration with high level tools like OIL, another version based on SystemC will have to be developed in the future.

Since configuring large networks by hand is almost impossible, we have developed some simple scripts to ease the tasks. At the same time they force us to structure our VHDL code in a "tool-friendly" way. The scripts are being programmed in Python and Perl.

3. Proteo architecture

3.1 Overview

Future large SoC designs will implement the *Globally-Asynchronous Locally-Synchronous* (GALS) paradigm, in which different subsystems will use unrelated clocks and communicate asynchronously[18]. Since most available functional IPs are currently designed as synchronous blocks, they will require a wrapper to interface the asynchronous environment. In figure 10.2 we have represented the functional IP and its wrapper.

The basic hardware elements in our network are:

Qualcomm Exhibit 1006

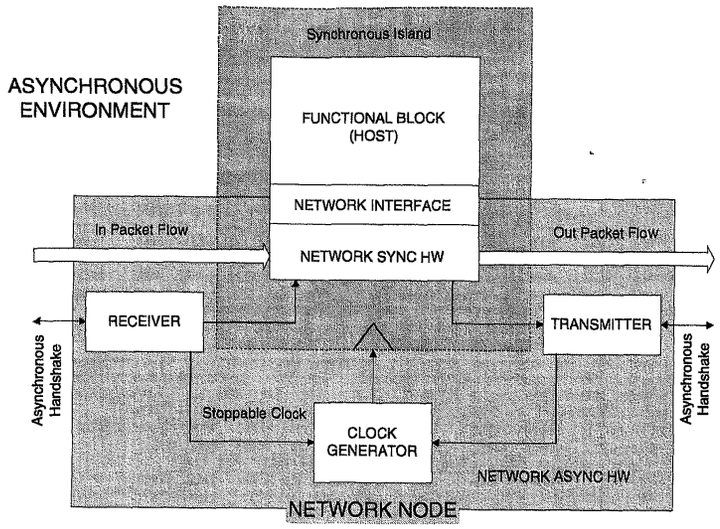


Figure 10.2. Network Node Architecture.

Hosts Every host corresponds to a functional IP[1] that will be connected to the network using a dedicated node as a wrapper. Any host compliant with the core VSIA-AVCI recommendation is supported at the moment.

Nodes Nodes regulate the interaction of the different packet flows in the network and interface the synchronous and asynchronous domains.

Links Links are the elements that actually move the information from node to node in the network in a clock-independent manner.

The system is divided in clusters, each cluster containing related functional IPs that share a common clock and performance requirements. The clusters are interconnected using a hierarchical network. This will comprise multiple subnets with different performance, topologies, packet formats, etc. An example topology being explored is a hierarchical network built from a system-wide bi-directional ring and several subnets with ring, star, or bus topology. The topology is represented in figure 10.3. Other studied structures are regular trees and meshes. The use

[1]Or a multicomponent subsystem, possibly using a different interconnection scheme internally. The essential point is that it presents a VCI-compliant interface to the Proteo network.

of regular topologies allow easy routing and direct replication of blocks throughout the system.

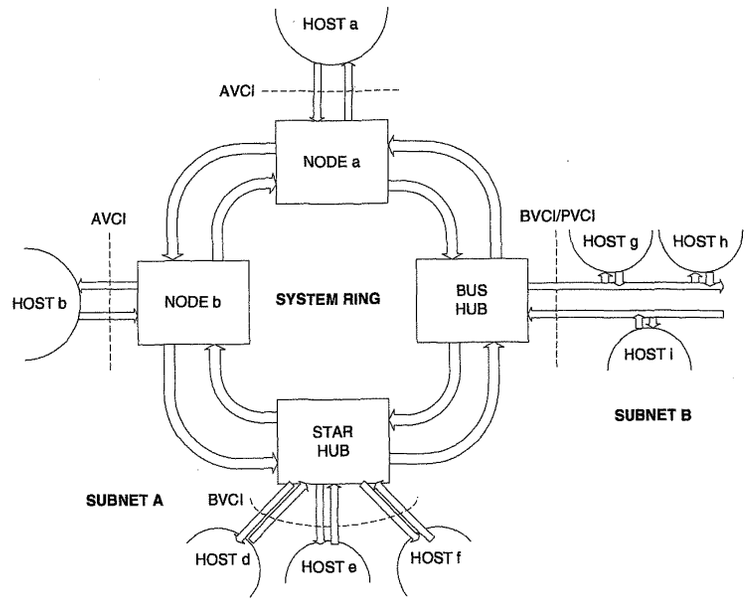


Figure 10.3. Example Proteo network topology.

3.2 Hardware elements

3.2.1 Nodes. The functionality of a node can be viewed as multiplexing and demultiplexing three different data flows: *input*, *output* and *bypass* flows. In the basic design we include a *FIFO* buffer for each of these flows, and a *link multiplexer* or a *link demultiplexer* block at each confluence point (see figure 10.4). The link-mux block distributes the block's output bandwidth among the output flow and the bypass flow. The link-demux block extracts the packets addressed to the node from the incoming flow, while sending the rest through the bypass path.

This set of blocks form what we call a *port-module* block. A basic node architecture represented in figure 10.5 is built using one port-module and an interface block. This interface block translates interface signals of the functional IP's (host) to the network packet format. All components are implemented as independent IP blocks. For the moment we have only implemented a VCI-compliant interface block, but other interfaces are

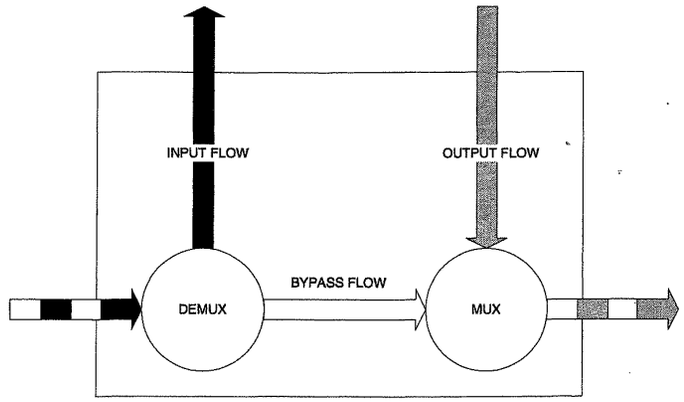


Figure 10.4. Basic node functionality viewed as a mux/demux of packet flows.

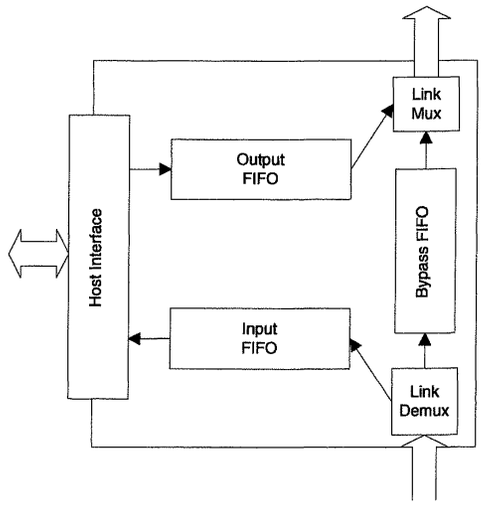


Figure 10.5. Basic Node Architecture.

also possible. Nodes are currently totally synchronous blocks, but we are investigating whether it would be beneficial to implement some of their components using asynchronous logic.

A more complex version of the node provides support for multi-dimensional networks (figure 10.6). Its modular structure consist of the interface block and several port-module blocks (called *layers*), together with a *node router* and a *node multiplexer*. The router block contains a configurable routing table used to decide to which port-module to send each packet forwarded by the interface block. The node-mux block selects the next packet from the ones offered by the pool of port-modules, based on some local priority scheme. Currently it uses a round-robin policy. The configuration parameters of an advanced node include the number of layers, FIFO geometry, supported protocols and packet formats. In general lower dimensional networks seem to work more efficiently than high-dimensional networks, so we expect to find more applications for the smaller nodes.

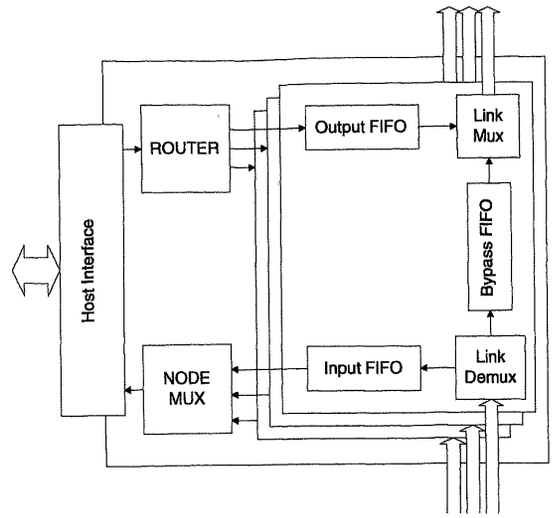


Figure 10.6. Advanced node architecture.

The basic node architecture enables only unidirectional data flow through the node. When bidirectional topologies are needed, two layers are used: one layer takes care of traffic in one direction, e.g., clockwise in rings, while the other layer handles the traffic in the opposite direction. In the ring topology, the router decision consists basically in deciding whether to send the packet clockwise or counter clockwise.

The architectures just presented limits the number of possible network topologies. For example, tree networks are not efficiently implementable using these kind of nodes. An alternative architecture for the port module block that decouples the number of inputs and the number of outputs is being developed. The main difference between the "normal" port module and the alternative one is the addition of optimized extra routing hardware in the latter. In the alternative (multi I/O) node

the routing functions are implemented in one layer that makes possible smaller and more efficient logic.

3.2.2 Floating nodes. For increased connectivity, nodes that are not attached to any host are allowed in our network. These nodes are called *floating nodes*. A basic floating node architecture can be built deleting the host interface of the "normal" node architecture and interconnecting several port modules together using slightly modified router and node-mux blocks. A dedicated architecture that allows further optimizations is being investigated at the moment.

3.2.3 Links. Links are asynchronous structures. They provide a high level interface, so they can be treated as modular elements and tuned independently.

Links interface the synchronous and the asynchronous parts of the network, using a special technique based on clock-stopping, similar to the one described in [18], to avoid metastability.

3.3 Protocols and transactions

The communication inside Proteo is built on top of point-to-point connections. The current library of synthesizable Proteo blocks supports several communication protocols. All current protocols are based on request-response cycle, where each transmitted request packet requires a response packet. A virtual circuit-switching protocol is being investigated, but it has not been fully implemented yet. Each functional IP (host) in the network can be either an initiator component which starts the communication cycle or a target that responds after a received request.

In the simplest protocol each transmitted packet includes exactly the amount of data that can be moved from the host to the Proteo node in one clock cycle. The advanced protocol can collect data during several clock cycles and build one packet with a larger data field. Each request collected to the same packet must be targeted to the same IP.

The advanced protocol cycle (figure 10.7) consists of the following steps:

- The initiator presents a request at its interface.
- The interface block acknowledges the request.
- If the initiator is generating more requests, i.e. issuing a packet, the node will aggregate as many requests as possible as defined by

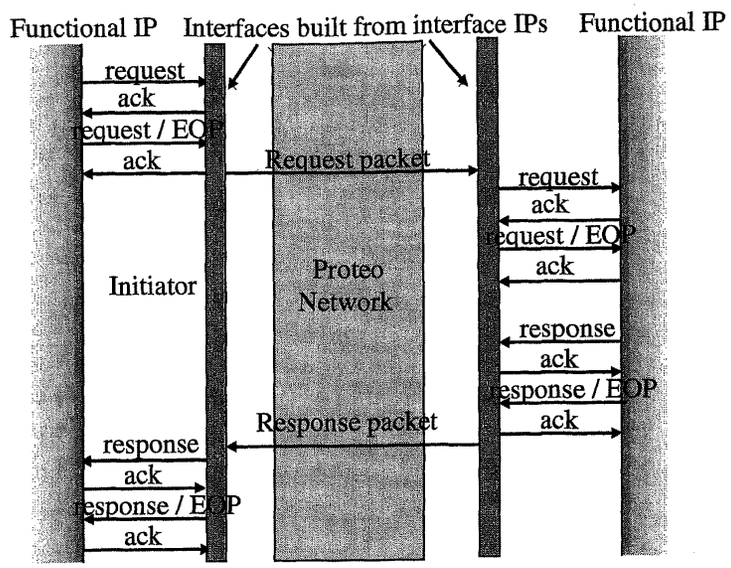


Figure 10.7. Advanced transaction model.

a network parameter that determines the maximum packet size or until the `End-of-Packet` (EOP) interface signal is asserted.

- One packet containing the requests is transmitted through Proteo.
- The node connected to the target host receives the packet. At clock cycle, the interface block sends a request to the target IP and a successful read by the target is acknowledged.
- The target generates a response for each request.
- One response packet (including one or several responses) is generated in the interface and sent to the initiator.
- The responses are extracted from the response packet and transferred to the initiator.

If the target node is not capable of delivering or storing the incoming packet, it can generate a dedicated re-send packet. The initiator node will resend the request packet, if configured to do so[2]. Other errors will

[2]There can be an application in which it is preferable to just discard the packet.

be notified to the initiator host, which is responsible for appropriate action.

Inside the Proteo network, the packets are processed with wormhole routing [19]. In one router the start of the packet is routed forward immediately before the whole packet has arrived if there is a free output link. If free outputs are not available, packets are stored to FIFO buffers. The amount of buffering is one generic design parameter.

3.4 Packet format

Packets create a communication overhead. Therefore Proteo packets are kept as simple as possible. A packet consist of two areas: the Proteo header and the payload. The header fields are needed to deliver data from the source node to the target node. The following fields can be part of a Proteo header:

Upper Address is used to route the request packet through the network.

Source and Destination Identifiers are fields used for identifying the initiator and target nodes.

Command defines the type of packet. Supported packet types are read, write, read response, write response and re-send.

Packet Identifier is used to order the packets, in case they arrive in a different order than they were sent. This can happen, e.g., when a packet is lost due to full input FIFO.

The payload section carries the data and the rest of the address as generated by the initiator host. Other VCI signals, like *BE*, can be included in the payload section to allow the reconstruction of the original interface signals at the target host interface.

The structure of the packet and the length of each field are configurable at design time. In figure 10.8 a possible packet format is shown.

Besides the packet bus, there are additional signals for the asynchronous handshake and sideband signaling. Sideband signaling is used to mark the start and end of a Proteo packet.

4. Node Implementation

The implementation library of Proteo contains customizable synthesizable VHDL nodes. These interconnect IPs can be classified in several ways:

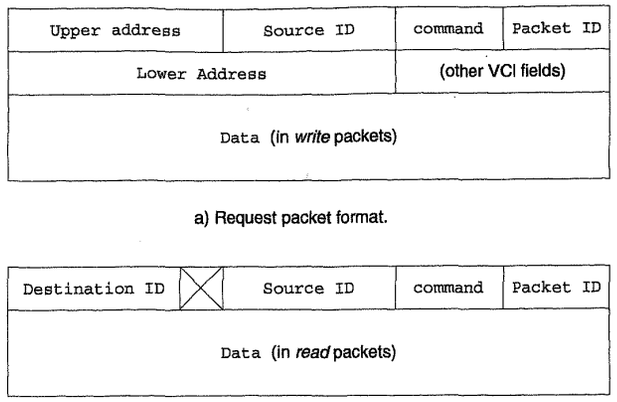


Figure 10.8. Example Proteo packet formats.

Functionality Nodes can be attached to a functional IP (host) or they can be placed alone in network (floating node), forming a special structure used to interconnect several subnets.

Number of links The number of Input/Output links is variable. We have explained that certain blocks of the node architecture may vary accordingly.

Protocol options The current Proteo library supports three protocols. Two of those uses packets which have one interface cell in a packet, the third protocol carries several cells of data in one packet. Additionally, options like the generation/checking of a CRC field can be activated.

4.1 Structure of FIFOs

The FIFO buffers inside the Proteo node are synchronous register banks. The size of the buffer is defined by two generics, FIFO width and length. Width and length define the number on parallel bytes in a packet (internal word width of FIFO) and how many maximum sized packets the buffer can store, respectively. That is, currently the designer defines the FIFO size using the amount of packets, not bytes or words. Though it is already noticed that finer grain tuning is needed. Buffers are constructed from two subblocks, control logic and register bank, as seen in figure 10.9. The register bank is the actual storage structure where the

control updates read and write pointers. Incoming data is stored when `WriteRequest` signal is asserted and there is space available. `NotFull` signal tells that there is space in the buffer and `NotEmpty` that there is some data available in the output of the FIFO. Output and Input FIFOs are also accessed through a control port which is used to deliver re-send and delete commands. Because there must be a response for each transmitted request packet, the request is not deleted from the Output FIFO until the response has arrived to the interface. If the response is late, the Output FIFO receives a re-send command from the interface block (timeout). Data can be deleted from the Input FIFO, e.g. when the tail of the incoming packet is lost due to full Input FIFO. In that situation the start of the packet will be deleted. On the other hand, the packets arriving to the Input FIFO are never re-sent so Input FIFOs do not support re-send command.

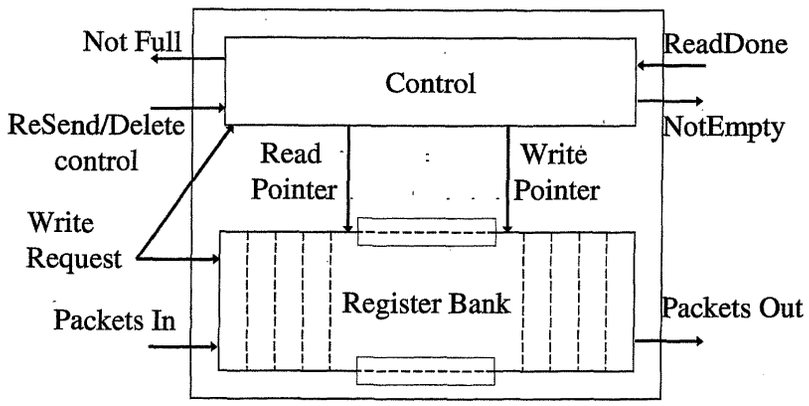


Figure 10.9. Structure of Proteo FIFO.

4.2 Link Demultiplexer

The link demultiplexer receives packets through the input link. First it detects the *destination ID* or the *upper address* fields of the packet. They are compared against its own ID or its table of acceptable addresses. If the test is positive the packet is written to the input FIFO and later forwarded to the interface block and host. In those cases where the test is negative the packet is written to the Bypass FIFO. The selection process is implemented in two subblocks, *Detector* and *Control*. The detector tracks the start and end of packets, so it can locate the appropriate

header fields, and sends them to the control block, which performs the test.

4.3 Link Multiplexer

The link multiplexer is used to transmit packets from the FIFOs to the output link. The multiplexer waits until there is a packet or several packets in the FIFOs and after that it forwards one packet to the network. The Link Multiplexer has a predefined priority that is used to determine how packets are routed from the FIFOs to the output link. The priority is fixed when the network is synthesized. The default priority favours the bypass FIFO over the output FIFO. This is used to prevent the generation of new traffic until the old traffic is processed.

4.4 Design Parameters

Each node implementation is based on generic parameters which define the physical structure of the network. The parameters (Table 10.1) are fixed at compile time. Route limits define the routing table in each node: each output link has an associated address range (or a node identifiers list). If the upper address field (or the destination ID) of the outgoing packet is included in those limits, the packet is routed through that link. More parameters will be added in the future to allow more detailed customization of the components.

4.5 Synthesis of Basic Node

There exist two implementation of interface IPs because the VCI interface defines different properties for the initiator and target of point-to-point communication. The following table (10.2) presents gate-level area costs of the initiator and target node, respectively. The cost estimates are generated using synthesis and 0.18 μm silicon process. The nodes have 2 layers with four network links in total.

It can been seen that the FIFOs consume a very significant area. In the previous example the initiator node had 560 flip-flops in Input, Output and Bypass FIFOs and the target node 512 FFs. That is together 1072 FFs which are consuming 81% of the silicon area in the node pair implementation. Using memories it should be possible to reduce the area of FIFOs significantly compared to the current situation. However, memory based FIFOs are not implement yet. Until network node implementations with memories are available, buffer sizing is the most important single issue when optimal implementation is searched.

Table 10.1. Design parameters

VCI Width	Number of bytes transferred per clock cycle in the interface between network and host
Address Width	Number of bits in the address field
PLEN Width	Number of bits in Packet Length -signal in the interface
CLEN Width	Number of bits in Chain Length -signal in the interface
ERROR Width	Number of bits in Response Error -signal in the interface
Word Width	Word width inside network elements
ID Length	Length of Source ID and Destination ID / Address fields in packets
PacketID Length	Length of PacketID field in packets
NodeID	Defines the ID of node
Output FIFO Depth	How many maximum-sized packets can be stored in the output FIFOs of the node
Input FIFO Depth	How many maximum-sized packets can be stored in the input FIFOs of the node
Bypass FIFO Depth	How many maximum-sized packets can be stored in the bypass FIFOs of the node
Resend Time	How many clock cycles nodes will wait until they resend packet if no response has been received (timeout)
Layers	Number of layers (only in multilayer nodes)
Route Limit Up	Upper address limit for each output link.
Route Limit Down	Lower address limit for each output link.

Table 10.2. Implementation costs of basic Proteo nodes in mm^2

	Initiator	Target
Interface block	0.009	0.011
Output FIFO	0.033	0.027
Interface Demultiplexer	0.003	0.003
Interface Multiplexer	0	0
Link Demultiplexer	0.001	0.001
Input FIFO	0.027	0.034
Bypass FIFO	0.034	0.027
Error Checker	0.004	0.016
Link Multiplexer	0	0
Entire Node (with 2 layers)	0.179	0.198

4.6 Test Networks

Using the Proteo IPs it is very straightforward to build ring networks and connect those together. In figure 10.10 a synthesizable test network

using Proteo IPs is presented. The network has 8 simple processing elements (masters) and 2 memories (slaves) in the system. Each functional IP is connected to the NoC through an interface block. Each network node is a layered node with two layers.

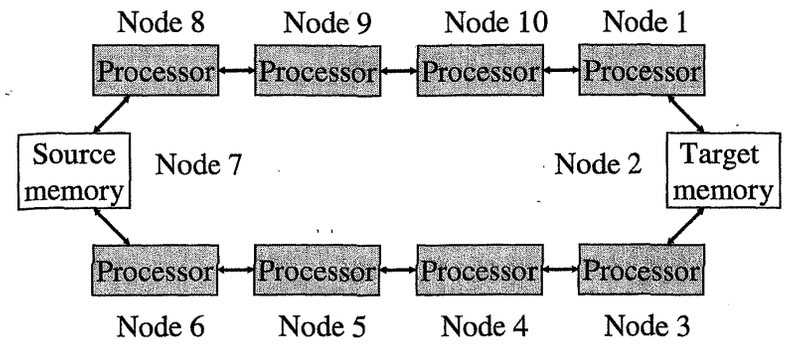


Figure 10.10. Test network with ten functional IPs.

The constructed network was verified with a test case where each processor read a picture (100 pixels) from the source memory, converted it to a negative image, and wrote it to the target memory. The transfer times are proportional to the payload length. E.g., when the payload length is 2 words (i.e., 2 pixels), one picture is transferred from one location to another with 50 data packets. In total there would be 100 packets, because each connection always has a request and a response.

4.6.1 Performance and Implementation Costs.

The needed number of clock cycles to do all the transfers as a function of the data size of the packet are presented in figure 10.11. In VCI-1 there is one byte in each VCI cell. Respectively, VCI-2 and VCI-4 have 2 and 4 bytes. Thus a VCI-4 packet with 3 cells is carrying totally 12 bytes of payload data. In all the test cases the word width inside the Proteo network is 8 bits. It can be seen that when the payload size is increased, the improvement rate of performance saturates. This is caused by the fact that the memory models used cannot serve arriving read an write requests any faster.

The whole network was synthesized using 0.18 μm standard cell technology. The FIFO sizes were selected to allow the saving of one complete packet in each buffer. The size of a packet depends on the size and amount of VCI cells transmitted inside the packet. The resulting cir-

Qualcomm Exhibit 1006

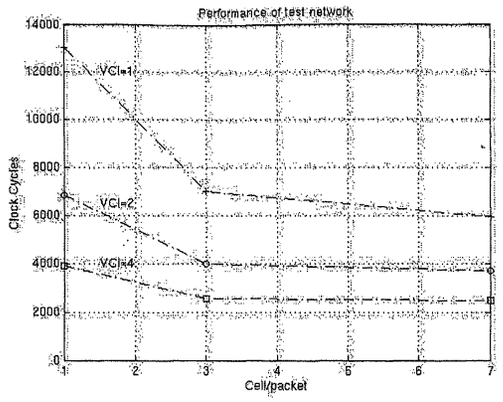


Figure 10.11. Performance of a bidirectional network with different payloads.

cuit area of the networks with different packet-sizes based on gate-level netlist for test is presented in figure 10.12.

