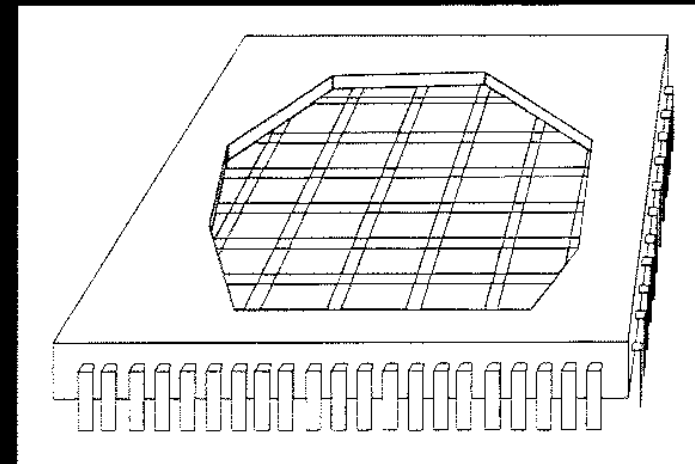


Networks on Chip

Axel Jantsch and Hannu Tenhunen (Eds.)



Kluwer Academic Publishers

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

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? <i>Axel Jantsch and Hannu Tenhunen</i>	3
2 A Design Methodology for NoC-based Systems <i>Juha-Pekka Soininen and Hannu Heusala</i>	19
3 Mapping Concurrent Applications onto Architectural Platforms <i>Andrew Mihal and Kurt Keutzer</i>	39
4 Guaranteeing The Quality of Services in Networks on Chip <i>Kees Goossens, John Dielissen, Jef van Meerbergen, Peter Poplavko, Andrei Rădulescu, Edwin Rijpkema, Erwin Waterlander and Paul Wielage</i>	61
Part II Hardware and Basic Infrastructure	
5 On Packet Switched Networks for On-chip Communication <i>Shashi Kumar</i>	85
6 Energy-reliability Trade-off for NoCs <i>Davide Bertozzi, Luca Benini and Giovanni De Micheli</i>	107
7 Testing Strategies for Networks on Chip <i>Raimund Ubar and Jaan Raik</i>	131

vi		
8		
Clocking Strategies for Networks on Chip		153
<i>Johnny Öberg</i>		
9		
A Parallel Computer as a NoC Region		173
<i>Martti Forsell</i>		
10		
An IP-Based On-Chip Packet-Switched Network		193
<i>Ilkka Saastamoinen, David Sigüenza-Tortosa and Jari Nurmi</i>		
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
<i>Eric Verhulst</i>		
12		
NoC Application Programming Interfaces		239
<i>Zhonghai Lu and Raimo Haukilahti</i>		
13		
Multi-level Software Validation for NoC		261
<i>Sungjoo Yoo, Gabriela Nicolescu, Iuliana Bacivarov, Wassim Youssef, Aimen Bouchhima and Ahmed A. Jerraya</i>		
14		
Software for Multiprocessor Networks on Chip		281
<i>Miltos Grammatikakis, Marcello Coppola and Fabrizio Sensini</i>		

NETWORKS ON CHIP

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

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 assess 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

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

3

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

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 mathematical 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

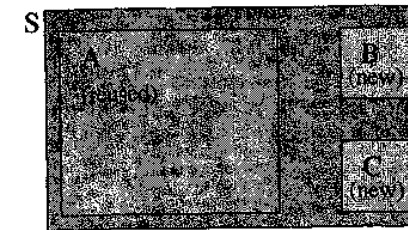


Figure 1.1. With the arbitrary composability property the design effort to add new components to an existing system depends on the integration effort and on the new components but not on the size of A because the design effort to build A has already been spent and is reused as well: $\text{Deffort}(S)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

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 $A_1, \dots, A_n$ 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: $\text{Ieffort}(n)$. Thus the total design effort for S is*

$$\text{Deffort}(S) = \text{Deffort}(A_1) + \dots + \text{Deffort}(A_n) + \text{Ieffort}(n)$$

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

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 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 (NoC) 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.

