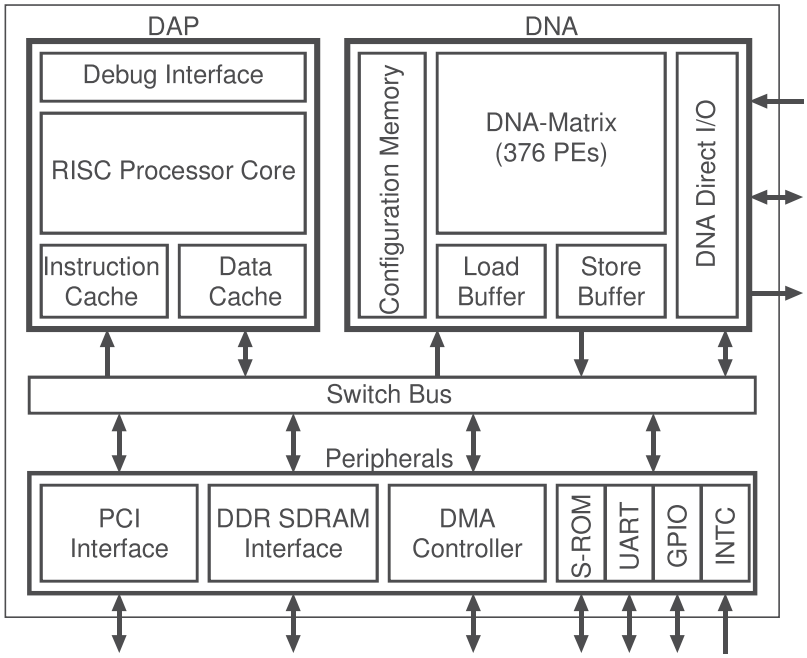


Fig. 3.17: DAPDNA-2 Architecture

Matrix provides capability for high speed data processing and scalability, with parallel computing and dynamic reconfigurability, respectively. This tightly coupled dual-core architecture realizes both flexibility and high performance.

The DNA-Matrix includes several types of processing elements as shown in Table 3.1. To map an algorithmic data flow graph directly onto the hardware, several types of PEs are placed in a two-dimensional PE array consisting of 8 x 8 PEs. This 8 x 8 unit is considered to be a segment, and the DNA-Matrix includes 6 segments, the total number of PEs being 376. There are two major categories of PEs. The first category is data processing where the combination of EXE/DLE/RAM has a complete capability to execute various operations on a data stream. The second is data I/O, and it is possible to generate data streams autonomously.

The DNA-Matrix has rich bus structure to realize various types of algorithms, and the DNA-Matrix Bus consists of a column-wise vertical bus which includes 8 x 32-bit wires corresponding to the row number. The DNA-Matrix Bus covers all rows in a segment, and the source PE can output its signal to all columns in a segment. Therefore, connections between arbitrary PEs are available, regardless of the relative position between source and destination PEs in a segment. The segment boundary connections are restricted slightly to realize a fixed operating frequency. The boundary PEs generate their output at the position of boundary PEs within the neighboring segment.

Configuration data of the DNA-Matrix is in two orders of magnitude smaller than fine-grained FPGAs. With this configuration size, the DAPDNA processor can hold entire DNA configuration data within context memories of each PEs. The DAPDNA-2 holds four contexts in the context memories: one is the foreground and three in the background. The proximity of configuration mem-

Table 3.1: Types of DAPDNA-2 PE

PE	Qty	Function
EXE	168	Arithmetic and logical Ops.
DLY	138	Variable delay for data sync.
RAM	32	Temporary memory
C16E	12	Address generator
C32E	12	Address generator
LDB	4	Data Input from processor bus
STB	4	Data Output to processor bus
LDX	4	Data Input from Direct I/O
STX	4	Data Output to Direct I/O

ory enables the DNA-Matrix to switch configuration in one clock cycle. The trigger of a dynamic reconfiguration event can be generated by two sources: the control signal from DAP core and reconfiguration signal from the DNA-Matrix itself. The total reconfiguration time using the DAP core is usually less than 1μs.