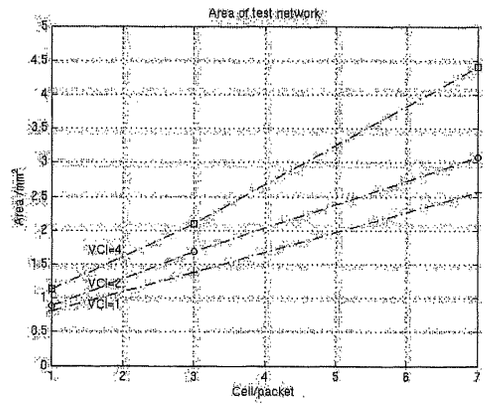


Figure 10.12. Area of 2 layered network with different payloads.

The total costs of communication in Proteo are illustrated multiplying the achieved transfer capacity with the hardware costs. The combined cost figure is presented in figure 10.13. There are relatively clear valleys in the curves, suggesting that in these application the payload size should be chosen within that optimal cost/performance ratio is achieved.

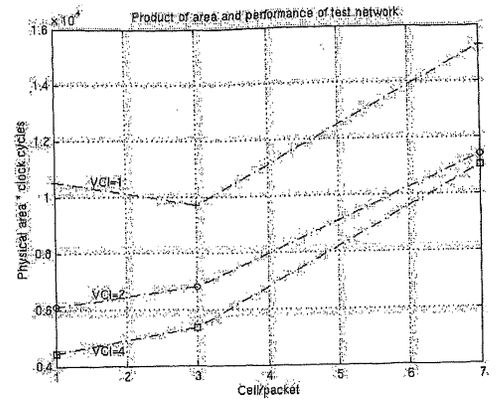


Figure 10.13. Product of area and performance in 2 layered network with different payloads.

Another parameter which was monitored during the simulation was the utilization of the bypass FIFOs. As long as the bypass FIFOs are not full, the network is capable to transfer packets forward and the possibility of packet loss in the network is small. However, there is still a chance that data is lost at the input FIFOs if the functional IPs are not processing the arriving packets fast enough. It was found that the average Bypass FIFO utilization was under 10% in all the FIFOs. On the other hand, the peak utilization was over 70% in all processor nodes and 35% and 55% in the memory nodes (memories have less bypass traffic). As a result, detailed buffer tuning (allowing depths to be specified with a finer grain) seems to be of interest.

5. Conclusion

We presented our Network-on-Chip project, Proteo. It is a flexible interconnect IP library that is implemented using generic VHDL blocks. The communication inside Proteo is based on point-to-point connections which are using packet based communication in the lower level of the protocol stack. The main elements of Proteo are the port module and the interface block. The implementation of Proteo is controlled through design parameters which define the protocol and structure and size of the node components. The modular architecture chosen seems flexible enough to allow the definition of a range of protocols.

Initial results show that, although the basic principles may be good enough, more resources, like the possibility of creating and using an optimized FIFO block, will be of great benefit to the project.

References

[1] M. Birnbaum and H. Sachs. How VSIA answers the SoC dilemma. *IEEE Computer*, pages 42–49, June 1999.

[2] Virtual component interface standard. http://www.vsi.org, April 2001.

[3] Open Core Protocol Specification. http://www.ocpip.org/home.

[4] A. Ferrari and A. Sangiovanni-Vincentelli. System design: traditional concepts and new paradigms. In *Proceedings of the International Conference on Computer Design: VLSI in Computers and Processors*, Austin, USA, October 1999.

[5] H. Chang et al. *Surviving the SOC Revolution : A Guide to Platform-Based Design.* Kluwer Academic Publishers, 1999.

[6] J. A. Rowson and A. Sangiovanni-Vincentelli. Interface-based design. In *Proceedings of DAC*, Anaheim, California, USA, June 1997.

[7] G. Nicolescu, Sungjoo Yoo, and A. A. Jerraya. Mixed-level cosimulation for fine gradual refinement of communication in soc design. In *Proceedings of DATE*, Munich, Germany, March 2001.

[8] The coreconnect bus architecture. http://www.chip.ibm.com/products/coreconnect/index.html.

[9] Sonics micronetwork: Technical overview. http://www.sonicsinc.com/Pages/Networks.html.

[10] E. Salminen, V. Lahtinen, K. Kuusilinna, and T. Hämäläinen. Overview of bus-based system-on-chip interconnections. In *Proceedings of ISCAS*, Scottsdale, Arizona, USA, May 2002.

[11] A. Hemani, A. Jantsch, S. Kumar, A. Postula, J. berg, M. Millberg, and D. Lindqvist. Network on a chip: An architecture for billion transistor era. In *Proceedings of NORCHIP 2000*, Turku, Finland, November 2000.

[12] P. Guerrier and A. Greiner. A generic architecture for on-chip packet-switched interconnections. In *Proceedings of DATE 2000*, Paris, France, March 2000.

[13] W. J. Dally and B. Towles. Route packets, not wires: On-chip interconnection networks. In *Proceedings of DAC*, Las Vegas, USA, June 2001.

[14] A. Sangiovanni-Vincentelli. Defining platform-based design. http://www.eedesign.com/features/exclusive/OEG20020204S0062.

[15] I. Saastamoinen, T. Suutari, J. Isoaho, and J. Nurmi. Interconnect ip for gigascale system-on-chip. In *Proceedings of ECCTD 2001*, Espoo, Finland, August 2001.

[16] C. A. Zeferino, M. E. Kreutz, L. Carro, and A. A. Susin. Models for Communication Tradeoffs on Systems-on-Chip. In *Proceedings of International Workshop on IP-Based SoC Design*, Grenoble, France, October 2002.

[17] P. Judge. *Open systems: The Basic Guide to OSI and its Implementation.* Computer weekly, 1988.

[18] J. Muttersbach, T. Villiger, H. Kaeslin, N. Felber, and W. Fichtner. Globally-asynchronous locally-synchronous architectures to simplify the design of on-chip systems. In *Proceedings of the 12th Annual IEEE International ASIC/SOC Conference*, Washington DC, USA, September 1999.

[19] Li-Shiuan Peh. *Flow Control and Micro-Architectural Mechanisms for Extending the Performance of Interconnection Networks.* PhD thesis, Stanford University, August 2001.

III

SOFTWARE AND APPLICATION INTERFACES

Qualcomm Exhibit 1006

Chapter 11

BEYOND THE VON NEUMANN MACHINE

Communication as the driving design paradigm for MP-SoC from software to hardware

Eric Verhulst
Eonic Solutions GmbH

Keywords: Semantics, Hardware-software co-design, Distributed real-time, Switch fabrics, links

1. Introduction

While the original von Neumann machine concept reflected the single clock nature of the original hardware, today's hardware and software are very far removed from it. This will be true in particular for MP-SoC where for efficiency reasons different processor types running in different clock domains will co-exist. Hence, this also requires a different way of programming and designing such a MP-SoC. In addition as the communication has become the bottleneck, one should consider an architecture that adds processing blocks as "co-processors" to a communication backbone. Several consequences are highlighted: the need for a communication-oriented specification and programming style, a communication subsystem with real-time QoS as a system service and reconfigurability to cover a wide range of applications with a single MP-SoC.

2. The von Neumann ALU vs. an embedded processor

2.1 When the state-machine goes parallel

Today we program most of the embedded processors in C, which by definition is a high level sequential programming language. Actually in the C syntax one can even clearly see that it was meant to program the underlying sequential ALU at a higher level of abstraction. This un-

A. Jantsch and H. Tenhunen (eds.), Networks on Chip, 217–238.
© 2003 *Kluwer Academic Publishers. Printed in the Netherlands.*

Qualcomm Exhibit 1006

derlying ALU is still very much the same as the original von Neumann machine. The latter was designed as a machine that reads its instructions and data from memory, executes them and stores the results into memory. Most embedded processors however operate in a slightly different mode. They read data from an external device, read their program and some data from memory, and write the results to another external device. Hence we have two operations of communication and only one operation on the ALU as defined by the von Neumann machine. While by memory mapping this whole sequence can be fit to the von Neumann sequential paradigm, it is clear that this leads to a mismatch when the system level parameters are being pushed to the limits. E.g. when the data-rates become very high (compared with the ALU clock speed), the ALU will not be able to keep up. The resulting system level performance even gets worse when the ALU is using pipelining techniques and memory devices start being clocked at a slower rate than the ALU. This brings us to the conclusion that to restore the balance, we need the introduction of parallelism as well as at the level of system architecture as well as at the level of the programming environment that is used. And while parallelism has been introduced piecewise both in hardware and in software, one can wonder why this is not yet the dominant paradigm. The reason in the end is simply history. Computer languages were often designed by computer scientists on workstations, while only embedded systems must process data in real-time. And to make things worse, embedded systems were often designed by hardware engineers. Nevertheless, solutions were found and applied. Time to bring them to the status of a dominant system design paradigm as will be explained in this chapter.

2.2 Multi-tasking

One of the solutions that emerged from the industry was the concept of software multi-tasking. This actually emerged naturally as a solution to the fact that software programs are really only models. In embedded systems, they model often real-world objects and events with the real world being parallel by nature. Its implementation principle is simple. Any executing program thread can be saved and restored by saving and restoring the status of the resources it is using on the processor. These resources are called the "context" and mainly consist of the registers and a task specific workspace. Conformal with the C programming model, each task will normally be a function, although this is not a must. Hence a task can best be viewed as a function with its own context and workspace.

Hence, non-sequential processing is nothing else than a technique that allows switching ownership of the processor between different functions. Besides that it brings the benefits of modularity, this capability actually reduces the overhead in an embedded application as its allows to switch to another task when a task starts idling (typically an "active" wait condition during which the task is waiting for an external event.) The switching itself is the job of the "kernel" or "operating system". Given the real-time nature of embedded applications we will stick to the term RTOS (Real-Time Operating System). The next problem to solve is to have a mechanism that allows scheduling the execution of the tasks in a way that they all still meet their real-time requirements, often an essential feature of an embedded system.

Scheduling algorithms. As this paper is not meant as an exhaustive overview of scheduling techniques, we will outline the dominant scheduling paradigms. See [2] and other publications of Kluwer for a more comprehensive overview.

Control-flow based scheduling. Control-flow based scheduling, also often called event-driven scheduling, is used when the tasks need to execute following the arrival of specific often asynchronous events or triggers. Following the events, the tasks must reach a point, called its "deadline" in order for the system to meet its real-time constraints. As the name implies, such systems are often found in embedded systems where the processor is used as a controller. A typical example is e.g. an airbag controller. The issues in such systems are mostly latency issues and the complexity when the system has multiple events at its inputs. The problem is to make sure that all system states are covered.

Data-flow based scheduling. Data-flow based systems can be seen as a superset of control-flow based systems. The main difference is that the event is not just a logical event (e.g. data has arrived) but that the arrival time of the data becomes significant. The data-rates can be very high, which means that this leave less time for the actual processing. In addition, the processing will often be much more compute intensive than in control-flow systems where the dominant type of processing is decision logic. Dataflow dominated applications are typically done with DSP processors. In dataflow-based systems, the complexity comes from handling data streams that have different arrival rates.

Time-triggered scheduling. If the arrival times of the events and data-streams can be known beforehand (e.g. because everything is

Qualcomm Exhibit 1006

driven by a known clock), one can calculate the timeslots available and needed for all tasks to reach their deadlines before the program starts executing. The pre-conditions of predictability at the event side and the stationary behavior of the system for time-triggered scheduling (a lesser form is often also called static or synchronous scheduling) are quite severe as it leaves little room to handle "asynchronous" events. But it has the major benefit that its leads to a predictable behavior, even when the CPU is loaded close to its maximum. The latter is essential for any form of safety-critical systems.

Dynamic scheduling. The dominant scheduling paradigm is based on a form of dynamic scheduling. In such systems, the decision to schedule (read: start or resume execution) a task will be done at runtime. The most widely used scheduling algorithm is based on RMA (Rate Monotonic Analysis). In this algorithm tasks get a priority proportional with their scheduling frequencies. Given a number of additional boundary conditions, it has been proven that tasks (at least in the case of a monoprocessor) will meet their deadlines if the total CPU workload remains below about 70 percent. In practice, the latter figure can, depending on the application, reach close to 100 percent. Hence, such a scheduler will use the priority of a task to decide which task to schedule first. Another algorithm is called EDF (Earliest Deadline First). While it has many variants, in these algorithms the time left to reach the deadline itself will be used as the scheduling criterium. It has been proven that EDF performs better and allows to reach all deadlines even when the CPU is loaded at 100 percent. In practice however, only RMA is implemented and used. The reasons for this situation is first that it is more complicated to work with deadlines than with priorities. A major reason for this is that as most processors have no hardware support to measure how far a task has progressed in reaching its deadline, software based implementations have to fall back on a rather course grain software clock, which undermines a lot of the potential benefits in real applications. In addition, RMA based schedulers often degrade gracefully when some tasks miss their deadline, whereas EDF based schedulers often degrade catastrophically. Also, until now, no EDF type algorithm has been found that works well on multi-processor targets.

A final remark however relates to the notion of task scheduling in theory versus the practice. RMA assumes that each task is a fully self-contained function with no interaction with the rest of the system, except at the moment it becomes ready to execute and when it finishes. Hence, a task has only two (de)scheduling points. Often, this is written in a loop as this avoids that the task needs to be reinitialized for

the next execution. In practice however, most tasks are written with multiple "descheduling" points. E.g. a task will first initialize some data-structures and then often start an endless loop. The loop however can contain multiple points where the task synchronizes and communicates using the kernel services, and hence can deschedule. The code segments between these descheduling points are the relevant ones for the scheduling, not the full task.

Real Embedded systems. Real embedded systems will depending on their complexity often be a combination of one or more of the above. Most systems have to react to asynchronous events as well to timer triggered ones, while more and more systems have to process an increasing amount of real-time data coming from e.g. sensors. In all cases will a good system design distinguish between the different functions of the application before scheduling is introduced. As such, it is important to point out that the different scheduling paradigms are often more a matter of style and implementation, At the conceptual level they are manifestations of the same: how to allocate the processor resources over time to the functions of an application. Another important conclusion is that scheduling and multi-tasking are orthogonal issues. Violating this always leads to unnecessary complexity and hence reliability issues. As will be put forward in the rest of this paper, in general system design and MP-SoC in particular will benefit from a rigorous application of a separation of concerns. It does not only applies to scheduling and multi-tasking, but also to processing and communication.

3. Why multi-processing for embedded systems?

3.1 Laws of diminishing return

Modern processors seem to be providing more performance than we could even imagine some years ago. Clock speeds are now in the GHz range but did the system level performance follow? It should be noted that decreasing the silicon features, which allows higher clock rates and higher densities on the chip, made most of these advances. At the same time, often micro-parallel "tricks" have increased the peak performance. E.g. pipelining, VLIW, out of order execution and branch prediction allow even to process instructions at a virtual clock rate higher than the real clock rate. While such solutions seem to be adequate for desktop and other general purpose computing systems where throughput is often the goal, for embedded applications they pose a serious challenge. E.g. embedded applications have often severe power consumption constraints, whereas the power consumption increases more than linearly with the

Qualcomm Exhibit 1006

clock speed. In addition the micro-parallel tricks of above are very much application dependent to provide a better performance. As these tricks require a lot of extra gates, it also increases the cost price. Last but not least, they increase the mismatch between the internal ALU frequency and the external world. Hence, I/O and memory become the bottleneck to reach the peak performance. Although internal zero wait state memory has bee introduced, when the application does not fit in it, the performance penalty for running from external memory can easily be a factor of 20. As for the real-time predictability, CPU designers have introduced fast cache memories to increase the locality of the data for the ALU, but these caches make predictable real-time scheduling extremely difficult.

Hence, the solution seems simple. Rather than clocking a processor at high speed, use the advances in silicon technology to use multiple, hence smaller, processors at a speed that is matched with the access speed to I/O and memory. It is clear that if these work together they can achieve a higher system performance at lower power. The catch is that this requires also an adequate interprocessor communication mechanism and adequate support in the programming environment. In addition, it should be noted that while scheduling, task partitioning and communication are inter-dependent, the scheduling is not orthogonal to the mapping of the tasks on the different processors - not necessarily of the same type - and the inter-processor communication backbone. Hence, the communication hardware should be designed with no bottlenecks and in the ideal case, even to be reconfigurable to match a given application. Note that e.g. FPGA chips have taken a similar approach at a very small grain level.

3.2 A task as a unit of abstraction

Tasks as virtual processors. Although multi-tasking started as a solution to a hardware limitation, it can also be used as a programming paradigm. Because a task is a function with its local context and workspace, it also acts as a unit of abstraction in a larger multi-tasking system. The RTOS will shield the tasks (at least when properly programmed) from each other and hence a task has an encapsulated behavior. In order to build real systems, one needs more. Tasks need to synchronize, pass data to each other and share common resources. These are services provided by the RTOS. For the critical reader, the terminology used in the market is not always consistent. E.g. in some domains, the terms multi-threading and processes are used. In the embedded world, a thread is often a lightweight task that shares the workspace

with other threads created by a common parent task. As this allows sharing data by direct reference, it can be a convenient but not always a safe programming technique. Hence, it is to be avoided. The process concept is often interchangeable for the task concept. We will use both terms depending on the context of the discussion.

Given that a task has a consistent processor context, a task can be considered as a virtual processor with the RTOS services providing logical connections, rather than physical ones. But important is to see that such a multi-tasking environment can emulate a real multi-processing system (at least at the logical level). If strict orthogonality between multi-tasking and scheduling is respected, one can see that the main difference between the two domains is mostly the behavior in time. We assume here a correct and time-invariant implementation of the synchronization logic.

A theoretical base for multi-task programming. As multi-task programming was created and evolved from a pure industrial background, the reality suffers from a number of problems. Most RTOS on the market don't have very clean "semantics". With the latter, it is meant that often for opportunistic performance reasons, the behavior of the RTOS services have side effects. E.g. rather than passing data, RTOS services will pass pointers to the data or the RTOS will provide a large set of complex services - often with little difference in the behavior - to provide communication services. The result is that the careful programmer either has to avoid using certain services of the RTOS or worse, he has to resort to so-called co-operative multi-tasking. This violates the first principle of orthogonality and is a prime source of program errors. There is however theoretical work, not always well understood that provides a consistent base for reasoning about multi-tasking. Although there are contenders, the most influential one was the CSP programming model from C.A.R Hoare at the University of Oxford [1]. CSP stands for Communicating Sequential Processes and is an abstract formal language for defining parallel programs. Let's note from the start that CSP is timeless, although later on work was done on Timed CSP trying to address real-time issues. The fundamental concepts behind CSP are simple: a program (read a model of the real world) is composed of a set of processes (inherently sequential but allowing hierarchical composition) and synchronous communication channels. The major merit of CSP is that it proves that one can formally reason and provide proof about the correctness of parallel programs. Of course, as formal methods have their limits, CSP semantics are simple compared

with the real world and this was perhaps the reason that the general programmer's public was not always enthusiastic about it.

Proof of the pudding : the transputer and occam. Nevertheless, the CSP idea was the basis of a processor concept, called the INMOS transputer that had its own special language called occam [4, 5, 3]. The transputer itself was a revolutionary processor, but like many it had a hard time to convince the mainstream market. At the basis was the concept of CSP inspired processes and channels. In order to have "cheap" processes and context switching, the transputer maintained two lists of processes (each running at a different priority) in hardware. Processes could communicate through synchronous channels. In order to have "cheap" processes, the ALU had a 3 deep stack of registers rather than a large register set, the contrary of today's "RISC" (Reduced Instruction Set Computer) processors. Channel communication was also supported in the hardware and used in a homogenous way to communicate between the processes, to read from the event channel (read: interrupt), to read the timer hardware and the interprocessor links, the latter fully supported in hardware with DMA. The transputer was also supported by a CSP inspired programming language called occam. Very strict on typing and semantics, occam is a very readable language compared with the very mathematical and formal CSP. It also features processes and channels. While it can take some time to adjust to thinking in terms of communicating processes (e.g. to program in a deadlock free manner), once mastered, it is a unique experience to have programs that run after the first time the compilation phase was passed successfully. Although the transputer and occam in the end were commercial failures, they are living proof that the constrictions of the von Neumann model can be overcome. Note however, that we take CSP as a model in the generic sense, not to be taken literally in all its details.

There is only programs. In order to illustrate that there are no logical reason not to make the switch to a CSP based programming model, we will take a very simple example. This example is the assignment statement a := b. In a semi-formal way, the assignment can be defined as follows :

```
BEFORE := a = UNDEFINED b = VALUE (b)
AFTER  := a = VALUE (b) b = VALUE (b)
```

The generic implementation in a typical von Neumann machine (but also in a mono-processor transputer system) is as follows :

```
LOAD b, register X
```



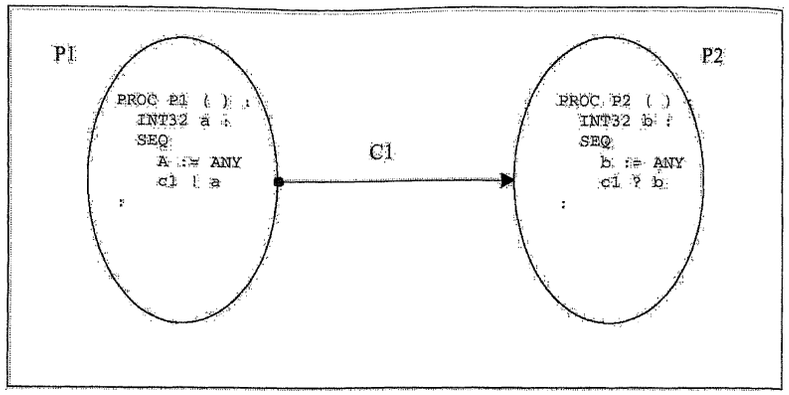
```
STORE X, a
```

How would this be expressed and implemented on a CSP machine? For the sake of clarity, let's express this as a sequential and a parallel occam program. The sequential version is simple :

```
SEQ
  INT32 a,b :
  a := ANY
  b := ANY
  b := a
```

The "SEQ" statement means that the statements in its scope need to be executed in sequence. The statements in the scope start and end with two space indentations.

Qualcomm Exhibit 1006



The parallel version in occam reads as follows:

```
PROC P1, P2 ://define two processes P1 and P2
   CHAN OF INT32 c1  ://define a communication channel

    PAR       // "PAR" means that the statements in
     P1 (c1) // scope are to be executed in "parallel";
     P2 (c1) // just connect the channels and execute
    :        // the processes

   // P1 and P2 are defined as follows :

   PROC P1 ( )
   INT32 a :
   SEQ
     A := ANY
     c1 ! a  // output (=write) to channel c1
   :

   PROC P2 ( )
   INT32 b :
   SEQ
     b := ANY
     c1 ? b  // input (=read) from channel c2
   :
```

Two things should be noted in this occam program. Firstly, there is no assumption about the order of execution. This means that the processes are self-synchronizing through the communication. They execute as far

as they can go and suspend until the communication has happened. And secondly, the sequential version is logically equivalent with the parallel version. The latter is a very important, although not often well understood conclusion. It basically shows that sequential programming and parallel programming are equivalent operations with differences related to the implementation. Let's examine this more in detail by looking at the parallel version as implemented on a von Neuman machine (which the transputer still is).

```
PROC P1 :
  LOAD b, register X
  STORE X, output register
  // hidden "kernel action" : start channel transfer
  // suspend PROC P1

PROC P2 :
  // hidden "kernel" action : detect channel transfer
  // transfer control to PROC P2
  LOAD input_register, X
  Store X, b
```

This same parallel program with P1 and P2 executed on the same processor can be optimized by mapping the input and output registers to an address in common memory. If two processors are involved, an intermediate communication channel (often called link) that acts as a buffer will be needed. But from a global point of view, one can see that the pure sequential assignment is actually an optimized version of the parallel one. Two remarks are needed. Firstly, the channel synchronization adds an extra feature: protection. The process data is local and a copy is physically transferred from one process to another. Secondly, one could argue that this is at the price of a serious performance degradation. In practice however, this is a matter of granularity and architecture. E.g. on the transputer at 20 MHz, the context switch was in the order of 1 microsecond, which made processes and channel communication rather cheap. Actually, as soon as each processor has a small number of processes, the overhead is often masked out because communication and process execution overlap in time. In addition, the overhead drops proportionally with the size of the data-transfer.