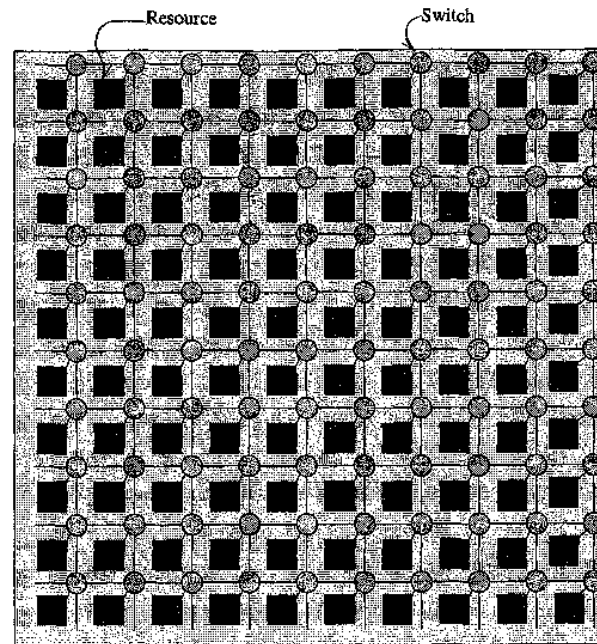


Figure 1.2. Each node in the mesh contains a switch and a resource.

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.

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

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 maxi-

mum 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 reduced 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

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 losing 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

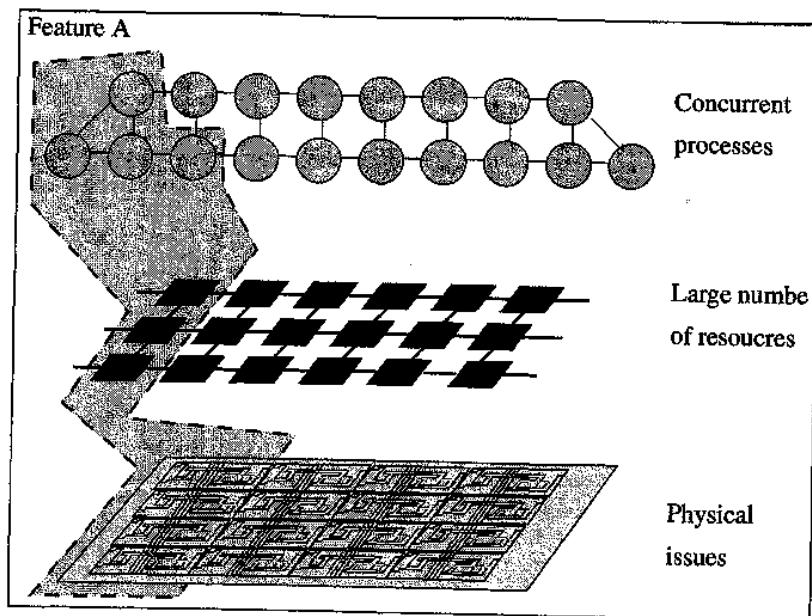


Figure 1.3. Features and components must be arbitrarily composable at the physical, the architectural and the application level to facilitate seamless feature integration.

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. International 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 Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 19(12):1523–1543, December 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 Soiminen, 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 SOC's. 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 Strumpfen, 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 Academic 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].

19

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, SOC 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

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 SOC 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 SOC. 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 losing 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.

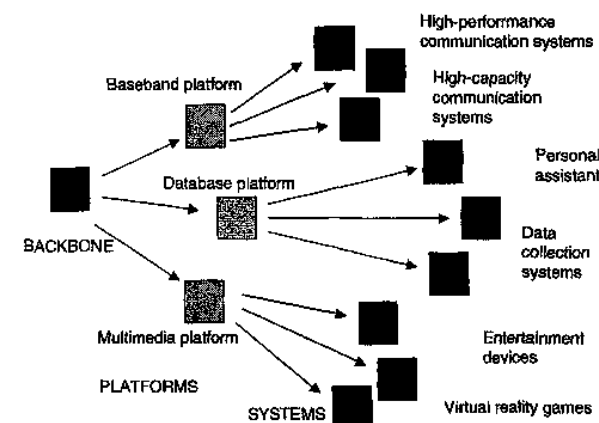


Figure 2-1. NOC-based product family development

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.