The result of performance evaluations shows the DAPDNA-2 performs more than 28,000 million instructions per second (MIPS) at a clock speed of 166MHz and is capable of more than 9,000 million MACs per second (MMACS) of multiply-accumulate operations with 16-bit inputs and a 32-bit output. And, the DAPDNA-2 can execute image filtering operations such as a Laplacian filter at 1 giga pixels per second. The same operation on a Pentium 4 (3.06 GHz) is topped at 16 mega pixels per seconds and this implies the DAPDNA-2 has 63-times performance gain.

In recent years, IPFlex continues to release enriched DAPDNA-2-based architectures. DAPDNA-3A [65] is based on multicore structure, and consists of 4 DAPs and 4 DNA-Matrices. Each DAP and DNA-Matrix is connected by a reconfigurable interconnection called AXION. Furthermore, the DNA-matrix in DAPDNA-IMS and DAPDNA-IMX [66] is enlarged to 955 16-bit PE array for the special purpose of image processing.

3.2.5 Hitachi FE-GA

Hitachi proposed Flexible Engine/Generic ALU Array (FE-GA) in 2005 [67]. The FE-GA is implemented using TSMC 90-nm Low Power CMOS technology, and it operates at 266MHz with power consumption of 200mW and with an area of 5mm^2.

The FE-GA consists of 24 ALU cells, 8 MLT cells, 10 load/store (LS) cells with a local RAM, a sequential manager (SEQM), a configuration manager (CFGM), a crossbar switch cell (XB), and a system bus interface as shown in Fig. 3.18. Each computing cell (ALU and MLT cells) is connected to its neighboring four cells. Therefore, the length of wires can be extremely short, making for a short delay that provides a high clock frequency. In addition, each of the LS cells, which are interface cells for local RAMs, can be connected to any computing cell on the utmost-left or utmost-right sides via XB connection.

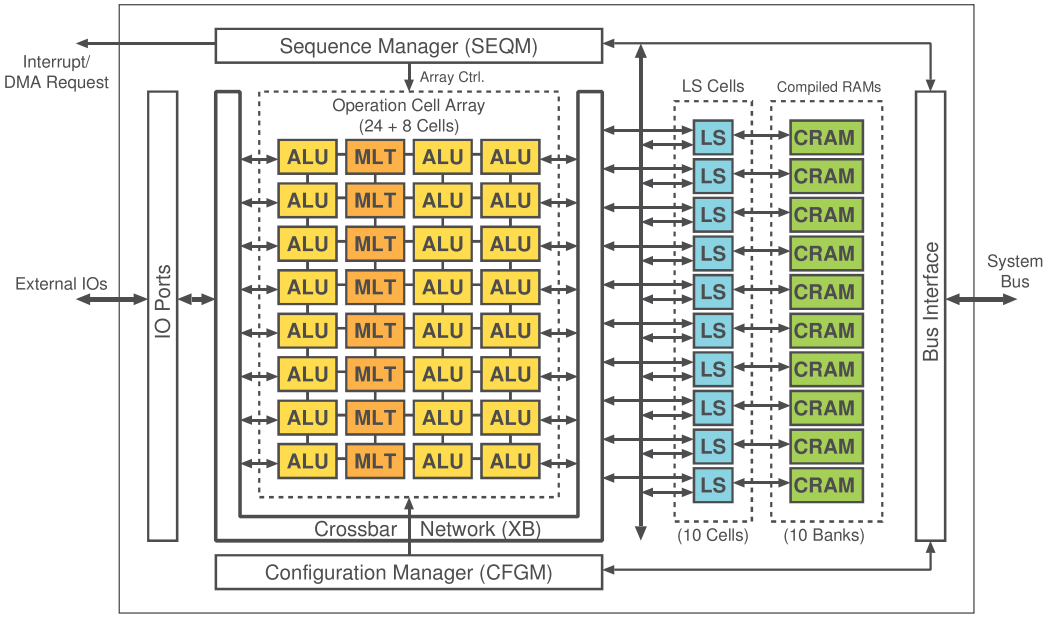


Fig. 3.18: FE-GA Architecture

The configuration data for each cell is managed by the CFGM and transferred to configuration registers (i.e., context memories) of each cell in the background. This configuration data is referenced by a pointer so that it can be compressed to an average of 20% its original size. The context switch of each cell is controlled by the SEQM, which generates a switching trigger using its built-in counters or the computation result of counters or comparisons of ALU cells. The SEQM can consider a conditional branch for the trigger, so the FE-GA can perform run-time context switches flexibly.