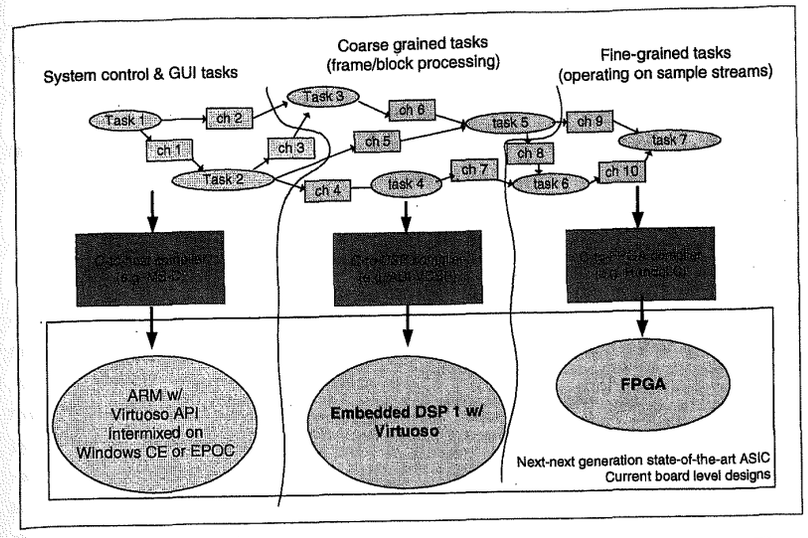
CSP semantics for programming micro-parallel hardware. The CSP semantics become even more interesting when used with re-programmable micro-parallel hardware as found in FPGA (Field Programmable Gate Array) chips. A unique example is Handel-C (available

from Celoxica) that can be used to program FPGA in C, but with CSP-like extensions. The following program segment illustrates this :

```
par // will generate parallel HW => 1 clock cycle
  chan chan_between;
  int a, b;
  {chan_between ! a
   chan_between ? b
  }

seq // will generate sequential HW => 2 clock cycles
  chan chan_between;
  int a, b;
  {chan_between ! a
   chan_between ? b
  }
```

In this case, the sequential version will even be slower than the parallel version. Sequentialisation on a FPGA will be done when the parallel version fans out and needs more logical elements than available. Hence, this is a bit the opposite of loop unrolling often done on sequential processors. This example also illustrates that current sequential programming languages like c (and others still reflecting the original von Neumann machine) are inadequate. By forcing programmers to express the program in a sequential way, actual information is lost. However with a CSP like approach, one only needs to add mechanisms to express the concurrency and communication. Inside a process the program remains sequential. The major difference in the programming style is that one should start programming with no global variables, as this is an implicit optimization and violation of the CSP semantics. For all purposes, a program - seen as a system specification - should be defined as much as possible as a set of (fairly small) processes, with sequentialisation being seen as an optimization technique on a sequential machine. Note that this also applies to chip level design. Most chips today are still designed to run with a global single clock, even if the chip is composed of multiple functional units. In the figure below, such a scheme is presented. Note that this is already possible today.



Lessons for MP-SoC design. As semiconductor features continue to shrink and as there are many reasons to believe future chips will be composed of multiple functional units, of which a large number of programmable processors, one can clearly see the benefits of the CSP-like approach. A major benefit will be that by using a common high-level programming language, one can expect a decoupling between the logic expressed in the parallel program and the actual mapping on the parallel hardware.

At this moment, we did not discuss yet what underlying support this requires from the hardware. For the communication, one needs a communication subsystem and automatic, deadlock free routing. To support the parallelism on the same processor, one also needs a process scheduler. While the examples above only communicate single values, in real applications, one needs to communicate data of variable size. This requires in general a packetisation and a buffering mechanism with the data transfer being done with DMA in the background. In a more general-purpose system, each packet will be composed of a header and payload.

More difficult to address are the real-time needs at the communication subsystem. While transferring large data packets gives a better throughput, it has the major drawback that during the data transfer the communication resource is blocked. For dynamic applications, one can use prioritized packet switching. The packetisation will limit the blocking action while the prioritization allows satisfying the real-time

requirements at the system level. In the ideal case, one should have the capability to "pre-empt" an on-going data transfer in favor of a higher priority one. For very critical real-time applications, one can still resort to static or time-triggered scheduling of the communication.

The major conclusion is that for MP-SoC to work, a communication subsystem that satisfies the real-time requirements is a must.

3.3 Economic factors driving the CSP like approach

NRE and time-to-market demand reprogrammability. There are however compelling economic reasons to start adopting a CSP like approach as well. The major reason is that the non-recurrent engineering costs increase proportionally with each shrinking of the chip line features. While these shrinking line features provide smaller, faster and hence cheaper chips for e.g. high volume applications, for high performance applications (often used in dataflow dominated applications with DSP algorithms in smaller volumes) there are problems on various levels : I/O performance, power consumption, electromagnetic interference, design complexity and yield. While we leave the argumentation aside, the result will be less chips but with more reprogrammability and a full system being used as a component. As the bottleneck is often at the I/O pins, we can expect bus interfaces to be replaced with high speed serial wires (using LVDS like signaling). On the inside, we will need to find a reprogrammable communication subsystem as discussed above, multiple small processing units that plug into it and likely also a number of reprogrammable gates to adapt the chip to the various applications. While one can argue that such chips contain a lot of extra logic that is not always used, the resulting chip will still be often cheaper to use than a set of dedicated chips, with software becoming the differentiating factor. Also at the software level, a CSP like development approach becomes a must to master the complexity and most importantly to be able to deliver the final application in a relatively short period of time.

Early examples.

- Board level examples
  Early examples are of course the INMOS transputer that had internal memory, a prioritizing process scheduler, channel communication and links with DMA. In the early 1990's a second-generation transputer (T9000) was designed that had a router build in. However, the design was clearly too ambitious for the design capabilities at that time. Nevertheless, the T9000 link that featured clock recovery from the data and introduced the concept of headers, was certified as IEEE1355 as a standard and subsequently (adding LVDS signaling) adopted for use in space as SpaceWire. DSPs like Texas Instruments' C40, Analog Devices SHARC also added links with DMA to the core CPU. While they deliver a higher performance (at a higher frequency) than the transputer, the links are complex to put at work and providing multi-tasking is error-prone. In 2002 however, the link communication concept got a major boost under the name of "switch fabrics". Departing from buses as communication mechanism, switch fabrics use meshes of point-to-point communication links. At this moment, many proposals still compete for adoption in the market of which one is IEEE1355. Other contenders are Infiniband, RapidIO, Stargen and others. Noteworthy is that while already introduced in PICMG 2.x (the CompactPCI set of standards), a special standard was created as AdvancedTCA in PICMG 3.x where the target market is telecommunication infrastructure equipment. Backplane buses have been completely eliminated. It should also be noted that most of the proposed switch fabrics not only provide a link interface but also a linkswitch to provide the capability to reconfigure the network. Another recent example is the CSPA architecture of Eonic Solutions. In this architecture a FPGA is the seat of an active communication backbone with processors being attached to it as co-processors. This allows to reconfigure the communication network (based on switch fabrics) and to insert processing in the data-stream. See section 5 for more details.

- Chip level examples
  Examples of announced chips are still rare but significant. One example is Motorola's e500, essentially a PowerPC where all peripherals are accessed through an on-chip RapidIO switch. The resulting RapidIO design however looks quite complex as it combines as well link communication and remote memory addressing. The latter feature is in the author's opinion an unnecessary complication to accommodate legacy designs.

  More important examples however are found with the Virtex-II-Pro and Stratix FPGA chips from resp. Xilinx and Altera. Both feature RISC macros (ARM, PowerPC), soft RISC cores, memory, and high-level compute blocks in a highly reconfigurable fabric. Links (up to 32/chips) are provided using clock recovery from the data and LVDS signaling at up to 3.4 Gbit/sec. While FPGAs provide an extreme example of reprogrammability at the chip level and

are not yet a cost-efficient solution for high volume or low power applications, they provide an excellent development platform and are clearly moving into the direction of in the 3.3.1 defined next MP-SoC.

4. Why multi-tasking as a design paradigm ?

4.1 Multi-tasking as a high level abstraction mechanism

From the discussion above, another view emerges. While multi-tasking for embedded applications historically originated as a set of services that came with a RTOS, multi-tasking with clean semantics can be viewed as "process oriented programming". This is somehow similar to object oriented programming, but less abstract and more closer to the reality of embedded systems. In process oriented programming a task acts like a unit of execution with encapsulated behavior and provides for modular programming. The main differences with object-oriented programming are that the inside of a process is never visible at the outside and that all interaction must be based on pure message passing. If one combines this with the use of a common high level programming language both for software and hardware (e.g.ANSI C and Handel-C), one gets programs that can be used as a common reference model both for software and hardware and with the capability to compile tasks for both back-end environments. In other words, multi-tasking is the basis for a hardware-software co-design environment. Note that while System-C was originally conceived as a cycle-true simulation environment, it is evolving in the same direction. A consequence however is that multi-tasking also implies well-defined semantics for the intra-task communication mechanisms. While it remains possible to "compile" these interfaces for best performance in an application specific way (e.g. CoWare), better re-use also implies that these are standardized. It might not seem obvious how this can be done for both software and hardware. Fortunately, industrial practice has stabilized on a number of common interfaces.

4.2 RTOS services as a system level orthogonal instruction set

Just like most processors have a similar set of instructions (at least at the semantic level), most RTOS have a similar set of services. The same applies for "hardware interconnects. Let's put the most used ones in a table :

RTOS service	Equivalent in hardware	Description
UNIT OF EXECUTION	CHIP, LOGIC BLOCK, MACRO	Buildings blocks
Task or process (a function with its own workspace)	State transition machine	Essentially a black box with well defined outputs as a function of the inputs. Internally a sequential or clocked piece of logic.
SYNCHRONISATION		
Binary event	Status bit	Used to identify a well-defined state. Has no memory.
Counting semaphore, resources	Status bit + counter	Event with a counter to remember how often the state was reached (and yet to be acted upon)
COMMUNICATION		
FIFO queue	FIFO memory	First in, first out memory buffer with full and empty status. Allows multiple readers and writers.
Mailbox + message, piped channels	Shared memory + DMA + status registers	Transfers data of any size with change of ownership. Allows multiple readers and writers.
Memory maps and pools	Memory Management Units	Protects data areas from being corrupted.

Above services are not present as such with all RTOS and when present there might be significant differences in the semantics and interface to call the services. However, if a common design specification is to be used, both for hardware and software, with as target a heterogeneous system with multiple cores - and let's assume that this will the typical MP-SoC target - one must not only standardize the high level language, but also the interfaces. This allows to keep the same 'source" code with different back-end compilers depending on the target. And just like in programming, designers can then still optimize at a local level (but at that moment loosing the link with the high level reference model). In the figure below, one can see how the same application can be mapped onto two different targets. The first one is a software implementation on 3 processes, the second one is an implementation on a mixed hardware-software target with just one processor.

Qualcomm Exhibit 1006

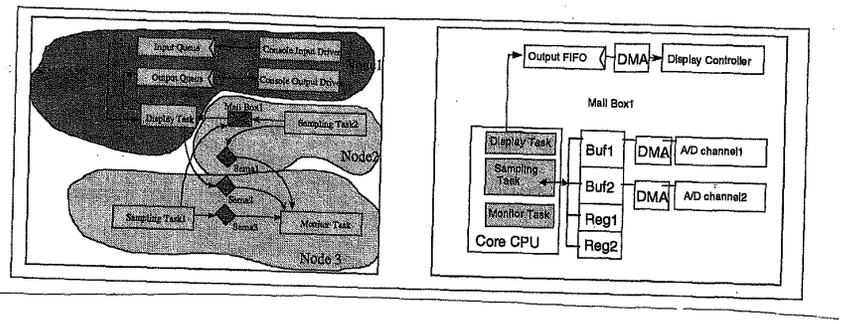


Figure 11.1. Same multi-tasking program mapped on a 3-node processor target and an ASIC with CPU core.

5. Lessons from a fully distributed RTOS and CSPA based ATLAS DSP developments

5.1 Virtual Single processor semantics in an RTOS

The Virtuoso RTOS [6] was originally developed by Eonic Systems in the early 1990's as a distributed RTOS for the transputer and later on ported to parallel DSPs like C40 and SHARC. The initial motivation was to have a RTOS with low interrupt latency and the capability to run fully distributed on parallel DSP targets with little memory. While the RTOS services in Virtuoso look very similar to those of other RTOS services, they have a distinguishing "distributed semantics". This means that any task can call almost any service independently of topology and object mapping on the network of processors. This was achieved by defining well-behaved semantics (no side-effects), a distributed naming scheme to address the different objects managed by the kernel and by introducing a system level communication system. The objects managed by the kernel (which is identical on each processing node) are tasks, events, resources, semaphores, FIFO queues, mailboxes, pipe channels, memory maps and pools. The key layer of the Virtuoso RTOS is the communication layer for which a separate system level is used with very low latency and context switching times. This layer has a build-in router that works on the basis of fixed size "command" packets that carry out the remote service calls and packetizes the data in user defined "data" packets. This packetisation is needed for several reasons. It allows to minimize the buffering needs at the communication layer, it allows to control the communication latency and it allows to prioritize the data

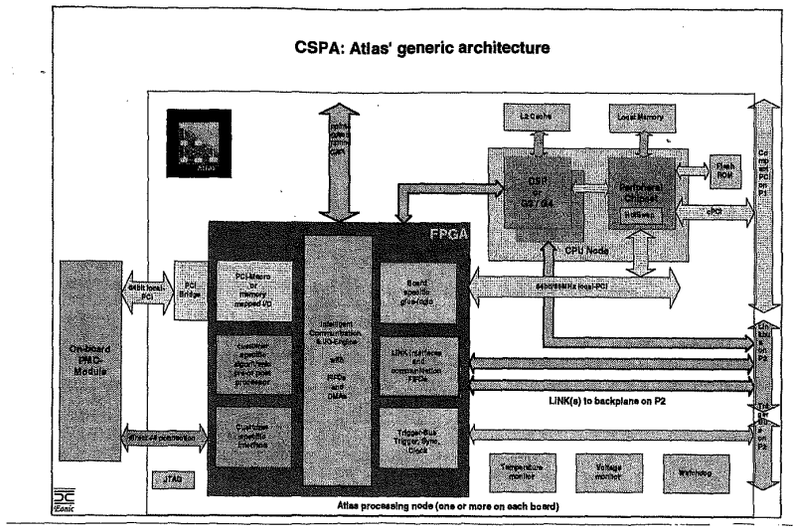
transfer at the system level to preserve the real-time properties. While this system was originally developed for high-end DSPs, typically each with 2 bi-directional DMA driven links, Virtuoso has been ported to other environments where the processing nodes as well as the communication links were fully heterogeneous. See [7] for more details on Virtuoso or visit www.eonic.com.

Another RTOS that allows distributed operation is OSE. OSE was specifically designed with signal processing for telecommunications in mind and features processes and "signals" (a kind of channels). Besides the focus on telecommunications, OSE differences from Virtuoso that it has more support for dynamic features. E.g. communication is done with "linkhandlers" with the capability to time-out if a connection is broken. This allows the application to reconfigure the routing the data-communication without the need to reboot the system. See www.OSE.com

5.2 The Communicating Signal Processing Architecture (CSPA) of the Atlas DSP Computer

The original design goals of the Virtuoso RTOS were to provide an isolation layer for the real-time embedded developer between the application and the increasingly complex DSP hardware. As high-end systems often use multiple DSPs (up to several 100 to 1000's), Virtuoso VSP not only isolates the application program from the target processor but also from the underlying network topology. A major benefit of the approach is total scalability without the need to change the application source code of a program. A second benefit is that the combination of the distributed semantics and multi-tasking result in very modular programs allowing to remap the tasks as modules on different target processors.

If the hardware architecture supports this model, this results in a very efficient but a very flexible system design methodology. In order to guarantee true scalability, this means that the hardware must at least have a communication to computation ratio > 1 and that no bottlenecks in bandwidth or latency may be present. In typical dataflow dominated DSP applications however, the input and output streams have a high bandwidth. Often, the performance limitations are not so much due to a lack of processing power but due to communication bottlenecks. The latter applies in particular to memory I/O and shared buses. In the CSPA concept applied in Eonic's Atlas DSP computer this is addressed by implementing an active communication backbone to which all I/O, processing nodes and memory are connected as a kind of "co-processors".



This communication backbone is implemented using high-end FPGAs. The result is that the on-board processing nodes (e.g. SHARC, C62 and G4) are very much relieved from the data-communication and interrupt processing. This approach also allows to reconfigure the communication network depending on the application requirements and to insert processing steps in the data streams. While the latter introduce some delay (but measured in clock cycles), the pipelining in the FPGA keeps the bandwidth intact. This architecture also allows providing "switch fabrics" (LINKs in the Atlas terminology) even for processors that have little support for communication. Contrary to shared buses, this allows scalability and a much higher degree of flexibility even when using processors with no communication links (e.g. PowerPC).

5.3 Taking the next step: CSP based hardware design

The experience gained with the CSPA architecture confirms the benefits of designing multi-processing systems around a flexible and scalable communication backbone. It also points into directions that can increase the efficiency of processor cores and how these can be assembled into MP-SoC. While the lessons are less relevant for MP-SoC designs with a limited number of processing cores (e.g. typically < 4), they certainly apply for MP-SoC with a larger number of processing cores. The major difference (from a technology point of view) between MP-SoC and rack) level parallel processing is that MP-SoC is less hampered by the problems encountered when going off-chip, e.g. issues of passing high speed signals with multiple wires through connectors are not encountered.

Processor cores as co-processors for a communication backbone. In the paper it was shown that processing and communication are equally important at the system level, but that we don't find this always back in how systems are designed. Often, systems are still designed and programmed from the view that the processor is central. We could turn things around and not only make the communication subsystem the central system, but also make it reprogrammable. The result is that a flexible MP-SoC system could be designed as follows:

- a number of synchronous processing blocks, each running at their own frequency. This preserves the legacy of tools and IP blocks that exist. As each block can run at its own frequency, it also reduces power consumption.
- an asynchronous but reprogrammable communication subsystem. This is likely to be based on some form of FPGA logic and should include the I/O part as well as reprogrammable logic to embed processing in the data-stream.
- high-speed serial links using a higher level protocol and LVDS or similar signaling.

In this context, we need to define what the processing blocks could be. These can be general purpose (e.g. existing) cores but also function specific cores. The difference with current practice is that the interface should be defined at a higher level (e.g. links with FIFO buffer and DMA). Processing blocks designed for high throughput like heavily pipelined CPUs can also be simplified if all interrupt processing is removed and put into specific I/O processors. This allows more efficient designs and higher frequencies.

On links and switch fabrics. Link technologies were essentially put forward to solve the problems with communicating off-chip at high speed and over longer distances. As this often also requires error detection and recovery at runtime, as well as buffering for e.g. through-routing, it is natural to come to communication solutions based on packet switching, each packet being composed of a header and a payload. A good overview of some of the issues and what can be achieved can be found in [2]. Note however that such networks need to be matched by the programming environment to exploit the benefits. As mentioned above, a process and channels communication model is the most appropriate.

One could ask if this is a good solution for MP-SoC as well. MP-SoC implementations have the advantage that packet switching can be used without the need to go bit-serial and using LVDS type signaling. The conclusion from using packet switching at the board level is that while packet switching is very generic and flexible, it works well for average performance. E.g. packet switching suffers from set-up overhead that increases when using smaller packets. As an alternative, we could envision using circuit switching, e.g. a bus, but as seen this seriously hampers the scalability. The solution is to have a fully reconfigurable communication backbone that can be used for both by reprogramming and to put some of the system software intelligence in this hardware layer.

6. Conclusions

Communication and I/O are often seen as peripheral activities for the processing elements. This can be historically explained as the original von Neumann concept has been dominating processor design for decades. With the advent of embedded systems that can contain several tens of processing elements, it is clear that a system level approach is needed. We have argued that in such a system all elements must be designed as programmable building blocks. This is derived from the original CSP model that equally applies to software and hardware.

References

[1] C.A.R Hoare. *Communicating Sequentioal Processes.* Prentice-Hall, 1985.

[2] J.P. Lehoczky, L. Sha, J.K. Strosnider, and Hide Tokuda. *Scheduling and resource management*, section 1.2. Foundations of real-time computing. Kluwer Academic Press, 1991.

[3] INMOS Ltd. *Occam 2 Reference manual.* Prentice Hall, 1988.

[4] INMOS Ltd. The transputer databook. Technical report, 1989.

[5] INMOS Ltd. *Transputer instruction set.* Prentice Hall, 1998.

[6] E. Verhulst. Virtuoso : providing sub-microsecond context switching on dsps with a dedicated nanokernel. In *International Conference on Signal Processing Applications and Technology*, Santa Clara September, 1993.

[7] E. Verhulst. The rationale for distributed semantics as a topology independent embedded systems design methodology and its implementation in the virtuoso rtos. *Design Automation for Embedded Systems*, 6(3):277–294, March 2002.

Chapter 12

NOC APPLICATION PROGRAMMING INTERFACES

High level communication primitives and operating system services for power management

Zhonghai Lu
Royal Institute of Technology, Sweden
zhonghai@imit.kth.se

Raimo Haukilahti
Royal Institute of Technology and Mälardalen University, Sweden
rhi@imit.kth.se

Abstract Due to its heterogeneous and distributed nature, programming NoC communications may be very complicated if we treat NoC as individual elements of resources, switches, and interfaces. To mitigate the complexity, we raise the abstraction level and take NoC as a whole. To this end we propose a concept of NoC Assembler Language (NoC-AL) which serves as an interface between NoC implementations and applications, very similar to the instruction set of a traditional CPU. A central part of NoC-AL will be communication primitives for both message passing and shared memory. Starting with a NoC programmer model, this chapter discusses NoC-AL, and in particular the communication primitives. Moreover, we discuss NoC Operating System (NoC-OS) which is the underlying layer below NoC-AL. As power consumption has become one of the primary design constraints, we give an overview of low power techniques at the operating system level and demonstrate how process migration can improve the effectiveness of the techniques. At the end, we propose an API for power management of NoCs.

Keywords: Application Programming Interface, NoC Assembler Language, Communication primitive, Operating system, Power management API

A. Jantsch and H. Tenhunen (eds.), Networks on Chip, 239–260.
© 2003 *Kluwer Academic Publishers. Printed in the Netherlands.*

1. The Programmer Model

NoC is inherently a heterogeneous distributed system. Heterogeneity implies that different elements, like resources, switches and interfaces, are designed in various means. A number of languages, synthesis tools, software compilers and linkers are required for the design of individual elements. There is no single design flow which can be applied to the design of all these elements. Distribution implies that processes on different resources interact with each other via the on-chip communication network. NoC design will be communication-centric. In addition to the NoC architecture, we also have to address the design of process communications. If we treat NoC as individual elements, the design of NoC communications may be very complicated. Communication standards and protocols are desired to coordinate programming NoC communications. The relatively independent design of processes makes it hard to integrate all these elements. For example, how to program a process running on an ARM microprocessor to communicate with a process running on an ASIC through the communication network? To hide the complexity, we propose a NoC programmer model where we offer interfaces to NoC designers.

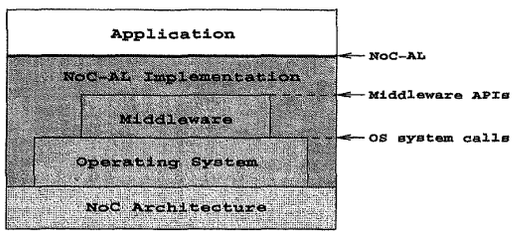


Figure 12.1. The NoC programmer model

Figure 12.1 shows the NoC programmer model where there are three layers of Application Programming Interfaces (APIs): NoC Assembler Language (NoC-AL), middleware APIs and OS system calls. An API provides an interface for its upper layer. However, its implementation details are hidden from its upper layer, thus opening up the possibility to have a flexible implementation while keeping the interface static.

NoC-AL provides an interface between applications and implementations. Using NoC-AL, NoC application designers are able to program NoC architecture and application. In application, we separate communication from computation. The difference between computation and communication lies in that the former uses only processing elements, while the latter uses both processing elements and communication media [21]. A conceptual illustration of a NoC-AL program is shown below,

where the NoC architecture is a $m \times n$ mesh of switches and each resource is connected to one switch via a Resource-Network-Interface (RNI) [11].

```
NoC Architecture Description
 {Topology: mesh 2 x 2
  Resource List:Row1: R1=SHARC DSP, R2=ARM CPU
                Row2: R3=FPGA, R4=ASIC}
NoC Application
  {R1:{computation_file1.c; communication_file1.c}
   R2:{computation_file2.cpp; communication_file2.cpp}
   R3:{computation_file3.vhdl; communication_file3.vhdl}
   R4:{computation_file4.verilog; communication_file4.verilog}}
```

The middleware APIs are reusable library functions. They offer standard services, and are viewed by NoC-AL implementations as APIs. NoC operating system offers system calls to NoC-AL implementations and middlewares. System calls are those calls most directly acting with the underlying hardware. They provide simple and clean methods to use hardware resources.

The implementation of NoC-AL uses both the middleware APIs and the OS system calls. We also notice that it may work on the underlying architecture directly. This is due to the fact that some types of resources consist of pure hardware, like FPGAs or ASICs, while others consist of microprocessor(s) without operating system support.