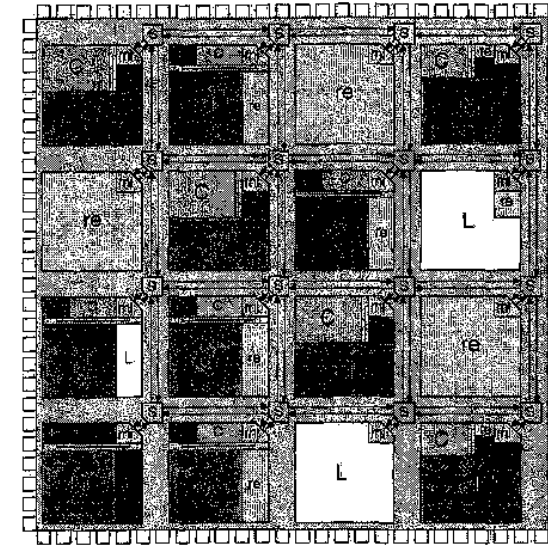


Figure 2-2. Example of a network on chip architecture. S = switch, rni = resource-network interface, P = processor core, C = cache, M = memory, D = DSP core, re = reconfigurable logic, L = dedicated hardware.

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 METHODOLOGY

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.

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 the 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.

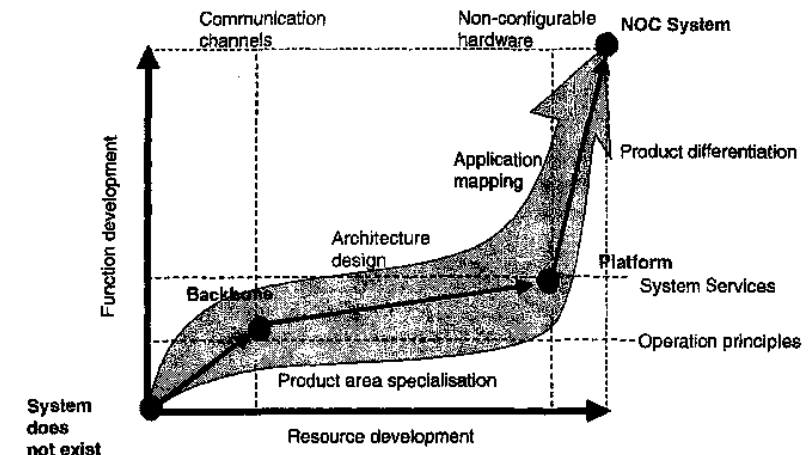


Figure 2-3. NOC design flow

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

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 Quality Press, 1993, 245 pp.
- [7] Wolf, W. Computers and Components - Principles of Embedded Computing: System Design, Morgan-Kaufman 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: Orthogonalization 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-

39

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

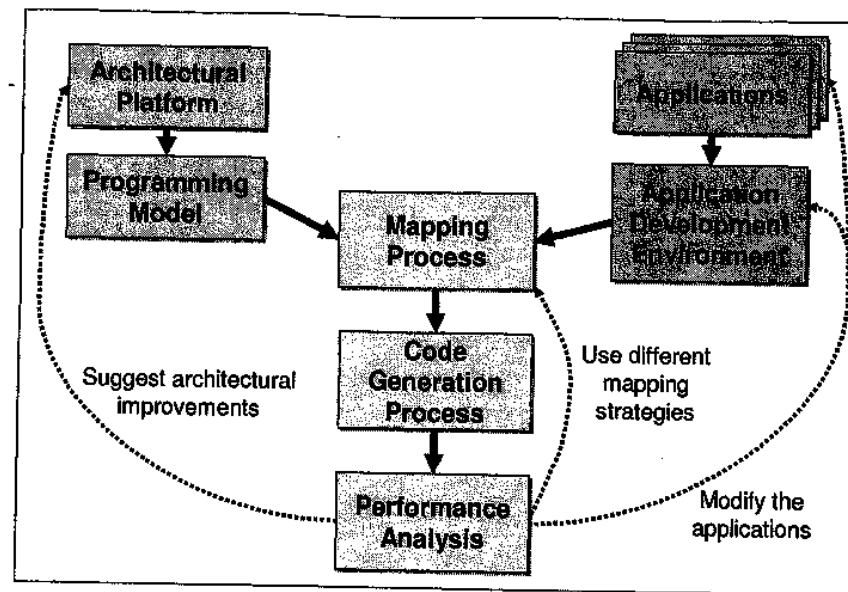


Figure 3.1. Y-chart Design Methodology

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.