Fig. 3.19 shows an ALU architecture of the FE-GA. Each ALU cell consists of a general 16-bit ALU, a shifter (SFT), a data through logic circuit (THR), and an input delay logic circuit. All components of the ALU cells can be operated simultaneously. The data through logic circuit transfers the data from a cell's output to its neighboring cell's input without blocking any ALU operation. In addition, some ALU cells can send their result to the SEQM as a switching trigger that switches the context of all the cells by the result.

Each MLT cell consists of, as similar to a ALU cell, a general multiplier, a data through logic circuit, and an input delay logic circuit. All components of the MLT cells can be operated at the same time as ALU cells. Each LS cell shown in Fig. 3.20 consists of a port (A0/A1) connected to the XB, a port (B) connected to the bus interface and a port (0/1) connected to a dual-port RAM (local RAM). Because the LS cells can generate load and store addresses without using the ALU cells, the ALU cells can be saved for user applications.

Evaluation results based on TSMC 90-nm LP CMOS technology demonstrates that the operational frequency is 266MHz in worst cases, and the performance is 17.0 GOPS for 16-bit integer operations. The estimated area of the FE-GA is 5.0mm^2 including 40-KB local RAM, and power consumption is estimated as 210mW. In addition, several signal processing applications such as FIR

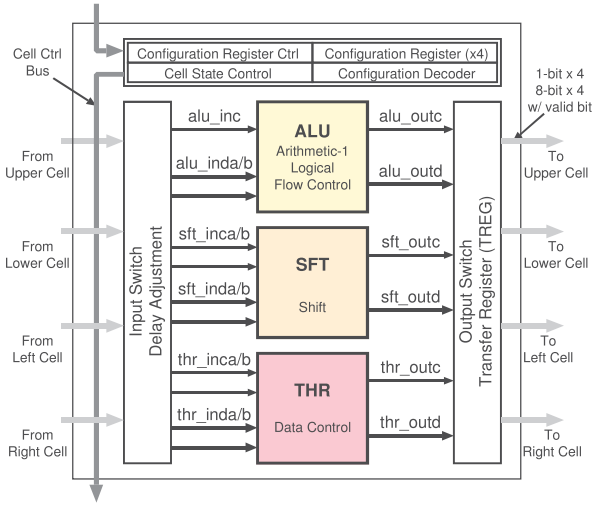


Fig. 3.19: FE-GA ALU Cell Architecture

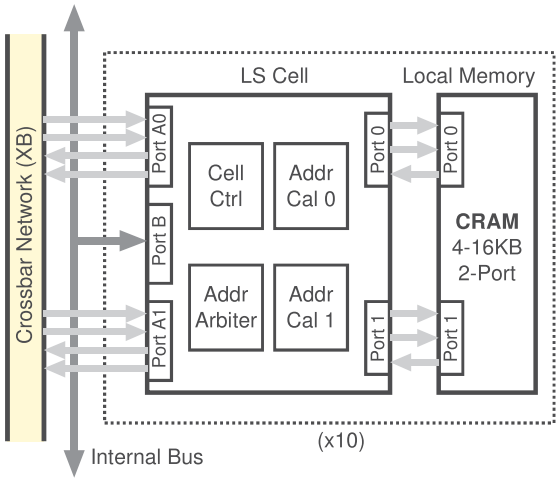


Fig. 3.20: FE-GA LS Cell Operation

filters and DCT have been implemented on the FE-GA, and it can perform at about 10 times fewer clock cycles than a DSP.

3.2.6 Elixent D-Fabrix

Elixent D-Fabrix[2] [68] reconfigurable processor is an evolution of the CHESS architecture [69] developed by Hewlett-Packard Laboratories.