The remainder of this chapter is organized as follows. We will discuss the NoC Assembler Language in section 2, focusing on NoC communications, and NoC Operating System in section 3, focusing on low power techniques and power management APIs. Finally, we summarize this chapter in section 4.

2. NoC Assembler Language

NoC Assembler Language [10] is defined as follows:

> *NoC Assembler Language (NoC-AL) serves as an interface between NoC implementations and applications, very similar to the instruction set of a traditional CPU. A central part of NoC-AL will be communication primitives such as send and receive, open and close, and a standardized way of using shared memory. Every instance of a NoC must come with a NoC assembler, which translates NoC-AL programs into a set of NoC configuration files.*

From the definition, we see that NoC-AL treats NoC as a whole instead of individual elements such as resources, switches, and interfaces etc. The design languages for these elements, e.g. *VHDL/Verilog* for hardware design, *C/C++* for software design, *SystemC* [9] and *SpecC* [8] for both hardware and software design, are coherent parts of NoC-AL. In fact, a NoC is a network architecture which integrates resources. Ideally we expect that resources can simply *plug-and-play* on the NoC archi-

tecture. To this end, NoC-AL should also offer methods to describe NoC architecture and process communications besides computational tasks. The NoC architecture concerns NoC topology, resource list and process-to-resource mappings. The methods used for describing process communications are *communication primitives* which we will define in this section. There are a lot of open NoC-related issues, such as what NoC architectures/topologies are good for which applications, how to efficiently map processes to resources at run time by task migration, and so on. Most of the topics are beyond the scope of this chapter. In this section we concentrate on NoC communications, in particular, communication primitives for both *message passing* and *shared memory*.

One question arises when defining NoC message passing primitives: why not simply adopt Message Passing Interface (MPI) [16]? MPI has become the de facto standard for distributed programming that defines a message passing API library. It comprises 129 functions offering extensive functionality, flexibility, and generality. The implementation of MPI demands the support of powerful operating systems which are very often not the cases for embedded operating systems that are real-time oriented and compact. This may lead to implement MPI on NoC difficult, and less efficient for a specific application. On the other hand, although NoCs use network communication to overcome the scalability problem of bus-based System-on-Chip (SoC), some communication features, such as transmission latency, bandwidth, and traffic type etc. need to be reserved. However, these features are not available in MPI, but important issues for chip design in order to achieve efficient implementations in terms of speed, area as well as power.

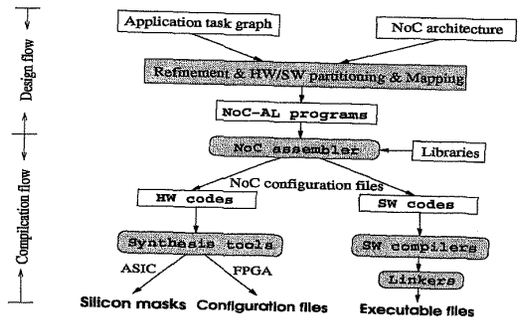


Figure 12.2. The NoC application design and compilation flow

NoC Assembler. A NoC-AL program includes NoC architecture description and application description. IP blocks, such as DSPs, CPUs,

and memories etc. are listed in the resource list of the architecture description. Since they are reusable cores, there is no need to synthesize or compile them. Instead we expect them to be pre-fabricated together with the NoC architecture. Thus, a NoC itself is a half-customized prototype. A NoC application can be expressed as a *task graph*, and then mapped onto the given NoC architecture after iterations of refinement and hardware/software partitioning until satisfaction. The design tasks are reduced to custom hardware like ASIC and FPGA, software, and communication interfaces in hardware and software. To translate NoC-AL programs into NoC configuration files including both hardware and software parts, we need a NoC assembler which does source-to-source processing, not generating low abstraction level codes, before standard tools for hardware synthesis and software compilation & linking are used. This procedure is illustrated in figure 12.2, where the libraries are implementations of NoC primitives, for example, communication primitives.

2.1 NoC Communications

Based on architecture and IP reuse, a NoC is a communication platform on which applications run. An application is composed of concurrent communicating processes which can be represented as a *task graph* in which a node denotes a process, and an arc a communication link.

Communication Styles. Memory organization plays a decisive role in InterProcess Communication (IPC) styles. The memories in NoC can be either *shared* or *private*. If they are shared, the memories are organized as a single global address space. Processes communicate via *shared variables*. Processes are synchronized by mutual exclusion and/or condition variables. Collective processes may use barrier for synchronization. Shared memory can be designed as shared centralized or distributed memory. If the memories are private, that means the NoC has multiple address spaces. Processes communicate by *message passing*, i.e. explicitly send and receive messages. In general message passing can be synchronous or asynchronous depending on what synchronization schemes are employed by send and receive. Basically a send and a receive can be *blocking* or *nonblocking*. A nonblocking operation allows the process to continue execution whereas a blocking operation suspends the process until receiving acknowledgment or timeout.

Channel Communication. A channel is an arc in the task graph connecting a communicating pair. It is an abstraction of communication media, either dedicated or shared, either physically or logically. At the task level, it does not incorporate implementation details, such as

interfaces, but a set of characteristics regarding performance, cost and Quality-of-Service (QoS) which is required by an application in order to satisfy design goals under given constraints. In the following, we use *C* syntax and *C* conventions to facilitate defining communication primitives when necessary. However, they are language-independent, and can be bound to various hardware and software design languages.

We identify and define some important channel characteristics in a *struct* as follows:

struct **channel_feature** {*direction, burstiness, connection, latency, bandwidth, priority, reliability*}

The *direction* gives the orientation of message transfer. It may be simplex, half-duplex or duplex. The *burstiness* reflects channel traffic characteristics, which can be periodic or aperiodic. It is useful for modeling chip network traffic, and reducing power consumption. The *connection* tells if a channel is hard-wired like circuit switching, or, in the case of packet switching, connection-oriented virtual channel or connectionless. The *latency* is the time for a single unit of message transmitted from source to destination. It is measured in either absolute time or relative time in terms of the number of clock cycles. It may have three values: minimum, average and maximum. The *bandwidth* is the channel capacity of transferring data. It is measured by the number of the basic unit transmitted by the physical layer, e.g. cells per second. It may take three values: minimum, average and maximum. The *priority* is a nonnegative integer representing the priority of a channel. It is useful for scheduling when multiple channels are contending for a shared resource, like buffer. *Reliability* is an othorgonal feature. Any channel can be designed with a certain reliability no matter the underlying layer is reliable or not. We can define several levels of reliability.

In ISO's OSI seven-layer reference model, a channel deals with the *session layer* offering network IPC services. Implementing a channel with various features requires the support of communication protocols at the lower layers. Berkeley *Sockets* interface [19] was designed to provide generic access to IPC services implemented by whatever protocols on a particular platform. In this sense, the *sockets* API is a good reference for implementing NoC channels. *SystemC* channels support hierarchical communication and communication refinement [9], thus it also provides a good reference for NoC channel implementations.

2.2 Message Passing Primitives

A process is uniquely identified by a tuple (*resource_number, process_number*). A channel is shared by a source process and a destination process, thus identified by a (*source_process, destination_process*) pair. A message passing procedure is a channel-based data transaction that consists of three phases: *channel setup, data transmission*, and *tear down*, as illustrated in Figure 12.3. Channel is set up by *request and response*. This handshaking procedure may not actually take place if the initiator asks for a connectionless channel. In this case, opening a channel just assigns the destination address to the initiator, and doesn't expect response. Data transmission may be one-way or two-way. The initiator doesn't necessarily send message first because a channel request may be initiated by the receiver who wants specific channel features. Finally a channel can be torn down by either end of the communicating processes.

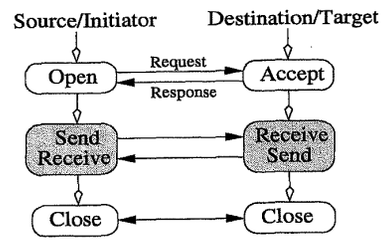


Figure 12.3. Message passing procedure between processes

In terms of the message passing procedure, we define the following communication primitives for message passing:

- Open a channel

  int **channel**(*source_process, destination_process, channel_feature*)

  This function initiated by the *source_process* opens a channel between the *source_process* and the *destination_process*. The *channel_feature* is defined as a *struct*, reflecting channel characteristics as specified in the previous subsection. It returns a *channel* descriptor, which is a nonnegative integer, if successful, or negative integers for different reasons of failure, such as network bandwidth requirement not satisfied, or destination not available, and so on.

- Listen to channel

  int **listen**(*maxQueueLimit*)

  This function sets the maximum size of channel request queue, and causes internal state changes to permit channel requests.

- Accept a channel

  int **accept**(*channel*)

Qualcomm Exhibit 1006

This function reads one channel request from incoming buffer, stores into *channel*, and responses the channel initiator if necessary. It returns 1 on success, or negative integers for various reasons of failure, such as channel request not available, parity check/checksum error, requested channel features not met etc.

- Bind a channel

  int **bind**(*expected_channel, channel*)

  This function checks if an accepted *channel* matches an *expected channel*. It returns 1 for successful matching, -1 for failure.

- Send message

  int **send**(*channel, msg, msg_size, msg_type, msg_id, sync_flag, timer, out-of-band, request*)

  This function sends message to the specified *channel*. It returns 1 for success, negative integers for various reasons of failure, and 0 if timeout occurs when blocking send is used. The *msg* is the initial address of the message to be sent. The *msg_size* is its size. The *msg_type* is its datatype. Since data representations vary in diverse microprocessors and design languages, an explicit data type is required to transmit for correct data type mapping. The *msg_id* is its identity number that enables the sender to do retransmission, and the receiver to maintain correct message sequence and avoid message duplication. The *sync_flag* with value 1 or 0 specifies whether the send function is blocking or nonblocking. The *timer* is used for two purposes. If the send is blocking, it specifies the maximum amount of time the sender waits for acknowledgment. When timeout occurs, the function returns 0. Whether to retransmit is up to the application. If the send is nonblocking, the timer with a positive value specifies the minimum amount of waiting time before retransmission, and if the timer equals to -1, no acknowledgment from the receiver is required, thus no retransmission. The *out-of-band* is a flag with value 1 or 0 to distinguish *out-of-band* data from *in-band* data. It is useful for conveying control information. The *request* is an optional object used later to query the status of the nonblocking communication or wait for its completion.

- Receive message

  int **receive**(*channel, msg, msg_size, msg_type, msg_id, sync_flag, timer, request*)

  This function receives message from the specified *channel*. It returns 1 for success, negative integers for various reasons of failure,

and 0 if timeout occurs when blocking receive is used. The *msg* is the initial address of the buffer into which to put the incoming message. The *msg_size*, *msg_type* and *msg_id* are its size, datatype, and identity number, respectively. The *sync_flag* specifies whether the receive function is blocking or nonblocking. The *timer* is used for two purposes. If the receive is blocking, it specifies the maximum amount of time the receiver waits for message available. When timeout occurs, the function returns 0. Whether to continue polling the channel is up to the application. If the receive is nonblocking, it specifies the minimum amount of waiting time before re-polling the channel. The *request* is similar to that explained in the **send** function.

- Check nonblocking completion

  int **check**(*request, status*)

  This function checks if the operation identified by *request* completes. It returns information on the operation in *status*.

- Close a channel

  int **close**(*channel*)

  This function closes the specified *channel*. It returns either 1 for success, or -1 for failure.

In addition to these basic primitives described above, we need some other primitives for channel management such as **getchannelopt**() and **setchannelopt**(), data conversions, multicast, and so on.

2.3 Shared Memory Primitives

Shared memories can be used statically such as defining global shared variables, or dynamically. Here we are concerned with dynamic use of shared memory. The way of using shared memories should be standardized. It consists of three phases, namely, *allocation*, *access* and *release*, as shown in Figure 12.4. Memory can be written and read in one-word-based or multiple-word-based.

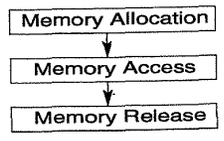


Figure 12.4. The procedure of using shared memory

Qualcomm Exhibit 1006

According to this procedure, we define shared memory primitives as follows:

- Memory allocation

  int **memory**(*resource, start_address, end_address, memory_type*)

  int **memory**(*resource, number, memory_type*)

  The two functions request a memory segment from the memory *resource*. The first one specifies *start_address* and *end_address*. If successful, it returns a *memory* descriptor, which is a nonnegative integer. The second one gives the requested *number* of words space. Upon success, it returns the start address of the allocated memory segment. On failure, both return different negative integers for various reasons of allocation failure, such as memory resource not available, memory full etc. The *resource* is the location of the memory. The *memory_type* is defined as a *struct* concerning if the memory allows multiple concurrent reads:

  struct **memory_type**{read: multiple | single; write: single;}

- Memory access – Read one or multiple words

  int **read**(*memory, number, start_address, dataarray, flag*)

  This function reads *number* data starting from the *start_address* in *memory*, and assigns to addresses pointed by *dataarray*. The *flag* denotes the atomicity of this read operation. It takes either 1 for interruptible, 0 for non-interruptible. If the *number* equals to 1, this function reads only one word.

- Memory access – Write one or multiple words

  int **write**(*memory, number, start_address, dataarray, flag*)

  This function writes multiple words pointed by *dataarray* into the specified address space starting from the *start_address* in *memory*. The *flag* denotes the atomicity of this write operation. It takes either 1 for interruptible, 0 for non-interruptible. If the *number* equals to 1, this function writes only one word.

- Memory release

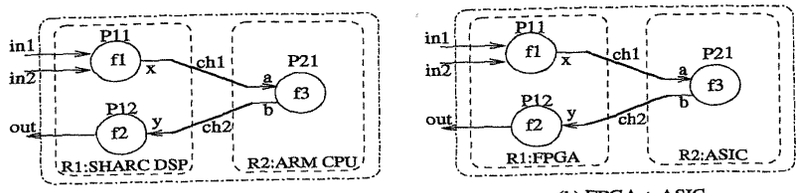
  int **free**(*memory*)

  This function releases the allocated *memory* space. It returns 1 for success or -1 for failure.

Functions **read** and **write** return a positive integer denoting the number of successfully accessed words upon success, and negative integers for various reasons of failure, such as read/write contention, memory address error etc.

2.4 An Example of NoC-AL Program

Suppose there is a simple application illustrated by the task graph in Figure 12.5.(a). Assume we have a NoC which only consists of two resources, a SHARC DSP and an ARM microprocessor.



(a) SHARC DSP + ARM CPU (b) FPGA + ASIC

Figure 12.5. An example of NoC application in task graph

We manually map processes P11 and P12 to R1, the SHARC DSP, process P21 to R2, the ARM CPU. Using the proposed primitives, a NoC-AL program might be coded as follows:

```
NoC Architecture {
  Topology: mesh 1 x 2
  Resource List: Row1: R1=SHARC DSP, R2=ARM CPU}
NoC Application {
R1:{#include <NoC-AL-SHARC.h>
    #include <f1.h>
    #include <f2.h>
    double in1,in2,out,x,y; int ch1=0, ch2=0, ach2=0;
    Process P11 {
     while (1) {
      x=f1(in1,in2);
      while (ch1<=0) {ch1=channel(P11,P21);} //Open a channel ch1 until success
      while (send(ch1,x)!=1) {continue;}}    //Wait for send to ch1 success
     while (close(ch1)!=1) {continue;}}      //Close channel ch1
    Process P12 {
     while (1) {
      while (ach2<=0) {ach2=accept(&ch2);}   //Wait for channel ch2 accepted
      while (receive(ch2,y)!=1) {continue;}  //Wait for receive from ch2 success
      out=f2(y);}}}
R2:{#include <NoC-AL-ARM.h>
    #include <f3.h>
    double a,b; int ch1=0, ch2=0, ach1=0;
    Process P21 {
     while (1){
      while (ach1<=0) {ach1=accept(&ch1);}   //Wait for channel ch1 accepted
      while (receive(ch1,a)!=1) {continue;}  //Wait for receive from ch1 success
      b=f3(a);
      while (ch2<=0) {ch2=open(P21,P12);}    //Open a channel ch2 until success
      while (send(ch2,b)!=1) {continue;}}    //Wait for send to ch2 success
     while (close(ch2)!=1) {continue;}}}}    //Close channel ch2
```

Here the **open**, **accept**, **send**, **receive** and **close** primitives are simplified without additional arguments. These primitives are implemented

in software libraries "NoC-AL-SHARC.h" for SHARC DSP, and "NoC-AL-ARM.h" for ARM CPU. Both libraries are used as *include* files.

We should note that the task graph can be also mapped onto hardware resources, or both hardware and software execution resources. If we map the processes to hardware resources, say, FPGA and ASIC, the primitives are used similarly, but implemented in VHDL/Verilog/SystemC libraries depending on which language we are utilizing to describe hardware processes. Figure 12.5.(b) reflects one possibility of the mappings. Accordingly its architecture and application description will be modified. Due to space limitation, we only show its architecture description as follows:

```
NoC Architecture {
  Topology: mesh 1 x 2
  Resource List: Row1: R1=FPGA, R2=ASIC}
```

3. Operating Systems for NoCs

Since the 1950's a number of various OSs have been developed and tried in practice. The types of services offered by OSs have been varying during these years. Commonly an OS offers services such as process management, memory management, file management, and communication. There are two views of the OS as depicted in figure 12.6. In the top-down view, an OS is a piece of software that abstracts hardware details from the programmer by presenting an *abstract virtual machine*. This abstract machine offers services to the upper layers (application or middle-ware programmers), via *system calls*. From the hardware point of view (bottom-up) the OS acts as a *resource manager* multiplexing hardware resources to application processes.

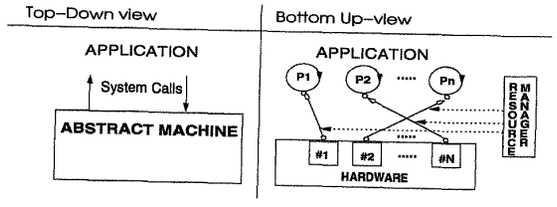


Figure 12.6. Two views of an OS

3.1 NoC-OS Options

A complex NoC architecture will definitely need an OS. Once a processing element is expected to have more than one process executing, process management support is needed. Moreover, implementation of NoC-AL and software parts of communication protocols will be facilitated if an OS offers sufficient services. NoC architectures will likely consist of several heterogeneous processing elements. How should we organize the OS for such a system? For a NoC-OS we have several choices:

- A centralized OS. The NoC resources run under only one OS and the programmer sees the system as one resource, not several.
- A local OS on every node. Each processing element (PE) runs its own OS instead of running part of a global system wide OS.
- A mixture of the two above. Resources can be grouped together to run under the control of a centralized OS while other resources still run local OSs.

The design of a NoC-OS is complicated due to the distributed heterogeneous elements. However, implementation aspects and details of these alternatives are out of scope for this chapter as we focus on low-power techniques and power management APIs.

3.2 Low-power Techniques for Operating Systems

The OS coupling with power consumption has increased during the last years and we believe this trend will continue or even accelerate. At the OS level a lot of information can be utilized at run-time and an OS has also the ability of controlling the execution of application processes and hardware resources. Here we will describe and propose ideas how the NoC-OS can assist to facilitate the power and energy problems. We classify operating system techniques related to power consumption as shown in figure 12.7. The term *low-power OS* can be interpreted as an OS that consumes little energy itself (right branch in figure), or as a *power-aware* piece of system software (left branch) using information at the OS level to shut down idle units or trading performance against power dynamically at run time.

OS Optimizations. Studies show that the OS can account for a significant fraction of an embedded system's energy consumption [7]. Hence, there is a reason to consider how to rewrite the software to reduce the power consumption. Known software solutions include code transformations [5] and use of new compiler techniques focusing more on power than performance. To use a specialized OS structure that is well adapted to application needs can also give significant energy savings [12]. By implementing OS functions in hardware there may be a possibility to both speed-up the OS and make it more energy efficient.

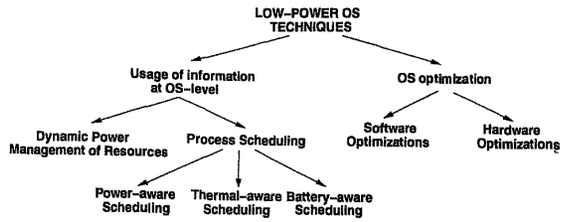


Figure 12.7. Classification of techniques for low-power operating systems

Dynamic Power Management of Resources. When resources are idle, they can be set into idle modes where the clock frequency and/or voltage is reduced to some extent. This technique is called Dynamic Power Management (DPM). The state transition decisions are performed by a Power Manager (PM) which sends request to a component to change its state in order to reduce the power consumption and save energy. A PM uses a *power management policy* for deciding when a state transition should occur. A transition from one state to another has both a time and an energy penalty. Since the OS is a resource manager it is a proper place to integrate a PM in it [13].

Process Scheduling. In this category the algorithms are related to process scheduling in the OS and are therefore implemented as a part of the OS scheduler. While DPM is often considered to only switch between running and sleep states with different characteristics, Dynamic Voltage and Frequency Scaling (DVFS) is a technique with finer granularity where performance is traded against power. When applications do not need full processing power, the clock frequency and the voltage can be reduced to save energy. Scheduling algorithms guaranteeing real-time constraints [17] and multiple PEs have also been presented [20].

Battery-aware Scheduling tries to give a fair use of the available battery energy for processes. Fidelity can also be traded against power. For example, if you want to watch a half-hour movie on a laptop and only have batteries for 20 minutes, a battery-aware OS scheduler can reduce the quality for the user by decreasing the frame-rate, picture size or avoid the execution of processes that are unnecessary at that moment.

Recent systems have started to use a technique called *Dynamic Thermal Management* [4] where the chip temperature is monitored at runtime. When the temperature approaches the safe temperature limit, actions such as putting a processor into a sleep mode or reducing the operating frequency, are taken. Processes with high energy consumption are treated as "hot" processes. When the temperature is increasing

and reaching close to the safe limit, the activity of these processes are limited.

3.3 NoC-OS Process Migration

The trend of integrating more and more functionality on the same silicon die continues while performance is increased every 18-24 month. Thus, the *power density* (power consumption per unit area) is running out of control. We believe new power saving techniques, in addition to the ones previously described, are needed for a NoC and many of these should be implemented at the OS-level.

Process migration is traditionally a technique used for *load balancing* in multiprocessor and distributed systems [15]. The OS can decide to move a process from a heavily loaded processing node (e.g a processor) to another resource in the system where computation is currently lower. Here we will describe how process migration in a NoC-OS can help reducing energy dissipation and keeping silicon die thermal constraints within safe limits.

Process Migration for Maximized Energy Savings. Power-aware scheduling algorithms reduce frequency and voltage when applications do not need full processing power. If the application needs full processing power nearly all the time, the possibility of decreasing the frequency and voltage to save energy is limited. On the other hand if the application has bursty characteristics where the needed processing power varies with time, Dynamic Voltage and Frequency Scaling (DVFS) has great potential. We believe that migration of processes can increase the effectiveness of DVFS algorithms. Consider the system depicted in figure 12.8 with the two processing elements PE1 and PE2, and three processes with their corresponding need of processing power stated below. Assume that the two PEs will run in three modes:

- Full speed. f=800 MHz, V_{DD}=1.3 Volts, P=800mW, 3000 MIPS
- Half speed. f=400 MHz, V_{DD}=0.315 Volts, P=200mW, 1500 MIPS
- Idle. f=0 MHz, V_{DD}=0 Volts, P=0 W, 0 MIPS

The numbers for full and half speed modes are derived from an early version of the X-Scale processor [6]. For simplicity we approximate the performance to be 50 percent of the maximum when using the half speed mode, and a perfect idle state in which no power is consumed. Moreover, we have assumed that a DVFS scheduling algorithm is used to scale down frequency and voltage whenever possible. In this example, PE1 and PE2 would have been idle for 20 percent and 80 percent respectively of the

Qualcomm Exhibit 1006

time without DVFS. If we use DVFS the processors will slow down in order to save power as depicted in figure 12.9 where also the options for process migration are shown.

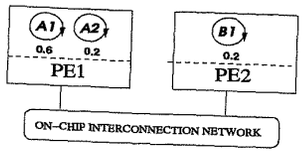


Figure 12.8. A system with two processing nodes and three processes.

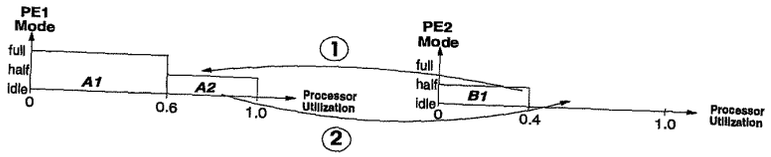


Figure 12.9. Migration alternatives in a 2-processor system

Initially before process migrations the average power consumption for PE1 and PE2 are:

$$P_{PE1} = 0.6 * 800 + 0.4 * 200 = 560mW, P_{PE2} = 0.4 * 200 = 80mW \quad (1)$$

Yielding a total consumption of 640mW.