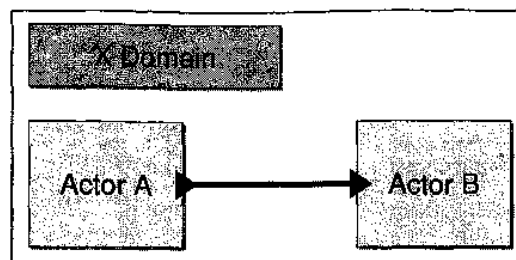


Figure 3.2. Simple Ptolemy Application

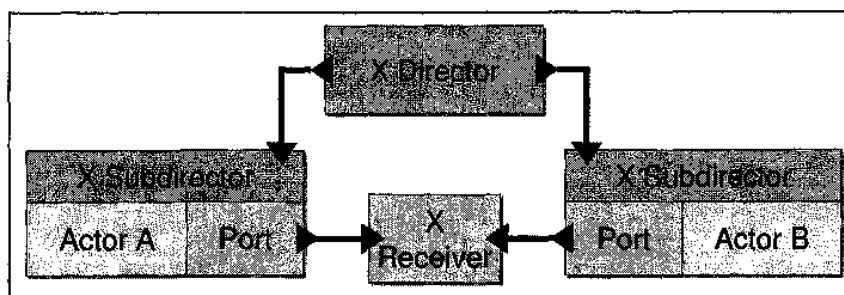


Figure 3.3. Communication Implementation View

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

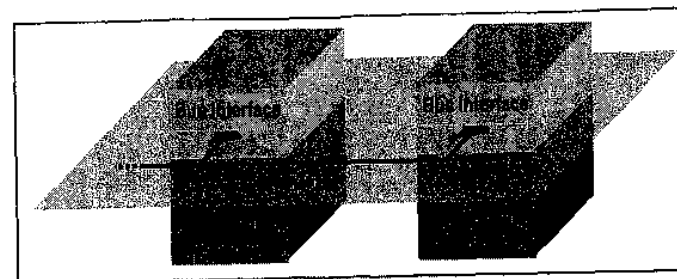


Figure 3.4. General Architecture

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

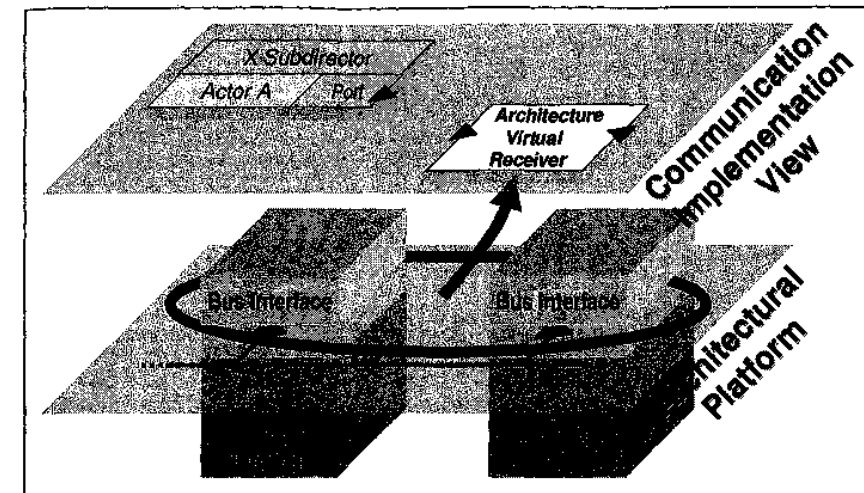


Figure 3.5. Exporting Virtual Receivers

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-

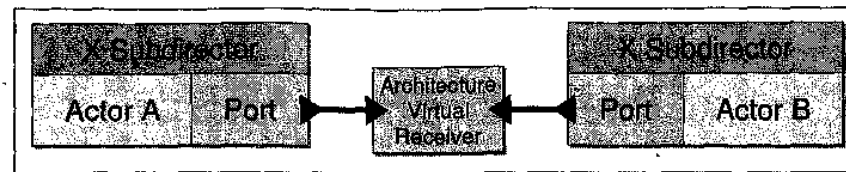


Figure 3.6. Natural Fit Mapping