The goal of the CHESS architecture was to increase both arithmetic computational density and the bandwidth and capacity of internal memories significantly beyond the capabilities of current FPGAs, while enhancing flexibility. Toward the goad, the fundamental computational unit is a 4-bit ALU with a primary set of 16 instructions. This provides efficient arithmetic capabilities suitable for cascading to useful media operations, or for supporting nibble-serial implementation styles. The entire user-visible routing structure is also based on 4-bit buses.

Fig. 3.21 shows the CHESS chessboard- or checkerboard-style reconfigurable arithmetic architecture. Each ALU is adjacent to four switchboxes, and each switchbox is adjacent to four ALUs. This allows very powerful local connectivity, with each ALU having input and output buses on all four sides, and being able to send data to or receive data from any of the eight surrounding ALUs as shown in Fig. 3.21 by the dotted arrows.

At run time, any switchbox in a CHESS array can be used as a 4-bit x 16-word memory. In this mode, all of the switches in the switchbox are disconnected, although the buses running over the switchbox can still be used. These memories are distributed through the array, attached to user plane wiring buses, and typically controlled by surrounding ALUs which generate an address and R/W control. The above design above provides one block RAM per 16 ALUs[3], using about 25% more

[2]The technologies of the D-Fabrix is now licensed by Matsushita Electric Industrial Co., Ltd.

[3]As like MeP-combined version described later, in recent D-Fabrix arrays, the basic block called Tile consists of 2 ALUs and 2 Swichboxes, and every 2 x 4 Tile has one 256 bytes block RAM.

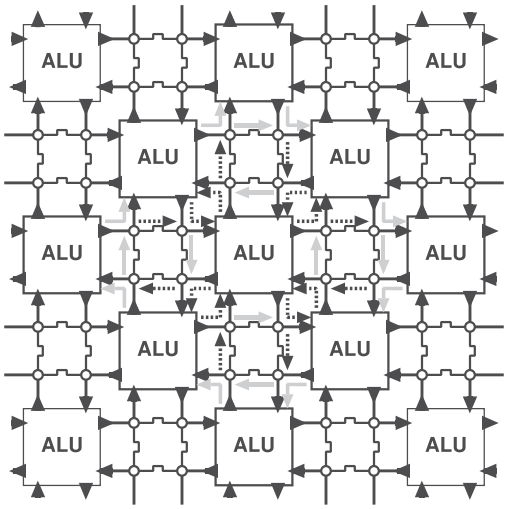


Fig. 3.21: Chessboard-style ALU Array

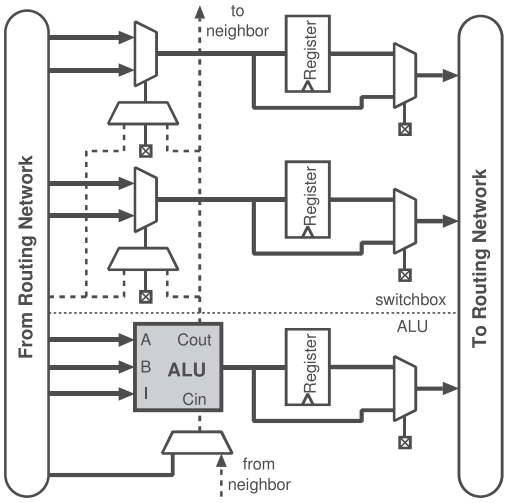


Fig. 3.22: D-Fabrix ALU and Switchbox

area than the basic array without the block RAMs.

Fig. 3.22 illustrates the ALU and Switchbox pair of Elixent D-Fabrix architecture. The architecture is almost the same as the CHESS architecture, for examples, 4-bit ALU array and chessboard-style interconnections. There is, however, one significant difference between the D-Fabrix and the CHESS. The multiplexers that select signals from the routing network to pass to the pipeline registers are statically configured in the CHESS, but the D-Fabrix allows them to be dynamically controlled, using either an ALU Cout or a signal from the routing network as a multiplexer control signal.