We have now two interesting options. Either we migrate process B1 to PE1 and put PE2 in the idle state or we may migrate processes A2 from PE1 to PE2. What does these options mean in terms of power and energy consumption? Let's analyze the results of the migrations (figure 12.10):

If we move all the load to PE1 (case 1) we get:

$$P_{PE1} = 1.0 * 800 = 800mW, P_{PE2} = 0mW \quad (2)$$

If we instead balance the load (case 2) we get:

$$P_{PE1} = 0.2 * 800 + 0.8 * 200 = 320mW, P_{PE2} = 0.8 * 200 = 160mW \quad (3)$$

In this example, by balancing the load with process migration, we increased the effectiveness of the DPM algorithm which resulted in a reduction of the average power consumption with 25 percent (640mW to 480mW). Concentration of the computation to one PE results in

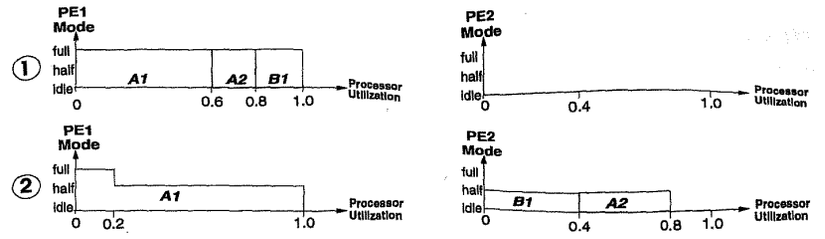


Figure 12.10. Processor modes after process migration

increased power consumption, from 640mW to 800mW. As seen in figure 12.10 case 1 actually corresponds to a single processor architecture while case 2 is a multiprocessor solution. The example shows that a multiprocessor architecture can be a more energy-effective solution than a single processor architecture. If we also include the energy overhead of communication we could get contradictory results. Imagine that process B1 is a producer of the data while processes A1 and A2 are consumers. If B1 is physically far away from the other processes, the data must go through many switches in the NoC and may lead to higher consumption than if all three processes execute on a single node.

Process Migration and Dynamic Thermal Management. Design techniques to minimize the average power consumption are used in order to prolong the battery life. In contrast, when attacking the peak power problems, the goal is to limit the power consumption peaks to prevent damage of the silicon. To ensure that the chip temperature never exceeds a safe maximum, designers run benchmarks to characterize worst-case scenarios. There is no way to guarantee that these unrealistic worst-case scenarios will occur, so often the package must be designed in a conservative way to avoid possible damage. This leads to an expensive product. Borkar [3] has estimated that additional power dissipation above 35-40 Watts the total cost per chip increases by more than 1$/W.

The trend of integrating more and more functionality into one chip together with the dynamic nature of applications makes it interesting to control overheating on a finer grained level. In a NoC it is likely that some parts have a lot of computation and communication while other parts are being more sparely used or even stay in idle states. To avoid large temperature differences and to keep local resource temperature within limits, processes from "hot" resources can be dynamically migrated to low-temperature PEs. Several on-chip temperature sensors

Qualcomm Exhibit 1006

[14] can be used to collect temperature data in order to make intelligent migration decisions. Ideally, the technique could result in a less conservative designed package while still maintaining performance and reliability.

Migration Issues. Although we pointed out the possibilities to improve the effectiveness of DVFS algorithms and balancing the temperature of the chip by using process migration it introduces new questions and challenging problems. In a heterogeneous platform, software processes run on different types of PEs. Hence, in order to be able to migrate a process, the source and target PEs have to be of the same type. Process migration decisions could be taken *statically* (off-line) or *dynamically* (on-line). For a NoC, a *semi-static* approach is interesting. Conditions for migration could be derived from simulations and then used as guidance at run-time to decide when to migrate processes.

3.4 NoC-OS Power Management API

To manage the power consumption of a NoC, the programmer must have a set of services provided by an OS module in the NoC-OS. We focus on interface of these services , the *power management API*, and not on the implementation. APIs for power management already exist but have limitations for NoCs. Current APIs for power management such as Advanced Configuration and Power Management Interface (ACPI) [1] only defines primitives for controlling power states of resources. ACPI neither provides ready-to-use power management policies nor power-aware scheduling.

Because applications have different needs and characteristics, a single scheduling algorithm or power management policy, cannot be optimal solutions. Instead, we propose programmers should be equipped with higher-level primitives for controlling power-aware scheduling and power management policies provided by the OS. The programmer may decide to let resources and NoC regions [11] run different policies with different characteristics that match the application needs.

Power Management Approaches. Most of NoC resources will likely have several power states with different power-performance tradeoffs. The *responsibility* for initiating power state changes can be one of the following alternatives:

- Resource-based management. The resources have an integrated power manager and take decisions based on the local state, i.e the current workload.

- Application-based management. Programmers have full control over the power state changes and have the responsibility to explicitly change power states of the resources.
- OS-based. Instead of letting programmers control the power state changes this responsibility is passed over to the operating system.

If the resources control the state transitions, programmers do not need to think about power management at all. Limitations of this approach is that the transition decisions are based on a limited amount of information (usually based on time-outs). By letting the application control the state changes explicitly, the power management can be improved but it also increases the complexity of the programming task. This can be avoided by letting the OS control the power state changes based on the available information at this level. We will support both application and OS-based power management.

Benini and De Micheli proposed a different classification for power management approaches related to the *information* the decisions are based on, namely *node-centric* and *network-centric* [2]. In the former case, a local power manager bases the state changes on the current local OS information and the workload. In the network-centric approach resources send messages to neighbors to request state changes. These requests are both generated and serviced at the OS-level.

The proposed API will support both network-centric and node centric approach and combinations of them which have been reported to achieve good results [18].

Power Management API Services. Both application-based and OS-based power management need primitives for controlling the power states and power management policies. Thus, an API for this purpose is needed. We divide the interface into two layers, Layer A and Layer B as depicted in figure 12.11. Layer A provides higher level basic power management services. Layer B supports Layer A with primitives for changing power states and collecting statistics. Due to lack of space, Layer B primitives are not described here in detail, but they can also be used to implement application-based power management of resources.

The Layer A primitives have the following functionality:

- **registerResource**-registers a power manageable resource. Only those resources which are registered will be managed by the OS. The purpose is to allow a mixed application-based and OS-based power management which otherwise might cause inconsistency problems. The arguments are the resource name and the characteristics of the resource power modes.

Qualcomm Exhibit 1006

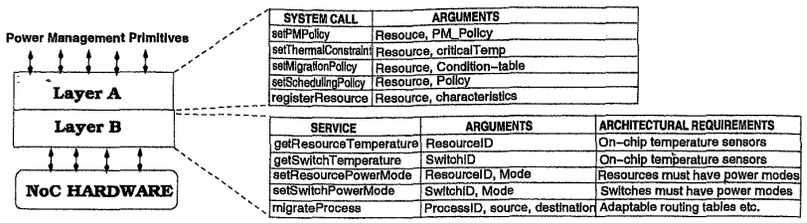


Figure 12.11. NoC-OS power management layering and primitives

- **setPMPolicy**-controls the power management policy. We assume that a set of policies implemented in the OS have been characterized with respect to performance degradation, power/energy savings and real-time performance. The NoC-OS and the resources must provide the application enough processing power while minimizing the energy costs. Therefore power management policies and process scheduling have a strong connection with each other as shown in figure 12.12. *PM_policy* is set to either: 1:high-performance, 2:real-time, or 3:low-power.

- **setSchedulingPolicy**-selects process scheduling algorithm for a resource given by the argument *Policy*. We assume that the NoC-OS will be adaptable and can include any scheduling algorithm the programmer wants to use. The algorithms can have different characteristics such as scheduling for real-time performance, fairness, power-awareness, and battery-awareness.

- **setMigrationPolicy**-sets the process migration policy. The user can provide a *condition-table* to help the OS taking "intelligent" migration decisions.

- **setThermalConstraint**-specifies a maximum temperature. If the user specifies a maximum temperature for a resource (*criticalTemp*), the power manager will guarantee it will never be exceeded.

The relations between the process scheduler, power manager, and NoC backbone are depicted in figure 12.12. The power state messages and requests are either sent to a central power manager agent or to the neighbors.

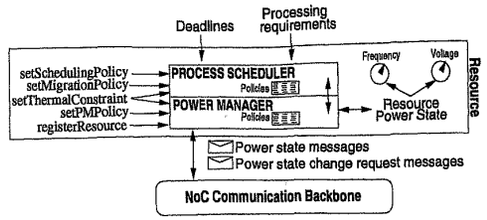


Figure 12.12. NoC-OS power management architecture

4. Summary

Based on the NoC programmer model, we have presented a new concept for NoC design: NoC Assembler Language, which serves as an interface between applications and implementations. To this end a NoC assembler is required to translate NoC-AL programs into NoC configuration files before standard tools for hardware and software design are used. To handle NoC interprocess communications, we use communication channels. Furthermore, we have proposed a set of basic communication primitives for *message passing* and using *shared memory*. NoC operating system is the underlying layer below NoC-AL. We have given an overview of operating systems with emphasis on low power techniques, which we also have classified into *OS-optimization* and *OS-information-based* techniques. In particular, we have shown the potentiality of power management by process migration and also proposed a power management API.

References

[1] L. Benini and G. De Micheli. *Dynamic Power Management, Design Techniques and CAD Tools*. Kluwer Academic Publishers, 1998.

[2] L. Benini and G. De Micheli. Powering networks on chips. In *International Symposium on System Synthesis*, Montral, Canada, October 2001.

[3] S. Borkar. Design challenges of technology scaling. *IEEE Micro*, pages 23–29, 1999.

[4] D. Brooks and M. Martonosi. Dynamic thermal management for high-performance microprocessors. In *Int. Symp. on High-Performance Computer Architecture*, 2001.

[5] F. Catthoor et al. Code transformations for data transfer and storage exploration preprocessing in multimedia processors. *IEEE Design and Test of Computers*, pages 70–82, 2001.

[6] L. T. Clark et al. An embedded 32-b microprocessor core for low-power and high-performance applications. *IEEE Journal fo Solid-state Circuits*, 36(11):1599–1608, 2001.

[7] R. Dick et al. Power analysis of embedded operating systems. In *Design Automation Conference*, 2001.

[8] D. D. Gajski et al. *SpecC: Specification Language and Methodology.* Kluwer Academic Publishers, 2000.

[9] T. Grötker et al. *System Design with SystemC.* Kluwer Academic Publishers, 2002.

[10] A. Jantsch. Networks on chip. In *Proceedings of the Conference Radiovetenskap och Kommunikation*, 2002.

[11] S. Kumar et al. A network on chip architecture and design methodology. In *IEEE Computer Society Annual Symposium on VLSI*, 2002.

[12] S-F Li and J. Rabaey. Low power operating system for heterogeneous wireless communication systems. In *Workshop on Compilers and Operating Systems for Low Power*, 2001.

[13] Y.-H. Lu, L. Benini, and G. De Micheli. Power-aware operating systems for interactive systems. *IEEE Transactions on VLSI SYSTEMS*, 10(2):119–134, 2002.

[14] L. Luh et al. A high-speed CMOS on-chip temperature sensor. In *European Solid-State Circuits Conference*, 1999.

[15] D. S. Mikijicic et al. Process migration. *ACM Computing Surveys*, 32(3):241–299, 2000.

[16] http://www-unix.mcs.anl.gov/mpi.

[17] P. Pillai and K. G. Shin. Real-time dynamic voltage scaling for low-power embedded operating systems. In *Proceedings of 18th ACM Symposium on Operating Systems Principles*, October 2001.

[18] T. Simunic and S. Boyd. Managing power consumption in networks on chips. In *Design, Automation and Test in Europe*, 2002.

[19] W. R. Stevens. *Unix Network Programming, Volume 1 - Networking APIs: Sockets and XTI, second edition.* Prentice Hall, 1998.

[20] P. Yang et al. Energy-aware runtime scheduling for embedded multiprocessors SOCs. *IEEE Design and Test of Computers*, pages 46–58, 2001.

[21] T. Yen and W. Wolf. Communication synthesis for distributed embeded systems. In *IEEE International Conference on Computer-Aided Design*, 1995.

Chapter 13

MULTI-LEVEL SOFTWARE VALIDATION FOR NOC

Sungjoo Yoo, Gabriela Nicolescu, Iuliana Bacivarov, Wassim Youssef, Aimen Bouchhima, Ahmed A. Jerraya

TIMA Laboratory, Grenoble, France

Abstract: This chapter presents a problem in conventional methods of validating software design for NoC: software validation at different abstraction levels. As a solution to resolve the problem, a method of multi-level software validation is explained.

Key words: Application-specific operating system, abstraction level, cosimulation

1. INTRODUCTION

1.1 NoC Design and OS Role in NoC

NoC is a set of computation nodes connected via sophisticated on-chip communication network. The operating system plays a crucial role in establishing the communication over the communication network. Figure 1 exemplifies the role of OS in NoC design. In the example, we assume that we use a NoC design method in [1]. As an example of NoC design, a VDSL system is presented [2]. Figure 1 (a) gives a high-level specification where tasks (ovals in the figure) communicate with each other via communication channels (arrows in the figure). For the communication, we can have high-level communication services such as fifo (MPI [3], TCP/IP, etc.), signal, shared memory (bold arrows in the figure), polling, semaphore, multicast, etc. Each task is mapped on a processor or a HW IP

A. Jantsch and H. Tenhunen (eds.), Networks on Chip, 261–279.
© 2003 *Kluwer Academic Publishers. Printed in the Netherlands.*

Qualcomm Exhibit 1006

(rectangles in the figure represent the mapping). In Figure 1 (a), task T1, T2 and T3 are mapped on a processor #1 and task T4 to T9 on processor #2.

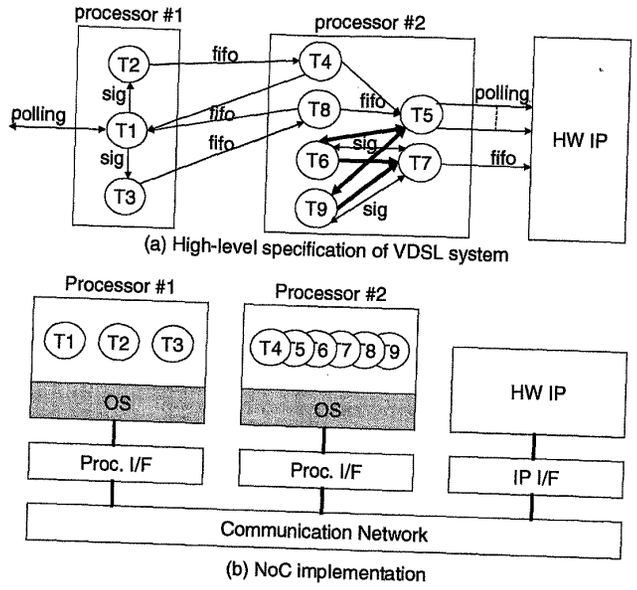


Figure 1 NoC design and OS functionality.

Figure 1 (b) exemplifies the NoC implementation from the high-level specification. In the figure, the implementation consists of processors/IP(s), processor/IP interfaces (I/Fs), and a communication network. On each processor, an OS is implemented. The communication network can be on-chip bus, packet/circuit switched networks [4][5], etc.

The OS has three major functionalities: task management, interrupt processing, and communication (I/O). For NoC design, scheduling for multiprocessor is still an important design problem as in classical multiprocessor systems design [6]. Communication gets more and more important for NoC designs since the OS needs to support complex and application-specific communication networks such as TDMA bus [7], packet/circuit switched networks [4][5] as well as complex communication services (in addition to basic ones exemplified in Figure 1 (a), e.g. fifo, shared memory, etc.) such as TCP/IP protocol, (RT) MPI [3], etc. Interrupt processing is related to both task scheduling and communication. Interrupt processing is closely related to the HW target architecture (e.g. processor types, interrupt controller, timer, communication network, etc.).

1.2 Application-Specific OS Design and Validation

Due to the tight cost and performance constraints in NoC design, the OS design needs to be optimized in an application-specific way. It is known that more than two thirds of embedded OSs are in-house ones [8][9]. Figure 2 shows the NoC implementation of the VDSL system by applying a NoC design method [1]. In the implementation, we use two ARM7 processors for processor #1 and #2 in Figure 1.

The figure shows that the two OSs for the two processors are different in terms of OS functionality. For instance, the OS for ARM7 processor, proc #2 has OS services for shared memory (**Shm**, **GShm** in the figure). However, the OS for the other processor, proc #1 does not have the functionality since the three tasks on proc #1 do not use the shared memory OS service as shown in Figure 1 (a).

In terms of OS size, the OS for proc #1 gives about 1.9 KB (1,484 byte in code and 500 byte in data) and the OS for proc #2 gives about 3.6 KB (2,624 byte in code and 1,020 byte in data) as their sizes (for more details, refer to [1]). Compared to the sizes of conventional OSs (e.g. the minimum size of VxWorks is about 130 KB [10]), the application-specific OS can give significant reduction in the OS resource usage.

Since the OS performs complex and critical functions (e.g. multi-tasking, synchronization, interrupt processing, I/O, etc.), validation of application-specific OSs needs extensive evaluation and debugging. Especially, since the OS implementation is closely related to the HW target architecture (i.e. target processor, interrupt controller, timer, peripheral devices, on-chip communication network, etc.), the interaction between the OS and the target architecture should be validated accurately.

2. MULTI-LEVEL DESIGN OF APPLICATION-SPECIFIC OPERATING SYSTEMS

Application-specific OS design needs to be performed at multiple abstraction levels. It is because the design space is huge. For instance, for OS scheduling service, we can have multiple choices, e.g. scheduling policies (round robin, RMS, EDF, etc.). In addition, for a choice of OS service, we can have many implementation choices, HW or SW or mixed HW/SW. To design an application-specific OS, for each OS service, we need to find an optimal choice and implementation. In NoC design, since we have many processors (maybe, up to or more than 100 processors), the design space for all the OSs can be extremely huge. To master the complexity of application-specific OS design, we need an incremental design and validation through multiple abstraction levels.

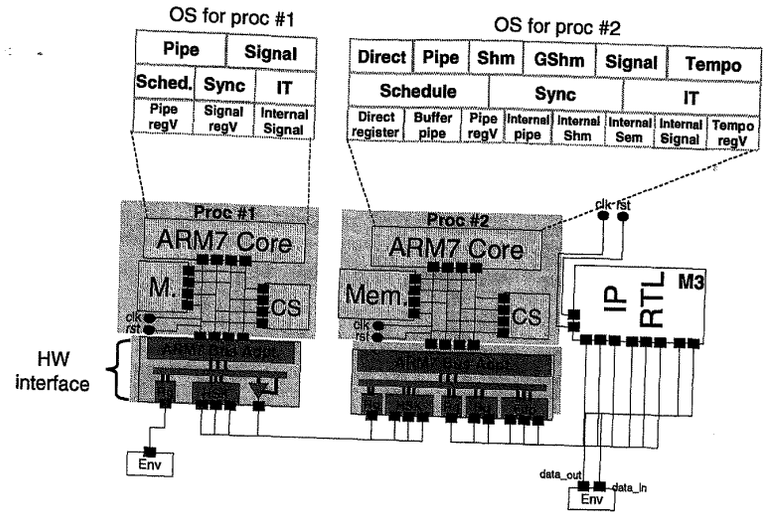


Figure 2 Application-specific OSs for the VDSL system.

2.1 Abstraction Levels in NoC Design

OS Service Categories and Abstraction Levels

To enable incremental and modular OS design, we divide the OS into hardware abstraction layer (HAL) and three service categories: task management (task creation/deletion, scheduling, synchronization, etc.), interrupt management, and I/O (communication). To enable incremental design through abstraction levels, we use three abstraction levels: OS architecture level, HAL (hardware abstraction layer) level, and ISA (instruction set architecture) level. Figure 3 shows the abstraction levels and OS service categories used in OS design.

At OS architecture level, the design objective is to determine a set of OS APIs (application programming interfaces) among possible sets of APIs, the implementations of OS services, and OS service parameter sets (e.g. task priorities). For instance, at this level, the designer can select POSIX APIs as the APIs of his/her OS design. He/she can also determine the detailed implementation of OS service, e.g. task scheduling service in hardware implementation [11][12]. Different sets of OS service parameters can also be tried. To validate the choices, the designer needs to perform simulation at OS architecture level. At this level, processor peripherals, e.g. processor interface are not yet fixed.

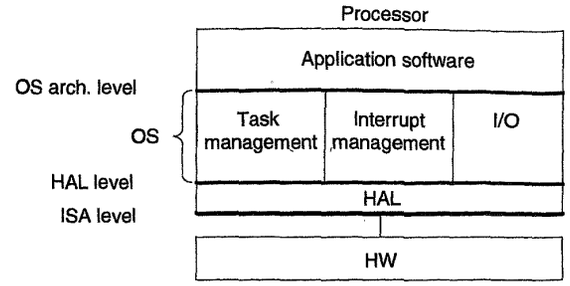


Figure 3 OS and abstraction levels.

At HAL level, the design objective is to determine the detailed implementation of processor peripherals and that of HAL. For instance, several HW timer functionalities and the HW interfaces are tried. According to that, the implementations of corresponding HAL API are also tried. For instance, for an HAL API of context switch, SW or HW implementations can be tried [11]. To validate the candidate implementations, the designer needs to perform simulation at HAL level. For more details of HAL, refer to [13].

At ISA level, all the details of OS implementation are fixed. That is, OS APIs, OS service implementations, HAL and peripheral implementations are all fixed. The design objective at this level is to obtain the final implementation of SW part (binary executable image of OS and application SW) and to validate the cycle accurate behaviour and performance of SW part.

Abstraction Levels in Communication Channel and Hardware Design

In NoC design, we use several abstraction levels for communication channel and hardware design as well as for OS. For hardware design, we use two abstraction levels: behavioural level and register transfer level (RTL). At behavioural level, only the functionality is fixed but the physical implementation of this functionality is not yet fixed. Communication with the rest of system is realized via high-level communication services. At RTL, the physical implementation is fixed.

For communication networks design, we use two abstraction levels: abstract netlist level and physical netlist level. At abstract netlist level, communication channels provide high-level communication services (e.g. fifo, semaphore, etc.). Their implementations are not yet fixed. At physical netlist level, all the communication details (i.e. communication protocols, interruption management, address decoding, etc.) are fixed at RTL.

Note that, throughout the NoC design flow, HW, SW and communication networks can be situated at any of the above-mentioned abstraction levels. Thus, the intermediate implementation contains mixed abstraction levels in HW, SW and communication networks. In our NoC design flow, to validate the mixed-level implementation, mixed-level cosimulation models can be generated. For more details, refer to [14]. In this chapter, we focus on validation of OS at different abstraction levels.

2.2 Multi-Level Validation of Application-Specific OS Design

Figure 4 shows the multi-level validation of OS implementation. At each abstraction level, the OS implementation to be validated is shown in a shaded rectangle(s). For instance, at OS architecture level, OS APIs can be validated as shown in the figure. At HAL level, OS services and HAL APIs can be validated. At ISA level, the entire OS implementation is validated.

For the validation, we use a cosimulation model where SW and HW simulation models are used for the simulation of entire system. In the cosimulation model, the OS implementation to be validated in the shaded rectangle can be OS simulation model (at OS architecture level or at HAL level) or a real OS (at ISA level).

The SW and HW models are connected with each other via a cosimulation bus. It can be an event-driven simulation environment or interprocess communication of host OS (when multiple simulators, e.g. instruction set simulators and/or HW simulators are involved in the cosimulation).

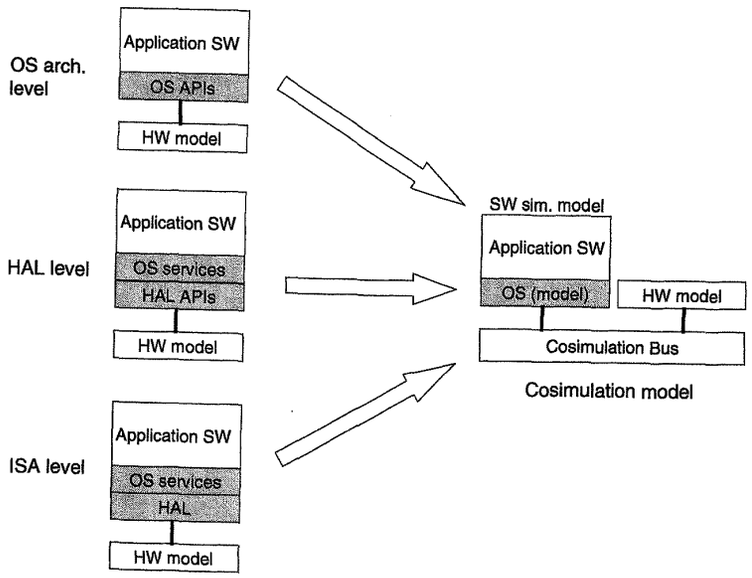


Figure 4 Multi-level validation of OS implementation.