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

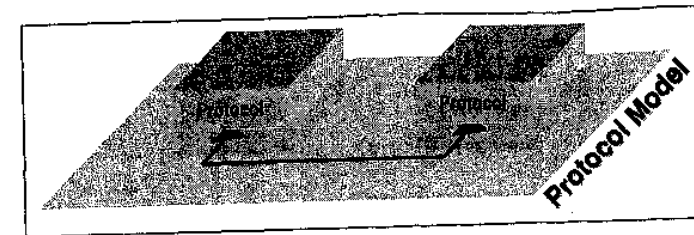


Figure 3.7. Protocol Model

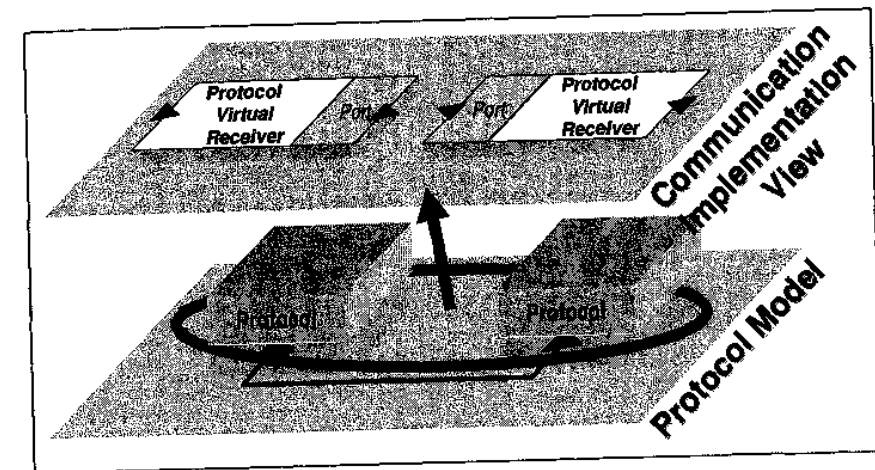


Figure 3.8. Exporting Protocols as Virtual Receivers

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-

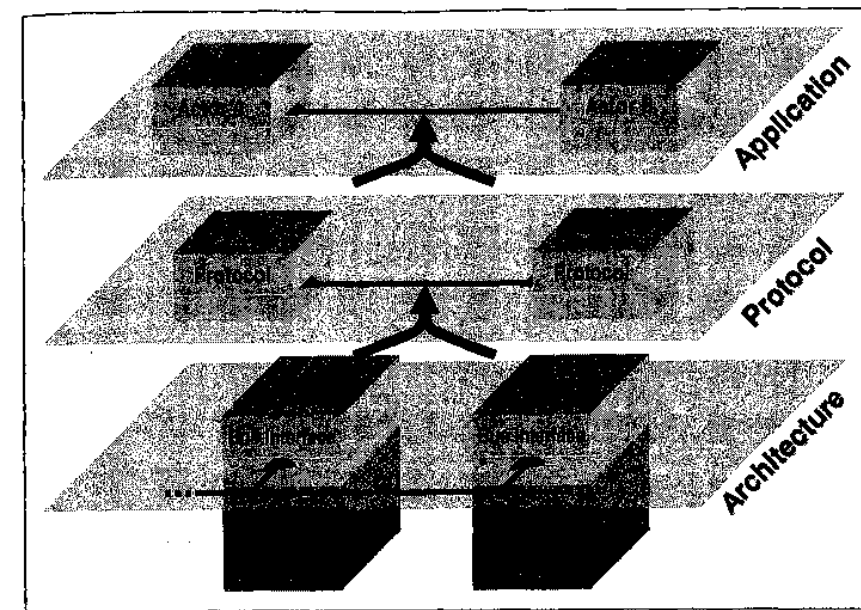


Figure 3.9. The Layers of Abstractions Behind a Protocol Stack

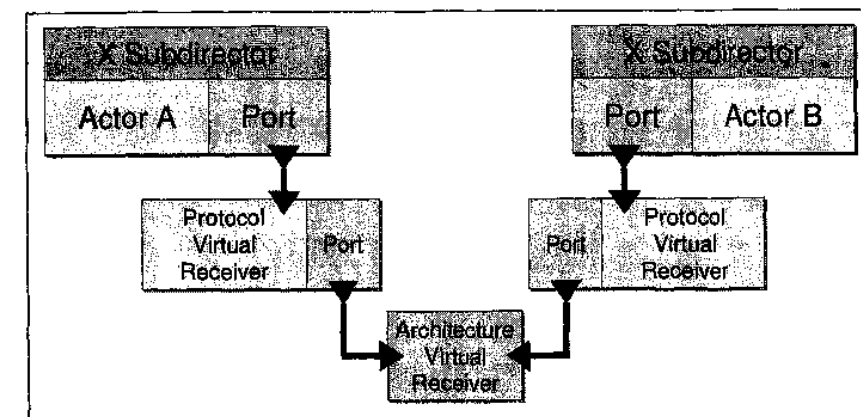


Figure 3.10. A Protocol Stack in the Communication Implementation View

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.