The area of the basic cell (ALU plus switchbox) of the CHESS architecture is 0.045 mm^2 in a process with 0.35μm feature size. The simulation results show that an operational frequency is 200MHz, and the CHESS architecture has a substantial performance advantage over some FPGAs in an equivalent semiconductor technology. In the D-Fabrix architecture, dynamic switchbox multiplexers have resulted in a slight overall increase in the area of a combined ALU and Switchbox. This increase is due to the addition of the circuits that select control inputs to switchbox multiplexers and their configuration memories. The area increase is no more than 10%. Compared to the ALU, the switchbox multiplexer is one-third of the size and it has half the propagation delay and 40% of the power consumption.

The D-Fabrix ALU array is actually integrated into Toshiba's Media Embedded Processor (MeP) [70] and Matsushita's UniPhier platform [71, 72] as a flexible hardware engine. Toshiba developed a reconfigurable LSI called ET1, in which the D-Fabrix works as an extension unit of MeP Core. Fig. 3.23 shows a block diagram of ET1. The D-Fabrix of the ET1 consists of 576 Tiles, which means 1152 ALUs, and 72 Block RAMs. In [70], it is demonstrated that the area-efficiency of D-Fabrix compared to an ASIC ranges from 1/2.8 to 1/16.3, and the maximum operational frequency ranges from 1/1.7 to 1/5.8.

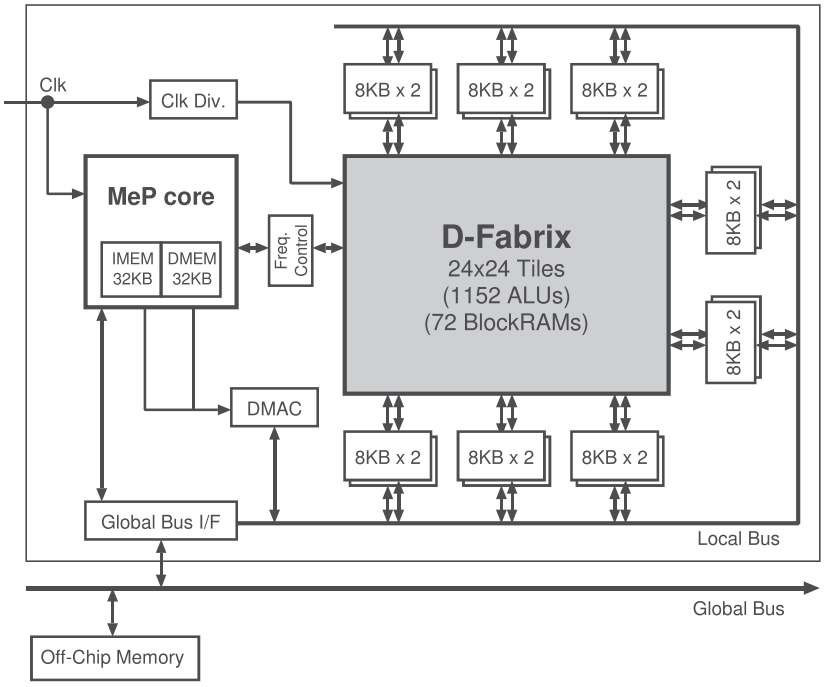


Fig. 3.23: ET1 Architecture with MeP core and D-Fabrix

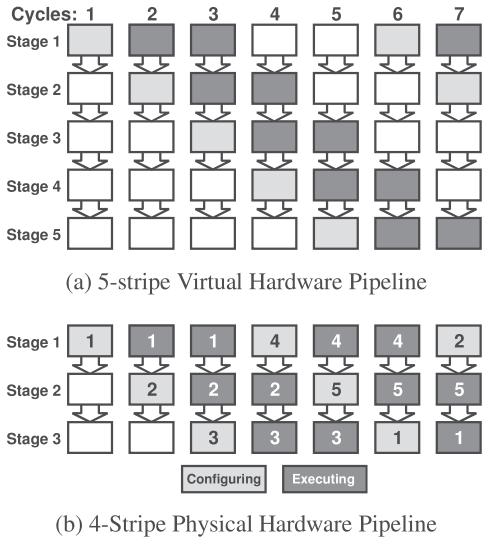
3.2.7 Rapport Kilocore

Rapport Kilocore [73] architecture is a commercialized reconfigurable processor derived from PipeRench [74], which has been proposed at Carnegie Mellon University in late 1990's.