3. EXISTING SOLUTIONS TO THE VALIDATION OF OS IMPLEMENTATIONS

Conventionally, to validate OS implementations (as well as application SW), there are two simulation methods: ISS[1] based cosimulation and native execution of OS and application SW. Figure 5 exemplifies the two methods. Figure 5 (a) shows a processor and the other part of system at RTL. The other part includes all the other RTL components (e.g. processor local bus, peripherals, on-chip communication network, other processors and HW IPs connected to the network, etc.).

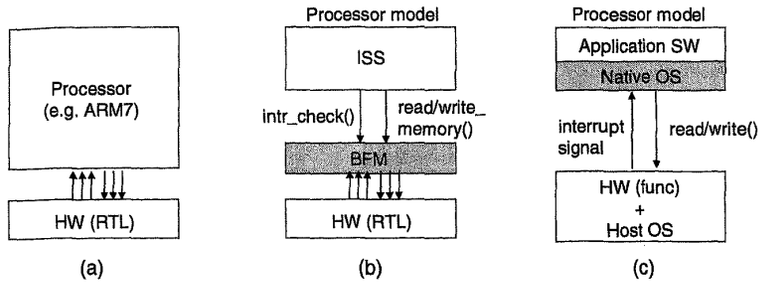


Figure 5 Conventional methods of validating OS implementations.

Figure 5 (b) shows ISS-based cosimulation method [15]. In this method, an ISS and a BFM (bus functional model) replaces the processor in Figure 5 (a). The BFM works as a cosimulation interface between the ISS and the other part of the system at RTL. The BFM provides the ISS with two kinds of API: interrupt check (**intr_check**()) and read/write operation (**read/write_memory**()) [16]. The interrupt check is used to simulate interrupt processing in the ISS. Function **read/write_memory**() transforms memory read/write operation to cycle-accurate events on the processor pins.

The advantage of this method is accuracy (at instruction/cycle/phase-accuracy). However, its main drawback is slow simulation speed. In NoC design, for instance, when there are tens of processors in the design, HW/SW cosimulation with tens of ISSs will be extremely slow.

Figure 5 (c) shows the native execution of OS and application SW (which we call **native execution** throughout the paper). In this method, application SW and OS are targeted on a simulation host. An example of commercial solution of native execution of OS (called **native OS**) is VxSim™ of VxWorks [10]. For more details of native OS development can be found in [17]. Compared to the interface between the ISS and BFM, in Figure 5 (c), the native OS calls function **read/write**() and receives **interrupt signal** (using host OS signals, e.g. Unix signal).

The advantage of native execution method is simulation speed since the application SW and OS are not interpreted by a simulator, but executed natively on

[1]. Instruction set simulator

the simulation host machine. However, its main drawback is lack of accuracy in terms of modelling SW execution time and in terms of HW modelling. It is because this method takes functional simulation for the SW and HW part of system. To be more specific, it does not simulate the effects of target processor (e.g. difference of execution times of OS and application SW on different processors). Moreover, it does not include the timed simulation of HW part of the entire system. Thus, with this type of simulation, the designer cannot distinguish the difference in the interaction between the OS and different HW implementations (e.g. different processors, interrupt controllers or physical timers, on-chip communication networks, etc.).

Recently, OS simulation models such as SoCOS [18], CarbonKernel [19] are presented. Compared to native OSs (which are real OSs), they are OS simulation models to emulate the (subset of) OS APIs, but not real OS implementations. In [18], the designer can simulate the entire NoC with SoCOS as an OS simulation model. However, to obtain a real OS after the simulation with SoCOS, the designer needs to develop again his/her OS (e.g. configuring an existing OS) which is different from the simulation model, SoCOS.

In [19], the designer starts OS design with a given basic OS. Then, he/she can add his/her OS service implementations to the basic OS. The OS can be simulated in an event-driven simulation environment. However, since he/she cannot tailor the basic OS, the simulation is limited to validating a part of the entire OS design, i.e. the designer's implementation of OS services. Another drawback of CarbonKernel is that it lacks in modelling the SW and HW interface. Although the processor peripherals can be modelled, the modelling is limited to functional modelling. Timed simulation of processor peripherals is not supported.

In terms of supporting different OS abstraction levels, there is no method to support OS architecture level in real OS design. In terms of simulation speed and accuracy, there is no method to support both fast and accurate simulation of OS at HAL level. In our work, we present a method of OS architecture level simulation and a method of fast and accurate simulation of OS at HAL level.

4. MULTI-LEVEL VALIDATION OF APPLICATION-SPECIFIC OPERATING SYSTEMS

In terms of multi-level validation of application-specific OSs, we have three combinations of abstraction levels of OS and those of communication channel.

- ISA Level - Physical Netlist Level
- HAL Level - Physical Netlist Level
- OS Architecture Level - Physical Netlist Level

Among the above three combinations, ISA Level - Physical Netlist Level was explained as ISS-based cosimulation in Section 3. In this section, the remaining two combinations are presented.

4.1 HAL Level - Physical Netlist Level

At HAL level, as explained in Section 2.1, different implementations of peripherals and HAL are tried. In terms of validation, the objective is fast and accurate simulation of the candidate implementations of peripherals and HAL. For the HW simulation, we assume that fast HW simulation methods (e.g. SystemC model [20]) can be used. In our work, we focus on the SW simulation model.

4.1.1 Timed Native Execution Model

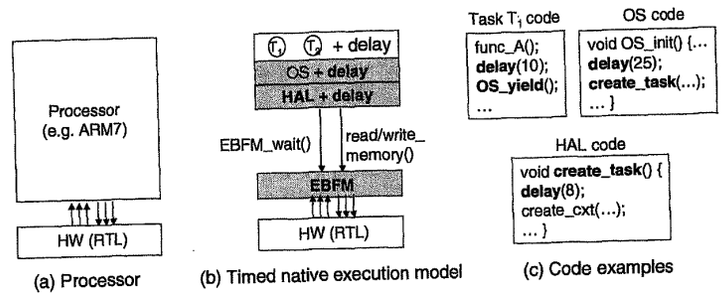


Figure 6 Timed native execution model.

For the simulation of HAL Level-Physical Netlist combination, we use a SW simulation model called **timed native execution model**. Figure 6 (b) shows the timed native execution model for the processor in Figure 6 (a). To simulate the application SW and OS running on the processor, the model consists of application SW with delay annotation (as exemplified with two tasks T_1 and T_2 in ovals in the figure), an OS with delay annotation, a timed simulation model of HAL, and a new BFM called Extended BFM, in short EBFM. Compared to the BFM in ISS-based cosimulation in Figure 5 (b), EBFM supports a new API, **EBFM_wait()** instead of **intr_check()** and the same APIs of **read/write_memory()**. More details of **EBFM_wait()** will be given in Section 4.1.3.

Figure 6 (c) shows code examples of timed native execution model: an application SW task (task T_1), an OS function (**OS_init()**), and a timed simulation model of a HAL API (**create_task()**).

Delay Annotation
To enable timed simulation, SW execution delay is inserted into the code of OS and application SW. As shown in the example of OS code in Figure 6 (c), we insert

Qualcomm Exhibit 1006

function **delay()** into the real OS code (as well as into the application SW code and into the timed simulation model of HAL). For delay calculation, we use conventional methods [21][22].

Timed Simulation Model of HAL
When building the timed native execution model, we need to separate the real OS into two parts: one is dependent on the target processor, i.e. HAL and the other is processor-independent. Since we run the OS on a simulation host, we can use the processor independent part of the OS code without change. However, for the processor dependent part, i.e. HAL, since we cannot run the original HAL (possibly in assembly code) on the simulation host, we need to build a timed simulation model of HAL that can run on the simulation host. We make the timed simulation models of HAL APIs of their codes in the assembly code of target processor. In Figure 6 (c), the timed simulation model of HAL API, **create_task** shows that function delay is inserted into the timed simulation model of HAL. Details will be given in Section 4.1.2.

Timing Synchronization between Timed Native Execution Model and Timed HW Simulation
In timed HW/SW cosimulation, timing synchronization between timed native execution and timed HW simulation is closely related to the emulation of multi-tasking operation of the OS. In our work, function **delay()** performs the synchronization. During the execution of **delay()**, the multi-tasking operation is emulated through the simulation of interrupt service routines (ISRs). Details will be given in Section 4.1.3.

4.1.2 Timed Simulation Model of HAL

To build timed native execution model, in a real OS, we replace real HAL with the timed simulation model of HAL. Figure 7 (a) shows an example of HAL API for context switch. For each HAL API that is dependent on target processor, we develop a timed simulation model. Figure 7 (b) shows a timed simulation model of the HAL API for context switch. In the example, we use SystemC as the simulation environment. We model the context switch operation using SystemC multi-threading functions, **wait()** and event **notify()**. The execution delay of HAL API code is calculated and annotated into the timed simulation model. In the example of Figure 7 (b), the total execution delay of context switch is estimated to be 37 clock cycles. For more details of building timed simulation model of HAL, refer to [27].


```
__cxt_switch        ;r0, old stack pointer, r1, new stack pointer
STMIA  r0!,{r0-r14}   ; save the registers of current task
LDMIA  r1!,{r0-r14}   ; restore the registers of new task
SUB    pc,lr,#0       ; return
END
```

(a) HAL API example: context switch

```
void context_sw(int cur_task_id, int new_task_id)
{
        delay(34);
        wakeup_event[new_task_id].notify();
        wait(wakeup_event[cur_task_id]);
        delay(3);
}
```

(b) Timed simulation model of context switch

Figure 7 An example of HAL API and timed simulation model.

4.1.3 Function Delay() and EBFM

Function **delay()** works in collaboration with EBFM. Figure 8 and 9 show pseudo codes of **delay()** and an EBFM API, **EBFM_wait()**. When function **delay()** is executed in the timed native execution model, the SW execution delay value is sent to the EBFM in line 10 of Figure 8 by calling the EBFM API, **EBFM_wait()**. As shown in Figure 9, **EBFM_wait()** advances HW (line 10 of Figure 9) simulation time watching on external events, i.e. processor interrupts (line 5).

EBFM_wait() returns in two cases. Before time period **delay** elapses, if there is a processor interrupt event, the function returns (line 5-8 in Figure 9). If there is no interrupt event during the time period, it returns after the entire time period **delay** elapses (after line 12-13).

When the function **EBFM_wait()** returns in function **delay()** of Figure 8, both SW and HW simulation times are synchronized (in line 11 of Figure 8). Note that the return value of **EBFM_wait()**, **event_value->time** is set to the current HW simulation time (**cur_HW_time**) in line 7 or 13 of Figure 9.

```
1  void delay(int delay, int granularity, int period_size, ...) {
2  int last_time;
3  static time2consume = 0;
4  time2consume[task_id] += delay:
5
6  switch (granularity) {
7        case ( instruction_level ) :
8              while( time2consume[task_id] > 0 ) {
9                    last_time = cur_SW_time;
10                   EBFM_wait(time2consume[task_id], event_return);
11                   cur_SW_time = event_return->time;
12                   time_elapsed = cur_SW_time – last_time:
13                   time2consume[task_id] -= time_elapsed;
14                   if( event_return->intr_flag == true )
15                         if( intr_mask_check() ) ISR();
16             }
17             break;
18       case (time_period) :
19             if( time2consume[task_id] < period_size ) break;
20             else // the same code from line 8 to line 16
21             ...
22 }
23 }
```

Figure 8 Function delay().

```
1  void EBFM_wait(int delay, ext_event* event_value) {
2        target_SW_time = cur_HW_time + delay;
3
4        while( cur_HW_time < target_SW_time ) {
5             if( proc_intr->new_event == true ) { // check the processor interrupt
6                   event_value->intr_flag = true;
7                   event_value->time = cur_HW_time;
8                   return;
9             }
10            advance_HW_time();
11       }
12       event_value->intr_flag = false;
13       event_value->time = cur_HW_time;
14 }
```

Figure 9 Function EBFM_wait().

If there is an interrupt before time period **delay** elapses, there is a remaining delay for the preempted SW task or OS code. Thus, the remaining delay value **time2consume** is calculated (line 13 in Figure 8). Then, if there is an interrupt and the interrupt mask allows the interrupt processing, the ISR is simulated (line 14-15). When the ISR returns, if there remains still a time delay for the preempted SW task or OS code (line 8), function **EBFM_wait()** is called again with the remaining delay value (line 10).

Timed Simulation of Multi-tasking and Nested Interrupts

Note that the multi-tasking of OS is simulated during the execution of ISR since the OS scheduler can be called in the ISR and another (preempted) task can be resumed by the OS scheduler. In terms of modelling processor interrupt, a method to model interrupt handling is presented in [23]. In the work, it is assumed that the order of task execution does not change by the interrupt handling. In our method, ISRs can call the OS scheduler to invoke new tasks before returning to the task execution preempted by the interrupt.

Since function **delay()** can be executed even in the ISR, processing nested interrupts is simulated. For instance, while function **delay()** called in the ISR is being executed, if there arrives a new interrupt and the current interrupt mask allows its processing (line 14-15 in Figure 8), a new ISR can be executed. Timed simulation of nested interrupts is performed in this way. In terms of timed simulation of nested interrupts and delay annotation, our timed native execution model enables more accurate OS simulation than a recent commercial solution of OS simulation [24] where timed simulation of nested interrupts is not allowed and delay can be simulated in a periodic manner.

4.1.4 Experiments

We applied the timed native execution model to the cosimulation of an IS-95 CDMA cellular phone system [25][26]. We compared cosimulation runtimes between two cases: cosimulation using ISSs and cosimulation using the timed native execution model. Since the IS-95 system has four processors (two ARM7's and two 68000's), in the case of using ISSs, we run five Unix processes: two ARMulator processes, two 68000 ISS processes, and one process for HW simulation in SystemC. For the communication between processes, we use Unix shared memory. We run the simulation of 0.4 sec (20 voice frames) in real time. The ISS-based cosimulation gives 72,744 sec of cosimulation runtime (on a Sun UltraSparc Iii, 333 MHz, 262 MB main memory). In the case of cosimulation with the timed native execution model, we replace the four ISSs with the timed native execution models. In this case, we run only a single Unix process for the simulation of entire IS-95 system. The cosimulation gives 37 sec of cosimulation runtime. Compared to the cosimulation using ISSs, the timed native execution model enables orders of magnitude higher performance in cosimulation.

Qualcomm Exhibit 1006

4.2 OS Architecture Level-Physical Netlist Level

At OS architecture level, as explained in Section 2.1, design objectives are to find OS APIs, OS service implementations, OS parameter sets (e.g. task priority assignments) trying their different choices and implementations.

4.2.1 OS Architecture Level Model in HW/SW Cosimulation

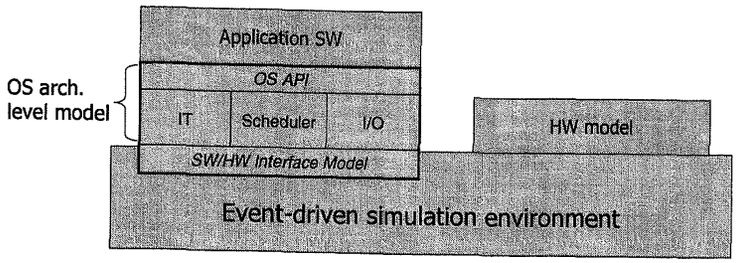


Figure 10 OS architecture level model and simulation environment.

Figure 10 shows the OS architecture level model in HW/SW cosimulation. Assuming that an event-driven simulation environment (e.g. SystemC) is used for cosimulation, the OS architecture level model communicates with HW models via the SW/HW interface model. Compared to the HAL level simulation model, i.e. timed native execution model in Figure 6, the model in Figure 10 differs in SW/HW interface modelling. In Figure 6, as the interface model, an EBFM is used while, in Figure 10, another SW/HW interface model is used.

4.2.2 SW/HW Interface Modelling

Figure 11 exemplifies a SW/HW interface model. In the figure, we assume that the communication network uses a handshake protocol and it has two signals, **request** and **ready** at its interface. Compared to the APIs of EBFM (i.e. **read/write_memory**() and **EBFM_wait**()), the SW/HW interface model at OS architecture level supports the following three APIs.

- send/receive_network()
- define_interrupt()
- OS_level_if_wait()

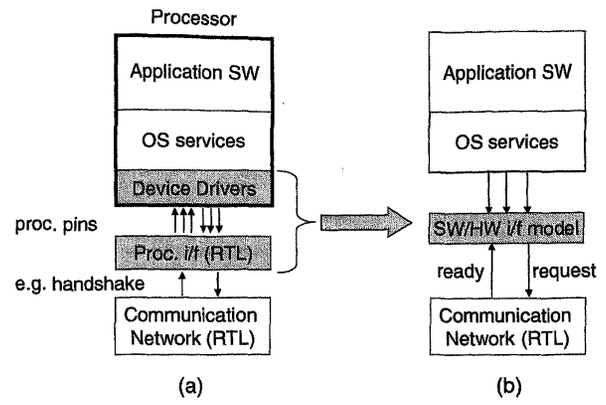


Figure 11 OS architecture level model and SW/HW interface modelling.

send/receive_network() correspond to the API, read/write_memory() of EBFM. The difference is that, as shown in Figure 11, send/receive_network() model both device drivers and processor interface while read/write_memory() model the cycle-accurate events at the processor interface. For instance, when a packet switch network is used as the communication network, the interface models device drivers for packet processing (control and transfer) functions and models the processor interface that will connect the processor and the communication network.

define_interrupt() allows the designer to model the source of processor interrupt at OS architecture level. For instance, when an event at the communication network interface needs to be propagated to the OS, the event can be defined as a source of processor interrupt using the API. For instance, in Figure 11, an event at the communication network interface, e.g. **ready** == '1' can be defined as a source of processor interrupt.

OS_level_if_wait() has the same function as **EBFM_wait**() in Figure 9 except that the processor interrupt test in line 5 of Figure 9 is replaced by the test of interrupt sources defined with **define_interrupt**().

4.2.3 Experiment: Investigating a Priority Inversion Problem

As an application of OS architecture level model to OS design, Figure 12 shows a case of investigating a priority inversion problem. In the figure, we assume a task priority assignment as shown in the figure (task T2 has the highest priority). Task T1 and T2 use a mutex for their synchronization.

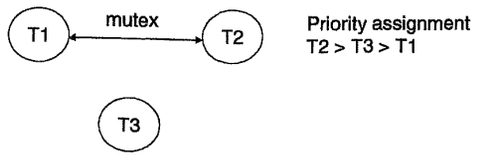


Figure 12 A priority inversion problem.

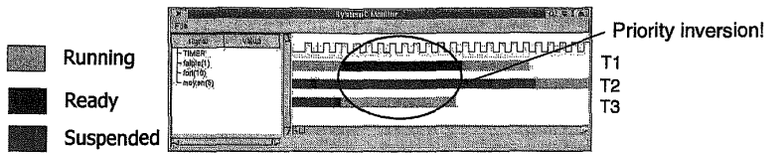


Figure 13 Detection of priority inversion by OS architecture level simulation.

Figure 13 shows a snapshot of simulation at OS architecture level for the example in Figure 12. In the snapshot, the designer can find a priority inversion problem where task T1 is holding the mutex, the highest priority task T2 is suspended waiting on the mutex release, and the execution of task T1 is preempted by task T3. At OS architecture level, the designer can investigate such a problem and try another choice of OS implementation. In the above case, to resolve the problem, the designer needs either to change task priorities (giving a higher priority to task T1 than task T3) or to use a mutex that supports priority inheritance protocol.

As shown in the above example, the advantage of OS architecture level simulation is that the designer can locate problems like priority inversion in an early stage of OS design. Thus, compared to the case where such a problem is found and debugged at HAL or ISA level, OS architecture level simulation enables to shorten the design cycle in OS design by eliminating the refinement of OS down to the low abstraction levels.

5. COSIMULATION MODEL GENERATION

To enable to reduce design cycle in OS design through multiple abstraction levels, at each abstraction level, cosimulation models need to be developed in a fast way. In our design flow, we generate cosimulation models as illustrated in Figure 14. The simulation model generator receives as input a specification called virtual architecture [1]. To generate simulation models, we use a simulation library where we have templates for simulation interfaces. The generator analyzes the virtual architecture specification and determines the functionality of each of required simulation interface, i.e. adaptation of different languages or different abstraction levels. According to the functionality, the generator extracts the appropriate

elements from the simulation library and configures the simulation interface with them. The generator also generates simulation interfaces for HW modules. For more details of simulation model generation, refer to [14].

Figure 14 also exemplifies simulation models for multi-level validation of OS implementations. For the cosimulation of ISA Level-Physical Netlist combination, we use simulators corresponding to RTL components: instruction set simulators (ISSs) for processors, HW simulator (e.g. HDL simulator) for HW cores, etc. For HAL Level-Physical Netlist Level combination, we use timed native execution model for the simulation of OS implementation, EBFM for processor interface modelling, and HW simulator(s). For the combination of OS Level-Physical Netlist Level, we use a simulation of OS architecture level, a SW/HW interface model and HW simulator(s).

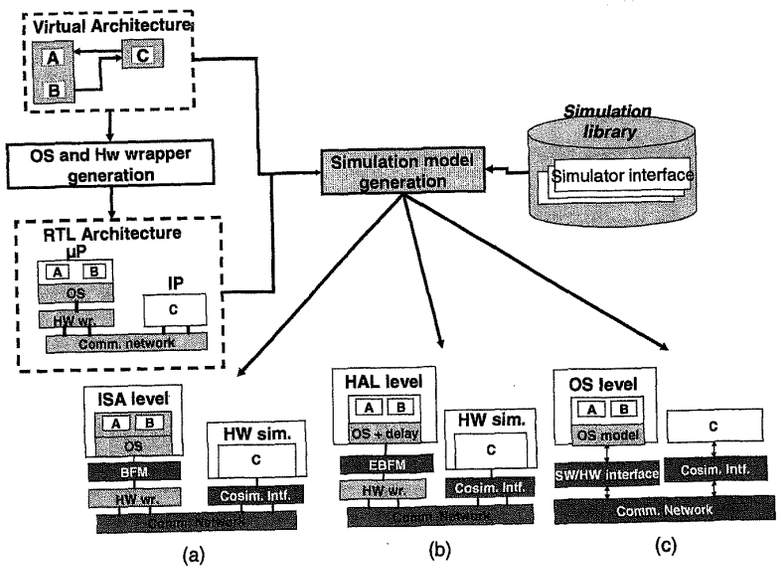


Figure 14 Automatic generation of simulation models and wrappers for multiprocessor SoC validation.

6. SUMMARY

In this chapter, we introduced application-specific OS design in NoC design and explained that incremental OS design is necessary. To support multi-level OS design, we defined OS abstraction levels: OS architecture, HAL, and ISA levels. To enable validation of OS implementation at each abstraction level, we presented two methods. One is for cosimulation between HAL level (OS) and physical netlist level (communication channel). Compared to conventional ISS-based cosimulation and native OS usage, it enables fast and accurate simulation of OS implementation. The other method is for cosimulation between OS architecture level and physical netlist level. It enables the designer to debug OS implementations in an early stage of OS design thereby reducing design cycle in OS design. Finally, we presented our environment for automatic generation of cosimulation models.