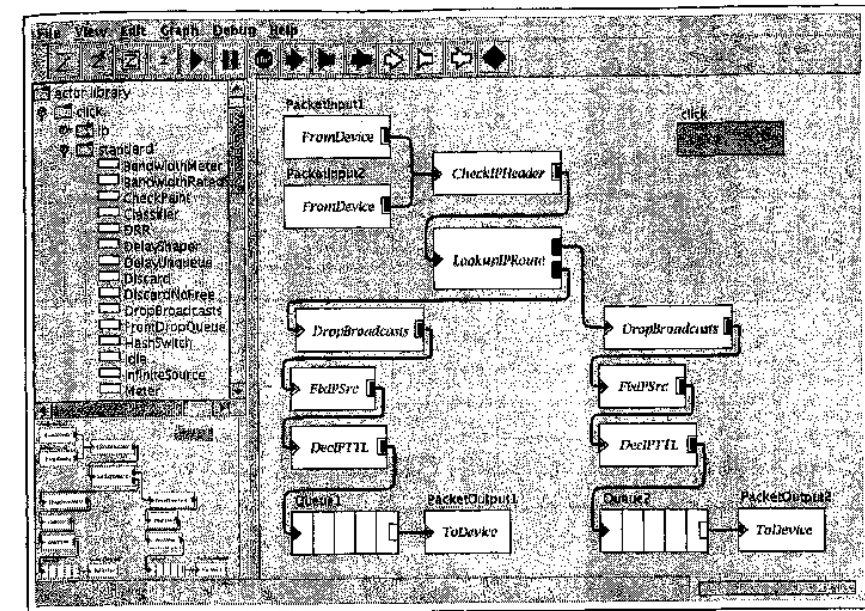


Figure 3.11. Simplified Click Router Application

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.

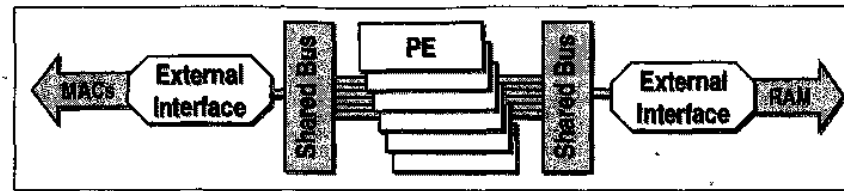


Figure 3.12. Target Architectural Platform

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

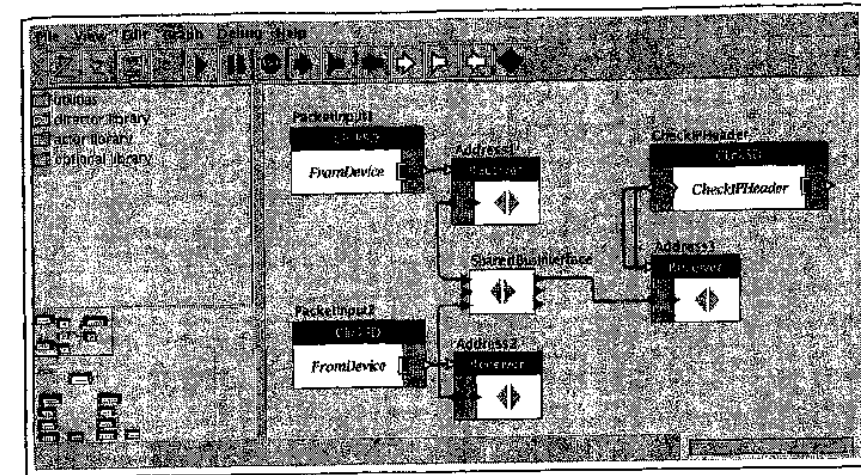


Figure 3.13. Portion of Click Router Communication Implementation View

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 Pro-*

- gramming*. 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.