PipeRench is based on an unique virtual hardware model called virtual pipelines described in Fig. 3.24. Fig. 3.24(a) shows a 5-stage (or stripe) virtual pipeline. Fig. 3.24(b) illustrates the first 7 cycles of reconfiguration of a 3-stage physical hardware pipeline executing the 5-stage virtual pipeline. Reconfiguration is performed by storing the configuration data of the entire virtual pipeline in an on-chip configuration data memory, and transferring them from the memory to the physical fabric every cycle.

Fig. 3.25 depicts a reconfigurable fabric of KC256, which is the first generation of the Kilocore architecture and virtual pipeline model. In Kilocore architecture, the functionality in each stripe consists of 16 8-bit PEs. All PEs within a stripe are interconnected within that stripe, which facilitates easier placement and routing of operations.

Fig. 3.26 is a block diagram of KC256's 8-bit PE. All selected inputs to multiplexers and shifters in this figure are connected to configuration bits stored in that PE. Special purpose interconnects are used to combine adjacent PEs to perform operations more than 8 bits in width. The shifters are also connected to the corresponding shifters in adjacent PEs to allow for efficient multi-PE shift operations. Each PE contains a register file with eight registers. The output from the functional unit can be written to any one register in the register file in that PE. If the value from the functional unit is not written to a register, the value from the corresponding register in the previous stripe is latched



(a) 5-stripe Virtual Hardware Pipeline

(b) 4-Stripe Physical Hardware Pipeline

Fig. 3.24: Virtual Pipeline Model

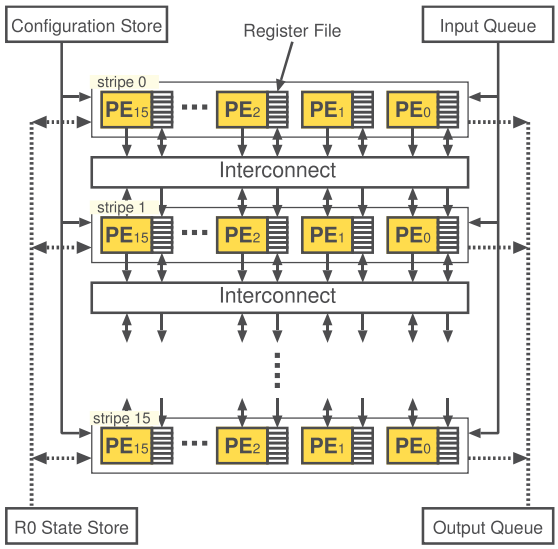


Fig. 3.25: Kilocore KC256 Architecture

into that register. The functional unit in each PE consists of eight 3-input LUTs that are identically configured. The functionality of the PE is specified by 42 configuration bits, which means that each stripe has 672 bits of configuration.

The process of configuring a stripe takes one cycle, so the pipeline can be configured one cycle before the first data of the pipeline arrives at that stage. If the virtual pipeline is larger than the real hardware, physical stripes will eventually be reconfigured with new virtual stripes. The state of the over-written virtual stripes are preserved by writing the value in R0 into the R0 state store memory. The state will be restored when the escaped virtual stripe is returned to the fabric. The process of reconfiguration continues until the last stripe in the virtual pipeline is configured. After that, reconfiguration of a physical stripe starting with the first virtual stripe will occur, and the computation proceeds with new inputs.

The KC256 is fabricated in a six-metal layer 0.18 μm CMOS technology. The total die area is 7.3 x 7.6 mm^2, and transistor count is 3.65 million. The fabric clock is designed to operate at 120 MHz under worst-case voltage and temperature conditions. At 120 MHz, the KC256 executes a 40-tap 16-bit FIR filter at 41.8 mega samples per second, which means that its performance is in the same range as Texas Instruments C64x family DSPs. The KC256 performs the IDEA encryption and decryption at 450 Mbps. By comparison, a 800-MHz Pentium-III processor executes it at a rate of 75.4 Mbps.