REFERENCES

[1] W. Cesario, A. Baghdadi, L. Gauthier, D. Lyonnard, G. Nicolescu, Y. Paviot, S. Yoo, A. A. Jerraya, and M. Diaz-Nava, Component-Based Design Approach for Multicore SoCs, Proc. Design Automation Conference, June 2002.
[2] M. Diaz-Nava and G.S. Okvist, "The Zipper Prototype: A Complete and Flexible VDSL Multi-carrier Solution," ST Microelectronics, J. of System Research, vol. 2, no. 1, Oct. 2001.
[3] Message Passing Interface, available at http://www-unix.mcs.anl.gov/mpi/
[4] P. Guerrier and A. Greiner, "A Generic Architecture for On-Chip Packet-Switched Interconnection," Proc. Design Automation and Test in Europe, 2000.
[5] J. A. J. Leiten, *et. al.*, "Stream Communication between Real-Time Tasks in a High Performance Multiprocessor," Proc. Design Automation and Test in Europe, 1998.
[6] B. Mukherjee, K. Schwan, Prabha Gopinath, "A Survey of Multiprocessor Operating System Kernels", Technical Report GIT-CC-92/05, College of Computing, Georgia Institute of Technology, Nov. 1993.
[7] D. Wingard, "Micronetwork-Based Integration for SOCs," Proc. Design Automation Conference, pp. 673-677, 2001.
[8] Startup gives developers a way to custom design a real-time OS, EETimes, available at http://www.eetimes.com/news/98/1000news/startup.html
[9] H. Takada, "μITRON: A Standard Real-Time Kernel Specification for Small-Scale Embedded Systems", Real-Time Magazine, 1997, q3.
[10] VxWorks and VxSim, Windriver Systems Inc. available at http://www.windriver.com/products/vxworks5/index.html and http://www.windriver.com/products/html/vxsim.html
[11] L. Lindh, et. al., "Hardware Accelerator for Single and Multiprocessor Real-Time Operating Systems", Proc. Seventh Swedish Workshop on Computer System Architecture, June, 1998.
[12] J. Lee, K. Ryu and V. Mooney, "A Framework for Automatic Generation of Configuration Files for a Custom Hardware/Software RTOS," Proceedings of the

International Conference on Engineering of Reconfigurable Systems and Algorithms (ERSA'02), pp. 31-37, June 2002.
[13] eCos, available at http://sources.redhat.com/ecos/
[14] Validation in a Component-Based Design Flow for Multicore SoCs, G. Nicolescu, S. Yoo, A. Boucchima and A. A. Jerraya, Proc. International Symposium on System Synthesis, Oct. 2002.
[15] James A. Rowson, "Hardware/Software Co-Simulation", Proc. Design Automation Conference, 1994.
[16] L. Semeria and A. Ghosh, "Methodology for Hardware/Software Co-verification in C/C++", Proc. Asia South Pacific Design Automation Conference, 2000.
[17] S. M. Tan, *et. al.*, "Virtual Hardware for Operating System Development", Technical rep., UIUC, Sep. 1995, available at http://choices.cs.uiuc.edu/uChoices/Papers/uChoices/vchoices/vchoices.pdf
[18] D. Desmet, *.et .al*, "Operating System Based Software Generation for Systems-on-Chip", Proc. Design Automation Conference, 2000.
[19] Carbon Kernel, available at http://www.carbonkernel.org/
[20] SystemC, available at http://www.systemc.org/
[21] M. Lajolo, M. Lazarescu, A. Sangiovanni-Vincentelli, "A Compilation-based Software Estimation Scheme for Hardware/Software Co-simulation", Proc. International Symposium on Hardware/Software Co-design, 1999.
[22] Virtual Component Codesign, Cadence Design Systems Inc. available at http://www.cadence. com/products/vcc.html
[23] J. Cockx, "Efficient Modeling of Preemption in a Virtual Prototype", Proc. IEEE International Workshop on Rapid System Prototyping, June 2000.
[24] M. Bradley and K. Xie, Hardware/Software Co-Verification with RTOS Application Code, Mentor Graphics Inc. avaliable at http://www.mentor.com/soc/fulfillment/mentorpaper_10280.pdf
[25] S. Yoo, *et. al.*, "Fast Prototyping of an IS-95 CDMA Cellular Phone: a Case Study", Proc. Asia Pacific Chip Design Languages, Oct. 1999.
[26] P. Gerin, S. Yoo, G. Nicolescu and A. A. Jerraya, "Scalable and Flexible Cosimulation of SoC Designs with Heterogeneous Multi-Processor Target Architectures", Proc. Asia South Pacific Design Automation Conference, 2001.
[27] S. Yoo, G. Nicolescu, L. Gauthier and A. A. Jerraya, "Automatic Generation of Fast Timed Simulation Models for OS in SoC Design", Proc. Design Automation and Test in Europe, Mar. 2002.

Chapter 14

Software for Multiprocessor Networks on Chip

Miltos Grammatikakis
ISD S.A.
K. Varnali 22
15233 Halandri, Greece
mdgramma@isd.gr

Marcello Coppola
ST Microelectronics
J. Horowitz 12
38019 Grenoble, France
marcello.coppola@st.com

Fabrizio Sensini
ST Microelectronics
Strada Ottava Z.I.
95121 Catania, Italy
fabrizio.sensini@st.com

Abstract: Multiprocessor SoC becomes increasingly software-intensive due to multiplatform design, real-time performance, robustness, reliability, availability, and safety constraints. In this chapter, we examine multiprocessor SoC software, by focusing on user-space, i.e. application and middleware layers, and kernel space, i.e. "RTOS, system libraries and device drivers" and hardware layers. For the RTOS substrate, which forms the backbone of system design, we relate software performance to parallel programming and concurrency issues, as well as program correctness to consistency, fault tolerance, reliability, and verification and validation aspects. A case study based on a multiprocessor set-top-box design by STMicroelectronics illustrates the complexity issues inherent to SoC software design.

Keywords: Concurrency, consistency, fault tolerance, reliability, SoC, software design

1. INTRODUCTION

Continuous advances in deep submicron technology lead to multimillion transistor IC designs that are incorporated into complex, faster and reliable embedded systems for Internet applications, multimedia cars, modern aircraft and spacecraft, robots, navigation equipment, consumer appliances, cameras and next generation telecommunication devices requiring a high level of integration. Thus, SoC interconnects one or several integrated microprocessors, memory, and application-specific peripherals, with

A. Jantsch and H. Tenhunen (eds.), Networks on Chip, 281–303.
© 2003 *Kluwer Academic Publishers. Printed in the Netherlands.*

Qualcomm Exhibit 1006

software development increasingly on the critical path of system design[1]. For efficient system design, a layered SoC approach has been used.

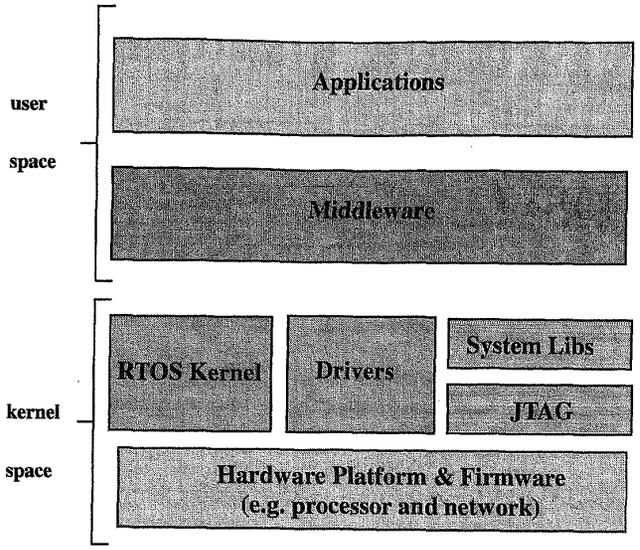


Figure 1. SoC architectural levels

As shown in Figure 1, an embedded system usually consists of four layers.

- The bottom layer consists of hardware and firmware components (processor and peripherals) providing functions to the software layers, e.g. instruction sets, timers, and memory and peripheral access.
- The next layer consists of the real-time operating system (RTOS), the system libraries and the drivers. Debug port (JTAG) software may be considered to belong to this layer.
- For firm or soft real-time systems[2], the middleware provides an API for distributed services, e.g. naming, configuration, deployment and events.
- Finally, the top layer consists of application software that is independent of the underlying hardware platform. Furthermore, specialized libraries of reusable software IP components implement the necessary functions for developing systems in given application domains.

[1]SoC designers spend 40% of their time on application code, with the remaining 40% spent on hardware and an additional 20% on making software work on top of hardware.

[2]Soft real-time systems have a minor consequence if the average response time is occasionally not within a fixed time window. Firm real-time requires an average response time always within a fixed time window. Hard real-time are usually hardware-based and require a discrete response time.

Previous embedded software was not expected to run on diverse hardware platforms and different operating systems. Nowadays, in order to decrease time between consecutive generations of products, an increasing number of companies target multi-platform software development across different generations of processors or DSPs, RTOS, or storage and peripheral devices. In order to avoid significant architectural changes, constant support calls to the original software team and duplication of testing, software developers approach multi-platform development in a reuse-oriented way through abstraction. Thus, CPUs, RTOS, storage devices, as well as interprocess communications (IPC), I/O, logging are decoupled from application algorithms. Some examples are provided below [24].

- Physical interfaces are abstracted into logical ones. This approach works well for networked applications, e. g. PCI, TurboChannel and VME can be abstracted through generic point-to-point or multi-point bus APIs.
- RTOS functions, e.g. scheduling, are encapsulated into an OS abstraction library consisting of wrapper functions posing a standard behavior irrespective of the underlying RTOS. Thus, developers may migrate applications and test-benches to different RTOS simply by porting the OS abstraction library.
- Application scaling from single node to large-scale can be accomplished by abstracting interprocess communication and synchronization (IPC).
- Configuration, installation and deployment can be automatic, using open, e.g. Unix autoconf and jam make, or proprietary environments.

Software components based on hardware-independent parts can be easily reused through intermediate standardization. Thus, *industry standard algorithms* (high-level libraries) exist between applications and middleware, and *RTOS and language standards* exist between middleware and RTOS layer. Since reusable hardware-dependent software is currently impossible, VSIA has created a working group to propose a new industry standard for a *hardware-dependent layer* that will isolate hardware from software, simplify access to hardware components, offer scalability and enable reuse for distributed systems [23]. Although this may be very hard to achieve because of the diversity of hardware platforms, it can perhaps be achieved for platforms aimed at specific application domains.

In this paper, we survey the software layers required by SoC, following a top-down approach. In Section 2 we discuss user-space applications and middleware. In Section 3, we examine real-time operating systems, focusing on parallel programming models, concurrency and real-time, consistency, fault tolerance, reliability, and validation and verification issues. In Section

4, we provide a case study based on the STMicroelectronics' set-top-box (STB) design illustrating software complexity in SoC design. We conclude this paper with important remarks and a list of references.

2. USER SPACE

2.1 Applications

In 1966, Flynn provided an interesting four-way classification of parallel architectures, based on a division between program control and data [3,9,16].

- Single-instruction single-data stream (SISD) corresponds to the traditional Von Neumann ``stored program" sequential computer.
- Single-instruction multiple-data (SIMD) enables a centralized data parallel approach, where all participating processors execute the same instruction on multiple sets of data and synchronize before executing the next instruction. In addition, most SIMD (hybrid) architectures offer fast scalar instructions through deeply pipelined units. SIMD is frequently called single-program multiple-data (SPMD). In SPMD, some processors may be selected to abstain from a particular type of operation. SIMD architectures include versions of modern general-purpose processors, e.g. Intel's Pentium and Motorola's PowerPC, traditional parallel systems, such as the TMC CM-2, and DSP architectures.
- Similar to traditional superscalar systems, multiple-instruction multiple-data (MIMD) exploits spatial instruction-level parallelism (ILP, or control parallelism), where processors may execute different instructions. Typical MIMD systems are the BBN Butterfly, Intel Paragon XP/S, Cray T3E, and the current top performance supercomputer, NEC Earth Simulator which achieves 35 trillion flops/sec. Hybrid MIMD systems can support SIMD capabilities, e.g. the TMC CM-5.
- Multiple-instruction single-data (MISD) exploits temporal ILP, by setting pipeline stages and executing several independent instructions simultaneously, e.g. vector pipelining in Cray-1. Pipelining has inherent limitations caused by hazards in dispatch logic and unavailability of ILP for creating a continuous stream of similar operations. Very-long-instruction-word (VLIW) systems were devised to reduce the complexity of issuing multiple instructions simultaneously. However, VLIWs are more complex than superscalar systems of similar performance, suffering when compilers do not have enough runtime information to efficiently schedule resources, as in non-continuous data-

streaming applications. VLIW (called now EPIC, for explicitly parallel instruction computing) was implemented in high-performance processors, such as Texas Instrument's C62x DSPs and Intel's Itanium and Merced.

In addition to control and data parallelism, new transistor-intensive methods appear promising in pursuing effectively the greater computation power required by SoC.

- Thread-level parallelism (multiple contexts or multithreading) allows relatively independent sequences of instructions (called threads) occurring between context switches to proceed in parallel. Multithreading, alike multiprocessing, provides a way to overlap computation and communication [25]. Medium-grain multithreading boosts processor throughput and execution unit efficiency, despite increasing memory latency, while fine-grain multithreading (or multiprocessing) exploits either embarrassingly parallel or data parallel applications, or coarse-grain ILP arising from independent problems. Notice that current SoCs have considerable ILP, since they integrate many different computing functions. Notice that a small number of contexts (two or three) may be sufficient for achieving most of the performance gain.
- Data prefetching based on look-ahead or long cache lines exploit spatial locality[3], thus reducing application latency. Keeping coherent multiple copies of shared variables while prefetching is very traffic intensive. Unlike multithreading, prefetching relies on the compiler or application user to insert explicit prefetch instructions for data cache lines that they would otherwise miss. This gives prefetching a greater flexibility and performance improvement over multithreading by adjusting the prefetch size (and iteration step size).
- Larger, higher utilized caches (hidden memory) exploit temporal[4] and processor locality[5] by limiting expensive (remote) main memory access, e.g., with a hit ratio of 90%, remote accesses decrease by a factor of 10.

Other software-based application latency hiding techniques are based on nonblocking communications, processor overloading by simulating a number of virtual processors, data redundancy, and relaxing memory consistency requirements. However, Mark Hill argues against using aggressively relaxed consistency models because, with the advent of speculative execution, these models do not give a sufficient performance

[3] Spatial locality occurs since memory accesses tend to refer to nearby memory words.
[4] Temporal locality occurs since memory accesses tend to refer to recently referenced words.
[5] Processor locality occurs since memory accesses tend to refer the same processor.

Qualcomm Exhibit 1006

boost to justify exposing their complexity to authors of low-level software [14]. Even without instruction reorders at least three methods for compiler-based optimizations exist: prefetching, multithreading, and caching.

Multimedia and digital signal processing (DSP) dominate the personal computer market, with multimedia data growing rapidly. These applications are similar to scientific applications involving also real-time constraints, since multimedia workloads are based on large-scale block-oriented data parallel computations on input streams of digital signals, e.g. signal filtering, encoding, and compression. Thus, multimedia processing has abundant fine-grain data parallelism making it appropriate for SIMD processing [5,6]. This data parallelism can be exposed to the hardware either by the programmer, or by the compiler. While F90, HPF, and sometimes C vectorizing compilers exist on supercomputers, bigger gains can be achieved by fundamentally designing new data parallel algorithms. Unfortunately few SoC programmers are trained enough to design such cost-effective solutions.

2.2 Middleware

The real challenge in SoC lies in achieving sustained performance on key applications. Thus, apart from high performance processors, low-latency, high-bandwidth interconnection networks and appropriate RTOS and device drivers, we must also deal with heterogeneous hardware and software environments, similar to mobile environments, or Internet. The middleware layer provides an interface so that multiple heterogeneous interconnected platforms can maintain a consistent shared state enabling remote execution of application programs. The focus is on Quality of Service (QoS), fault tolerance and reliability issues, security, and real-time (soft, firm, or hard) processing in heterogeneous systems.

- QoS in distributed applications relates to latency, throughput, and packet loss requirements (lost or rejected packets). QoS must also take into account future traffic requirements, e.g., arising from multimedia applications, scaling of existing applications, evolution of networks, as well as cost vs. productivity gain issues. Thus, new appropriate commercial benchmarks are necessary.
- Distributed system dependability and maintainability models analyze transient, intermittent, and permanent hardware and software faults. A robust system could rely on system rollback and work redistribution.
- Distributed system security is based on validating associated transaction management, authentication, certification, statistical logging, and interoperability procedures.

- Real-time constraints are often expressed as predictable or bounded delays, bit error rates, or inaccessibility, e.g. n times down with duration of at most t units of time. While soft and sometimes firm real-time can be achieved either through general distributed systems implementing a client-server approach, or distributed transaction systems, specialized systems based on optimized CPUs and direct application calls to RTOS are needed for hard real-time. Thus, for hard real-time systems, there is no middleware layer requirement.

Distributed transaction systems coordinate transactions using a concurrency control protocol that implements the following ACID[6] properties.

- **A**tomicity: an atomic transaction modifies its state with no observable intermediate steps.
- **C**onsistency: a transaction must produce consistent results; otherwise it aborts and returns all data to its state before the transaction was started.
- **I**solation: a transaction must behave as it would in single user mode.
- **D**urability: the results of a transaction that has completed successfully are not lost during a system crash.

Since no single interface or operating system is appropriate for all users, there is currently an attempt at building a general object-oriented communication interface for running distributed software written in various programming languages and spanning different OS. Unfortunately, the approach is not uniform, resulting in industry standard "container architectures" like CORBA, Java Enterprise Beans, the component object model (COM/DCOM) and several customized tools. All these technologies are based on object-level, local or remotely accessible components involved in distributed transactions, with client requests delegated to the desired component by the surrounding container environment, which can modify the transaction context.

3. KERNEL SPACE

3.1 Real Time OS – Drivers - System Programs

A real-time operating system (RTOS) is a special purpose operating system with the necessary features to support embedded (firm or hard) real-time applications whose correctness depends not only on the correctness of the logical result of the computation, but also on its delivery time. Explicit and

[6] ACID can be achieved by a two-phase commit (2PC), which ensures that either all sites commit to transaction completion, or none does and the transaction is rolled back.

Qualcomm Exhibit 1006

implicit timing constraints are derived in the requirements phase by examining the physical environment. In extreme real-time systems, an RTOS may have additional reliability[7], robustness, availability and safety constraints[8]. A RTOS consists of several modules, such as system tables, scheduler, communication, synchronization, and interrupt-service routines.

System tables provide
- a task descriptor for running each task,
- a device descriptor for using each I/O device, and
- service descriptors specifying parameters for RTOS requests.

The RTOS scheduler (dispatcher) determines which process will run next after a time-out or a blocked process triggers a context switch[9] to a new task. Processes alternate in a ready, running or suspended state. For each of these states there is usually a queue containing the corresponding process descriptor. Since processes are broken into threads, scheduler operations include thread create, run, suspend and exit operations, as well as synchronization primitives, e.g. mutex lock and unlock. RTOS scheduling algorithms must allow each task to execute on time, if this is possible. Several scheduling algorithms have been proposed, such as
- polled-loop, i.e. round-robin rotating between all tasks,
- phase- or state-driven based on discrete FSM states,
- interrupt-driven based on priorities,
- multitasking or parallel environments exploiting program locality,
- preemptive schemes that context switch upon high priority requests, and
- hybrid schemes based on a combination of these algorithms.

The RTOS includes various communication and synchronization primitives.
- Locks provide for mutual exclusion.
- Semaphores describe both mutual exclusion and scheduling constraints. They update atomically a counter indicating the number of system resources, thus providing a simple synchronization and mutual exclusion mechanism for several problems, such as producer/consumer synchronization with bounded buffers.
- Conditional variables offer mutual exclusion based on predicate calculus.
- Event flags allow a task to wait for a single condition or a combination of conditions represented as bits in a vector sequence. Thus, a consumer may wait for a producer to signal a necessary condition prior to consuming data. Event flags may have broadcast semantics, i.e. when a task sets some bits in the flag value, all tasks whose requirements are now satisfied are woken up one by one in priority.
- Signals provide asynchronous event processing and exception handling.
- Communication is based on queues, i.e. message queues and pipes for multiple messages, or mailboxes for single messages.

[7] Reliability is the rate of system failures. Safety is the degree in which system functions do not lead to accidents causing loss. Reliability does **not** imply safety and vice versa.
[8] Modifiability, portability, reusability, integration and testability may be less important.
[9] Context switch corresponds to saving and restoring info in order to switch between tasks.

An interrupt-service routine (ISR) runs on the processor for each I/O device and may trigger a system interrupt. The ISR controls the I/O device, providing data transfer between the device and the RTOS. High-priority ISRs are executed quickly.

Modern RTOS provides fast, predictable behavior with respect to timing (by providing formulas for context switch, reschedule, page faults, pipelined execution, synchronization, and interrupt handling), dynamic process priorities[10], pre-emptive scheduling, device driver support, graceful error handling and recovery, and tools for analyzing real-time scheduling using rate monotonic analysis techniques (RMA).

A device driver provides a well-defined API between hardware and software controlling setup and operation of an IP block, e.g. initialization, run-time configuration, testing, enable/disable function, and I/O processing. Driver operation is based on events that capture interrupts sent from IP blocks. In general, more than 50% of the silicon in SoC is dedicated to IP blocks, with 15% of the overall system code dedicated to device drivers. Device drivers are called frequently with their response time being critical to overall SoC performance. They also reduce hardware ambiguity, increase IP reuse and enable multiplatform software development at reduced cost and risk.

A common approach to designing device drivers for multiprocessor SoC is to partition driver functionality into two levels.
- Low level drivers provide basic functionality and are generally not re-entrant. They exploit the available communication and synchronization protocols that today's SoC bus and tomorrow's network on chip [2] offer for exchanging info between application and peripherals. For example, in the latest products based on STMicroelectronics' STbus crossbar we can decide to transfer information using a simple read or write, a set of consecutive reads or writes, or a pipelined transfer.

[10] Static priorities are established at compile/link time, while dynamic ones at run time.

Qualcomm Exhibit 1006

- High-level device drivers are abstract and have complex functionality. They use low-level drivers for accessing the hardware. These drivers are re-entrant, since they interact with the application. They must be able to handle peripheral activity, while the application is running.

Central SoC testing is inadequate, since timing depends on devices. Thus, operational testing is needed in addition to developmental testing. Special debug port with software is required to meet test goals, e.g. JTAG.

Commercial RTOS systems have become widely popular, with the market growing at some 35 percent each year.

- RT Mach supports predictable and reliable firm real-time Unix processes and real-time synchronization. RT Mach supports static process priorities with earliest deadline first scheduling, rate monotonic scheduling and priority inheritance protocols.
- CMU's Chimera has a priority based, multitasking microkernel with real-time scheduling based on fixed or dynamic priorities, with RMS[11] and earliest deadline, or maximum-urgency-first algorithms. Memory has no inter-task or intra-task protection, no paging and no virtual memory. Devices have Unix-like abstractions. Chimera is highly predictable, with fixed context switch time, dynamic reschedule and semaphore non-blocking calls. It is robust to processor exceptions and memory corruption, and provides error signaling.
- Mentor Graphics Corp. is making VRTXoc, the real-time executive OS SoC version, available through a community-source model.

For systems with multiple CPUs, parallel operating systems are based on simple modifications of the Unix OS, e.g., CMU's Mach OS. Symmetric microkernels execute on each processor, providing graceful degradation for faults. Each microkernel includes libraries for interprocess or external communications, I/O, interrupt handling, process and memory management. In addition, it provides support for parallel languages, interactive/batch execution, security, and accounting. The latest parallel systems provide, along with computing nodes, dedicated OS and I/O nodes. A new interesting research effort is towards fault tolerant distributed OS, such as Stanford's Hive Unix-like OS prototype.

[11] Rate monotonic scheduling (RMS) is based on task rates, e.g. the highest rate task has the highest priority, while rate monotonic analysis (RMA) analyzes timing constraints using external tools offline.

Today's multiprocessor applications have diverse computation, communication and I/O requirements leading to a wide range of QoS criteria, e.g. network bandwidth, latency, and packet loss. Thus, multiprocessors must support space sharing, providing efficient allocation of applications to distinct, independent subsets of processors. An important issue in processor allocation (called offline scheduling) and dynamic job scheduling problems is to provide contention-free partitions that maximize processor utilization, and minimize system fragmentation. Fragmentation prevents idle processors from being utilized again. It comes in two flavors:

- Internal fragmentation occurs when allocation assigns jobs to subsystems of certain size, thus causing any additionally allocated processors to be wasted.
- External fragmentation occurs when allocation fails to assign idle processors. This may happen due to:
  - ➢ Insufficient resources, i.e. too few idle processors,
  - ➢ Virtual fragmentation due to imperfect subsystem recognition of the allocation method, i.e. undiscovered idle processors.
  - ➢ Dynamic job departures that fragment the system into randomly scattered processor regions; although the total number of free nodes may be sufficient, there is no subsystem large enough to accommodate the incoming job.

Processor allocation and dynamic job scheduling has been studied especially for direct networks, including mesh, tori, and hypercube [8]. Depending on the network architecture and topology, constraints exist for the shape, e.g. embedded mesh or ring onto hypercube, and size of the partition[12], e.g. the number of processors must be an integer power of 2. Of particular importance is real-time scheduling, where each job is associated with a computation time and a deadline, in addition to a system dimension requirement. The goal of real-time scheduling is to determine whether all jobs can complete their processing before their fixed deadlines, and to find an efficient schedule; determining a schedule such that all jobs meet before their respective fixed deadlines when preemption is not allowed is usually an NP-complete problem. Thus, most research concentrates more on heuristic scheduling algorithms for non-preemptable real-time jobs.

While our demand and dependence on software grows, our ability to produce it in a timely, cost efficient way continues to decline. We next

[12] Most commercial multiprocessors, e.g. Intel Paragon and Cray T3E, can allocate variable size subsystems to incoming jobs. Network partitioning is usually implemented within the network layer (or lower) in order to take advantage of system architecture issues.

Qualcomm Exhibit 1006

discuss software design problems in SoC, focusing on parallel programming paradigms, concurrency and real-time systems, consistency, fault tolerance and reliability, validation and verification.

3.2 Parallel Programming Models

Two parallel programming models are possible. Explicit parallel programming makes the user specify exactly how sequential processes cooperate. In this case the compiler is implemented in a straightforward manner. Implicit parallel programming uses sequential algorithms and a "smart" compiler for parallelizing an application. Implementations using implicit parallelism are usually not as efficient as explicit ones.

In explicit parallel programming several paradigms are used, such as message passing, data parallel and (virtual) shared memory [3,9,16]. These paradigms differ in various aspects, such as:

- the programmers invest in writing parallel programs,
- the way code distribution and data allocation are specified,
- the architecture for which the paradigm fits best, and
- the maximum amount of application concurrency that can be exploited.

In message passing, processes update local variables and exchange information using explicit send/receive operations. Hardware platforms naturally suited for this model are multicomputers with point-to-point connections and distributed clusters. Message passing is easy to emulate on other platforms. Message passing languages are sequential languages augmented with standard message passing libraries, such as PVM, MPI (version 1.2 and a subset of version 2.0) and MPI-RT for real-time.

Data parallel programming supports single-instruction multiple-data (SIMD), or single-program multiple-data (SPMD) abstraction. Data parallelism is expressed by assigning data to virtual processors, which perform identical computation on its data. Since data parallel programs have only one control flow, problems with inherent distributed control cannot be parallelized efficiently. The compiler inserts synchronization constructs needed for parallel execution into the code, thus avoiding deadlocks or data races. Some compilers, e.g. HPF or C* provide data parallel behavior on various platforms, such as SIMD systems and distributed clusters.

In shared memory programming, processes read and write directly from/to a shared address space. The programmer has to express code distribution only, since most data is stored in global memory. Since data can be accessed

concurrently by different processes, mutual exclusion operations are vital. Shared memory languages consist of sequential languages, together with special libraries providing memory access and synchronization subroutines. For example parts of MPI-2, Open-MP, and ShMem are shared memory programming utilities.

In virtual shared memory (VSM) a global virtual address space is shared among loosely coupled processing nodes. Although distributed memory is viewed as a shared address space, the VSM programmer has to bear in mind that remote memory access may significantly increase memory latency, and thus he must try to exploit spatial, temporal and processor locality. Virtual shared memory languages are similar to shared memory languages. Nowadays most commercial supercomputers are NUMA VSM systems.

3.3 Concurrency and Real Time Systems

Pseudo concurrency is equivalent to multithreading on a single processor, while actual concurrency involves many processors and/or processes with instructions executed simultaneously. While sequential algorithms impose a total order on the execution, parallel algorithms only specify a partial order. Thus, a parallel algorithm is equivalent to two or more algorithms each specifying part of the final result. Concurrency can dramatically increase real-time performance by increasing the degree of simultaneity. However, parallel programs are complex to design, test, integrate and maintain, since major issues concerning resource management are introduced.

- To ensure correct application functionality the system must be free of deadlock. A deadlock may be defined as a cyclic dependency of ungranted resource requests for buffer or channel resources. Deadlocks may occur when processes mutually exclude each other from obtaining resources from a shared pool of resources.
- Starvation refers to tasks repeatedly being denied resources. Starvation usually reflects problems of fairness in static allocation or dynamic scheduling policies in the CPU, or system interconnect.
- Data races arise when threads or parallel processes access shared variables and at least one access is a write operation, while there is no synchronization as to the order of accesses to that variable.
- Priority inversion occurs when a lower priority task holds a resource required by a higher priority task that busy waits on the same resource. Priority inversion may lead to timeouts causing continuous system resets, as in the case of the fail-safe [13] Mars Rover landing vehicle.

[13] On a fail-safe system a fatal error causes system reset, e.g. submarine float operation.

Qualcomm Exhibit 1006

Deadlock, starvation, data races and priority inversion pose functional risks to parallel programming in real-time systems, including data consistency and system failure, since debugging can be very frustrating and costly. Thus, new tools that identify the sources and evaluate the effect of priority inversion are very important in scheduling analysis. Existing program analysis tools, such as the Eraser that detects deadlock and data races, are major steps in eliminating these problems [22]. In respect to deadlock, two techniques review this problem from different angles.

- Deadlock detection and recovery can be based on resource dependency graphs. This method is simple and effective only if the state of the resources and tasks is available.
- Run-time deadlock prevention based on spin locks[14] is the most effective strategy, but in its simplest form it offers no guarantee of fairness and has an increased overhead due to loss of flexibility resulting from a static or adaptive prevention strategy. These costs are alleviated in modern processors and RTOS using distributed synchronization mechanisms, such as Ticket lock and MCS lock that prioritize requests, remove central access mechanisms and allow for efficient resource management [18]. In this case, the system must provide good primitives and suitable libraries, and the programmer should exercise basic caution and alertness in avoiding common pitfalls.

In terms of performance, hierarchical locking refers to the ability to lock small parts of concurrent data structures, or even program code, e.g. critical sections, for exclusive access, thus increasing the level of concurrency. Lock granularity is an important tradeoff between increased concurrency and complexity, e.g. in database access one may hierarchically lock the complete database, particular tables, or data elements [8].

3.4 Consistency

A memory consistency model must be specified for every level at which an interface is defined between the programmer and the system, eliminating the gap between expected program and actual system behavior. We examine techniques that prevent, or detect and recover from this incoherence.

[14] A spin lock ensures that only one processor may modify a critical section in a shared data structure at any given time. Spin locks execute enormous number of times in concurrent data structures, e.g. in RTOS priority queues, or fault tolerance recovery techniques.

For a single von Neumann processor, program order implies that for any memory location, a read routine to that location will return the value most recently written to it. However, for a multiprocessor, we must specify the order in which memory operations may be observed by other processors. Thus, either implicit (e.g. in an SIMD system) or explicit synchronization (e.g. via semaphores or barriers) may be necessary when a new value is computed and other processes access this value [1].

The use of atomic hardware primitives, e.g. test&set, fetch&increment, fetch&add, fetch&store, compare&swap and load-linked/store-conditional with the added guarantee that all pending memory operations are completed before such an atomic operation, provides several potential advantages:

- concurrent protocols are easier to code,
- code is easier to understand and debug, and
- more efficient implementations are possible.

The simplest consistency model for shared memory programming is sequential consistency (SC). SC was defined as follows. "A multiprocessor is sequentially consistent if the result of any execution is the same as if the operation of all the processors were executed in some sequential order, and the operation of each individual processor appear in this sequence in the order specified by its program [17]."

Although SC provides a simple and intuitive model, it disallows some hardware and compiler optimizations, e.g. prefetching or nonblocking write, that are possible in uniprocessors by enforcing a strict order among shared memory operations [14]. In practice, we encounter many relaxed memory models[15], as in IBM 370, total store ordering model (TSO), partial store ordering model (PSO), processor consistency (PC), weak ordering (WO), release consistency (RC), PowerPC, relaxed memory order (RMO) and Alpha consistency [1]. It is either the programmer's, or the compiler's responsibility to provide synchronization operations enforcing a consistent memory view, e.g., by placing a fence after every memory operation, or more intelligently by looking for potential access cycles.

3.5 Fault Tolerance and Reliability

SoC is more susceptible to failure than conventional systems; transient, intermittent, and permanent hardware and software errors may occur at any

[15] The ArchTest suite analyzes shared memory behaviour using parallel accesses [7].

time. Though the probability of failure of a single chip decreases due to better technologies, the statistical chance that one element in a complex SoC system breaks down is not at all negligible. Furthermore, these errors occur especially in corner situations, arising from a mixture of cases. These errors are hard to predict using random or application-specific testing and new test generation techniques have been developed [15].

Both hardware and software (algorithmic) fault tolerance approaches have been considered. While hardware fault tolerance techniques focus on extensive replication, checksums, adaptive transaction tables and link keep-alive protocol checkers, algorithmic approaches maximize the probability of algorithm termination under faults. These techniques are explained below.

- Dynamic online fault recovery based on process adaptivity to the environment. For example, fault tolerant packet routing in the presence of transmission errors, or temporarily down links may be based on knowledge of faults in a local neighborhood.
- A process locking a resource may subsequently hang temporarily, e.g. due to cache misses or page faults, or even halt, e.g. due to faults and fail to unlock the resource within specified real-time. This may cause unacceptable delays or even starvation to another process. This problem can be avoided using non-blocking, lock-free algorithms. If there are several processes performing operations on a shared data structure, lock-free algorithms[16] guarantee that **some** process will complete in a finite number of steps [20,26,27]. Thus, lock-free data structures provide greater reliability with processor/process faults or aborts, which normally result in inconsistent (corrupt) data structures. While for lock-based methods the ``window of inconsistency" spans the entire critical section, for lock-free techniques the window is limited to a double compare&swap or load-linked/store-conditional operation. Lock-free implementations of shared linked lists and queues can be used in many concurrent algorithms, such as parallel branch and bound in databases, fuzzy search, heuristics for NP-complete problems, tree and graph algorithms, job scheduling in operating systems, and event scheduling in production and simulation.
- Static reconfiguration can be based on graph embedding [11]. Redundant fault-free processors and links or redundant data may be provided. A general (but idealized) semi-automated process for dealing with dynamic faults can be structured around reconfiguration based on four major steps.

[16] Wait-free algorithms guarantee that **each** process completes in finite steps.

  - Error detection. Two techniques are mainly used: (a) structural detection based on test code directed towards detection of memory and communication faults, e.g., parity, CRC, or self-tests, and (b) behavior-based detection based on comparing test patterns collected at compile/assembly time with the exhibited behavior at run time.
  - Fault diagnosis and isolation. Upon detection, the fault is analyzed and localized. Network devices and system routing tables are updated and recovery procedures are initiated. System diagnosis properties are based on graph theory. For example, a system is one step `t`-fault diagnosable, if given the fault bound `t>0` all faulty components can be correctly identified after a testing phase [4,10].
  - Automatic system reconfiguration based on embedding. Faulty components might be replaced by spares or existing non-faulty components, while application programs and data are repartitioned to this newly-defined fault-free set of components. In the case of packet routing, routing tables may be adapted to detour permanently failed components.
  - Consistent checkpointing during normal execution and rollback. We avoid restart of application by doing backward recovery. This is implemented through rollback to fault-free, user-inserted checkpoints.
- The information dispersal algorithm (IDA) maximizes the probability of algorithm termination under faults. IDA represents a reliable dispersal of file information into `n` locations and a subsequent file reconstruction from any `m` pieces [21]. This approach is based on error-correcting codes, since the message contained in a packet can be reconstructed at its destination from a small number of copies. Dispersal and reconstruction may be both space, and computationally efficient. IDA has been applied to file transfer in a distributed system with unreliable links by taking advantage of parallel paths that increase adaptivity and resource utilization. Similar to IDA, Huang and Abraham introduced algorithmic fault tolerance based on data redundancy using encoding [13] and global (or local) load redistribution [7]. Checksum techniques detect and correct errors when matrix operations such as addition, multiplication, scalar product, LU-decomposition, and transposition are performed on parallel systems. These techniques have been extended to Gaussian elimination and FFTs [7,19].

System reliability depends on fault detection and recovery within real-time constraints, such as peak load and maximum fault frequency. Best effort systems need not have rigorously specified load and reliability requirements. For studying system reliability, various mathematical models based on

Graph Theory and Combinatorics have been developed. Reliability models evaluate the probability of system failure.

- In the p-faulty model, component failures occur independently with a fixed probability p and may correspond to node failures, link failures, or both. These dynamic faults may correspond to a channel being unavailable, or to a full buffer.
- In the worst-case model, an adversary selects a number of faulty components (links and nodes). Such static faults could arise during fabrication runs. Notice that when a processing (or switch) node fails, all of its adjacent links also fail.
- In the binomial model we assume independent failure probabilities. The system fails when at least k failures occur. This model is equivalent to a birthday surprise problem.
- Fault trees with and/or gates are useful in evaluating static faults. General fault trees are converted to binary trees, and the probability laws for union and intersection of events are used. The intersection law in its simple product form can be used only if events are independent.
- Reachability matrices. These techniques can be used to evaluate both static, and dynamic faults. For the dynamic case, models both without repairs, and with repairs (with a number of repairmen) are considered.

System reliability models may focus on hardware, as well as software failures, e.g. due to internal functions, operational scenarios, or user input. These models can carry through to testing and maintenance phase if software is reused. They provide important benefits for real-time embedded systems, such as

- anticipating component an system failures,
- prioritizing critical component failures in terms of their severity, rate of occurrence, and probability of detection,
- predicting and analyzing risks to system functionality arising from faulty components, and
- implementing corrective actions by focusing on early failure detection, graceful degradation, and fail-safe techniques.

3.6 Validation and Verification in Real Time Systems

Verification corresponds to inspecting if there is a correct translation from each stage of the software life cycle to the next. Validation corresponds to testing the level of conformance between requirements and implementation of each stage of the software life cycle. Thus, verification in real-time systems is based on rate monotonic and event-sequence analysis and corresponds to defect prevention, while validation corresponds to defect

detection. Defect removal efficiency is defined as the relative number of defects found by customers compared to defects found prior to release. Defect-removal efficiencies in commercial software reach 99%.

System robustness is sometimes tested through black box testing based on normal and abnormal test sequences (especially those with catastrophic results) or random testing, focusing on critical and complex software modules, historically defect-prone modules, and modules developed by less-experienced developers. We distinguish three forms of testing.

- Top-down testing allows concurrent testing of software systems by replacing low-level modules with stubs. It is especially good for hierarchical systems.
- Bottom-up testing starts with low-level testing. It is effective for flat, distributed functionally partitioned, software systems.
- Regression testing is applied for assuring that modified software will still function in the same change.

Finally, individual components can be tested in two different ways.

- Dynamic property testing involves formal program correctness through control and data flow analysis, or RMA. Such techniques are useful for highly critical systems, but impractical for large systems.
- Static tests are used to estimate program quality without reference to program execution.

4. CASE STUDY: STM LINUX SET-TOP-BOX (STB)

As a case study on SoC software complexity, we introduce the STMicroelectronics' Linux STB generic reference software, a reference platform for Linux-based digital set-top-box (STB) based on the RISC CPU STGX1 and the media processor STi5514 chipset. The target hardware for this software is the MediaRef board, hosting the mentioned chips in a single board design. A precise goal of the project is software and hardware reuse, which has led (in hardware terms) to the dual processor architecture of the MediaRef board. The Linux STB software aims at implementing a high-end digital STB supporting hard disk drives as storage devices, handling multiple transport streams, e.g. recording and watching TV simultaneously, and providing flexibility for additional features, such as web browsing and graphics applications.

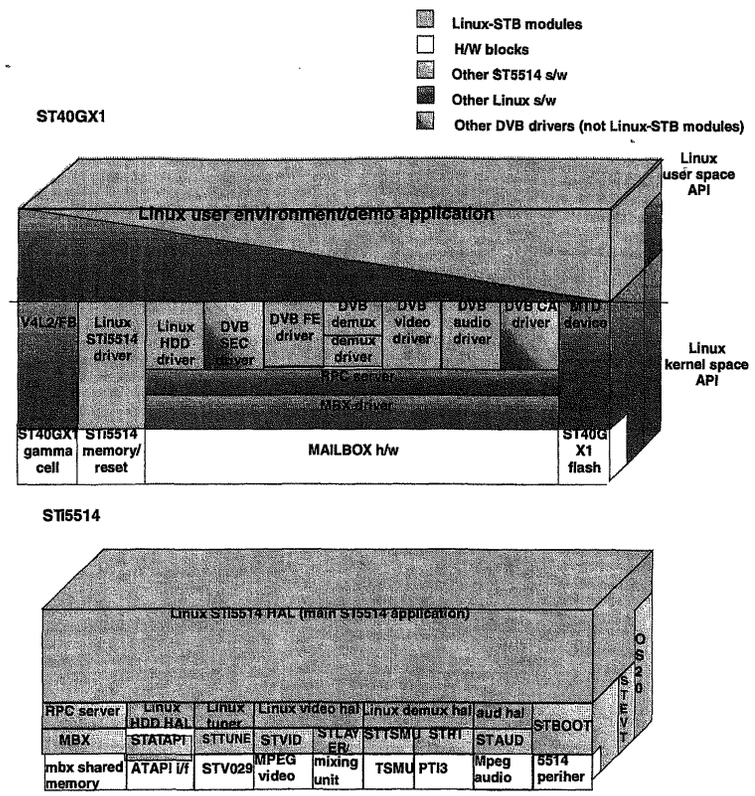


Figure 2. The Linux STB software architecture

The implementation follows open source standards in order to benefit from existing open source software. The Linux DVB API is chosen for kernel level services in STB applications, since it is the only publicly available interface specification, with many open source applications using its services. This API defines a generic interface to the devices present in a typical digital STB/VCR, e.g. satellite equipment control, satellite or cable tuner, TS/section filters, and MPEG decoders. Drivers are defined as standard Linux char devices, under the `/dev` file system.

Figure 2 shows the overall software architecture. Within this architecture, the STi5514 chip is targeted at running real-time tasks using OS20 microkernel services that provide static priority multitasking, concurrent control via Dijkstra semaphores, and message passing. The ST40GX1 is in charge of running complex, non real-time, end-user applications involving graphics and web interactivity, using the services provided by the Linux kernel. This functional split provides for

- isolation of real-time activities from non real-time ones, and
- protection of end-user applications from low-level hardware or driver faults occurring on the Sti5514 chip.

The Linux STB software stack uses the STAPI library built upon the OS20 microkernel in order to shorten development effort. STAPI is STMicroelectronics' standard set of drivers for accessing the peripherals in the Sti55XX family of chips, providing a uniform interface across different hardware platforms. To allow use of the STAPI library (and more general of code running on the STi5514 chip) an RPC mechanism is defined between the ST40GX1 and the STi5514 based on shared memory between the ST40GX1 and the STi5514 and specific hardware support.

Therefore, implementation of each Linux driver for STi5514 hosted peripherals has the following structure. A generic call to a driver function (executed in ST40GX1/Linux context) is translated into a set of calls to STAPI component functions to be executed in the STi5514/OS20 context. The context switch is not always immediate, and driver functions are usually implemented as a succession of calls to higher level functions, which use the STAPI interface. In any case, control needs to be passed to the STi5514, i.e. a function call implemented as RPC towards the 5514 performs the last part of the processing, including calls to the STAPI functions.

For each device, an interface is defined between the STi5514 and the ST40GX1. Figure 3 shows how the Linux devices map onto the STi5514 STAPI components. For each device the HAL (hardware abstraction layer) is a C function API. This API is exported into the Linux kernel namespace for use by kernel components. Such RPC exported API is mainly used for command and control operation. High throughput data exchange occurs using shared memory and RPC event signaling between the STi5514 and ST40GX1 processors. Concurrent access to shared memory is controlled by Dekker's two-phase lock mechanism.

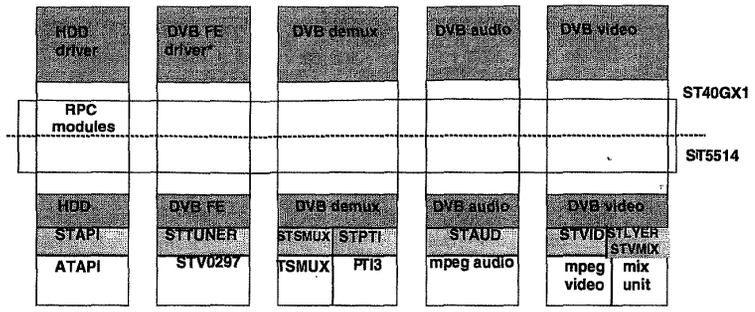


Figure 3. Mapping Linux devices onto Sti5514

5. CONCLUSION

SoC consists of components in user space, i.e. application software and middleware, and kernel space, i.e. real-time operating system, system libraries, device drivers and hardware platform. SoC applications, such as multimedia, video conferencing and HDTV are increasingly distributed, interconnected, real-time and ubiquitous. Thus, SoC design becomes software-intensive with improved performance, greater functionality and safety relying on multiprocessing, multithreading, prefetching, cache coherence, fault tolerance, and consistency aspects.

In this chapter, we have surveyed SoC software development issues, explaining caveats that make SoC software development hard, but interesting. We note that just by increasing clock frequency, without fully exploiting available concurrency in the design of new efficient parallel or fault tolerant algorithms within RTOS, system libraries, drivers, middleware and applications, SoC software development will be ineffective and particularly problematic.

REFERENCES

1. Adve, S. V., and Gharachorloo, K. "Shared memory consistency models: a tutorial". IEEE Trans. Comput. C-45 (12), 1996, pp. 1145--1155.
2. Benini, L., and De Micheli, G. "Networks on chips: a new SoC paradigm". Computer, Vol. 35 No 1, 2002, pp. 70-78.
3. Culler, D. E., Singh, J. P., Gupta, A. "Parallel computer architecture: a hardware/software approach". Morgan-Kaufmann, 1998
4. Dahbura, A.T., Sabnani, K.K., and King. L. "The comparison approach to multiprocessor diagnosis". IEEE Trans. Computers, C-36(3), 1987, pp. 373--378.
5. Diefendorff, K. and Dubey, P., "How Multimedia Workloads Will Change Processor Design". IEEE Computer, September 1997, pp. 43--45.

6. Diefendorff, K., and Duquesne, Y., "New degress of parallelism in SoCs". EE Times, Sept. 13, 2002
7. Elster, A.C., Uycar, M.U., and Reeves. A.P. "Fault tolerant matrix operations on hypercube multiprocessors". Proc. IEEE Conf. Parallel Proc., 1989, v. III pp. 169--176.
8. Grammatikakis, M. D., Hsu, D.F., Kraetzl, M. "Parallel System Interconnections and Communications", CRC press, 2000.
9. Grammatikakis, M. D., and Liesche, S. "Priority queues and sorting for parallel simulation". IEEE Trans. Soft. Engin. SE-26 (5), 2000, pp. 401--422.
10. Hakimi, S.C., and Amin, A.T. "Characterization of connection assignment of diagnosable systems". IEEE Trans. Computers, C-23(1), 1974, pp. 86--88.
11. Hastad, J., Leighton, F.T., and Newman, M. "Reconfiguring a hypercube in the presence of faults". Proc. 19th ACM Symp. Theory Comput., 1987, pp. 274--284.
12. Herlihy, M. "Wait-free synchronization". ACM Trans. Progr. Lang. Syst. C-13 (1), 1991, pp. 124--149.
13. Huang K.H., and Abraham, J.A. "Algorithm based fault tolerance for matrix operations". Proc. IEEE Conf. Parallel Proc., 1984, pp. 518--528.
14. Hill, M.D. "Multiprocessors should support simple memory consistency models". IEEE Computer C-31 (8), 1998, pp. 28--34
15. R.C. Ho, C.H. Yang, M.A. Horowitz, and D.L. Dill. "Architecture validation for processors". Proc. 22nd IEEE Symp. Comput. Arch., 1995, pp. 404--413.
16. Kumar, V., Grama, A., Gupta, A. and Karypis, G. "Introduction to parallel computing". Benjamin Cummings, 1994.
17. Lamport, L. "How to make a multiprocessor computer that correctly executes multiprocess programs". IEEE Trans. Comput. C-28 (9), 1979, pp. 690--691.
18. Mellor-Crummey, J. M. and Scott, M. L. "Algorithms for scalable synchronization on shared-memory multiprocessors". ACM Trans. Comp. Syst. C-9 (1), 1991, pp. 21--65.
19. Nair, V.S.S., Abraham, J.A., and Banerjee, P. "Efficient techniques for the analysis of algorithm-based fault tolerance schemes". IEEE Trans. Computers, C-40(9), 1996, pp. 499--503.
20. Prakash, S., Yann-Hang, L., Johnson, T. "A non-blocking algorithm for shared queues using compare-and-swap." IEEE Trans. Comput.,C-43 (5), 1994, pp. 548--559.
21. Rabin, M. "Efficient dispersal of information for security, load balancing and fault tolerance". J. ACM, 36(2), 1989, pp. 335--348.
22. Savage, S., Burrows, M., Nelson, G., et al. "Eraser: A dynamic data race detector for multi-threaded programs." Proc. 16th ACM Symp. on OS Princ., Saint-Malo, France, 1997, pp. 26--37.
23. Shandle, J., and Martin, G. "Making Embedded Software Reusable for SoCs". EEDesign, March 1, 2002.
24. Stolper, S.A. "Software that travels". EE Times, Oct. 1, 2002.
25. Tullsen, D.M., Eggers, S.J., and Levy, "Simultaneous multithreading: maximizing on-chip parallelism". Proc. 22nd IEEE Symp. Comput. Arch., 1995, pp. 392--403.
26. Turek, J., Shasha, D., and Prakash, S. "Locking without blocking: making lock based concurrent data structure algorithms nonblocking". Proc. 11th ACM Symp. Princ. Database Syst., 1992, pp. 212--222.
27. Valois, J. "Lock-free linked lists using compare-and-swap". Proc. 14th ACM Symp. Princ. Distr. Comput., 1995, pp. 214—222.