3.2.8 IMEC ADRES

IMEC ADRES [75] has two functional views as shown in Fig. 3.27: a VLIW view and an accelerator view as an array of reconfigurable cells (RCs). While the VLIW processor is optimized for control and load/store operations, the accelerator is optimized for data-parallel processings. In acceleration

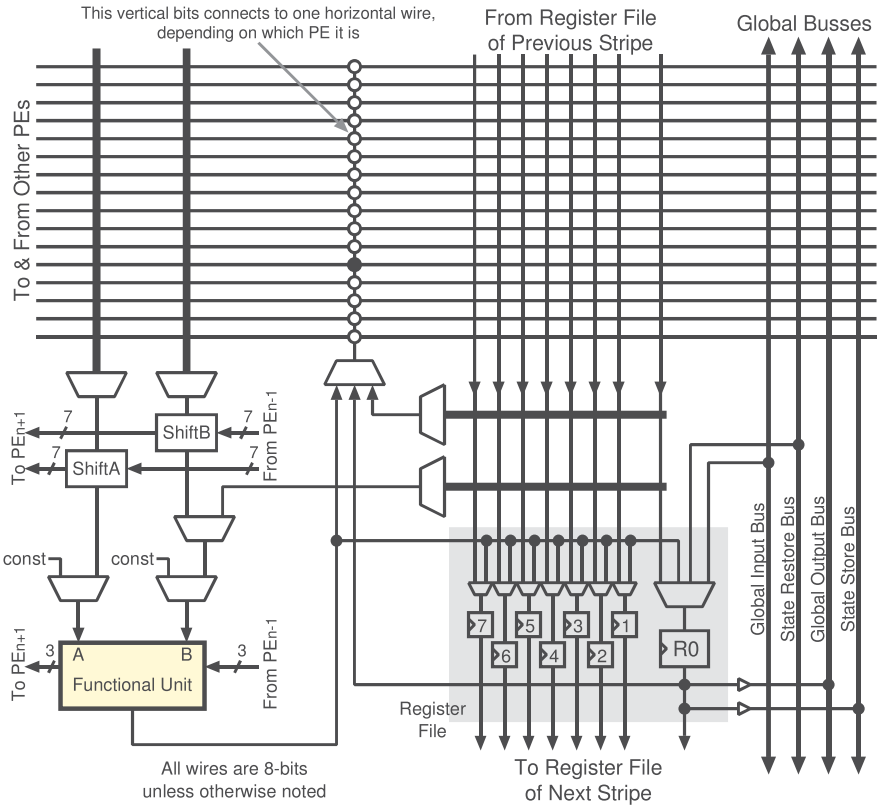


Fig. 3.26: Kilocore KC256 PE Architecture

mode, the functional units of the VLIW form the first row of the array. Orthogonal buses facilitate data transport within the array. The VLIW part includes up to eight functional units organized in a row. The units communicate with each other through a horizontal data bus for data exchange. A part of the units can communicate vertically with a common register file for data load and store.

The attached array is composed of several rows of RCs organized in a matrix form. The behavior of each RC is controlled by a locally stored set of several contexts. During the execution phase, a central controller allows dynamic reconfiguration of the RCs within a cycle. Each RC, as shown in Fig. 3.28, includes a local context memory (denoted by configuration RAM in the figure), an ALU, input and output multiplexers, and a register file for local data storage.

The data exchange between RCs is done with orthogonal interconnect-networks. There are two levels of interconnect for internal data exchange between the units: a global bus for each row or column spans the entire array. Additionally, the array is subdivided into four segments. Within a segment, a local interconnect is provided such that an RC can get input data directly from each of its horizontal or vertical neighbors.

The RC has been implemented with the Infineon Technologies CMOS 130nm 6 metals process technology. A context memory is designed with an 40-bit x 32-word SRAM. The schematics show that one RC requires 63266 transistor counts, and the total area is 0.196 mm^2. The area breakdown of the RC components is as follows: 50% for the context memory, 6% for external interfaces for

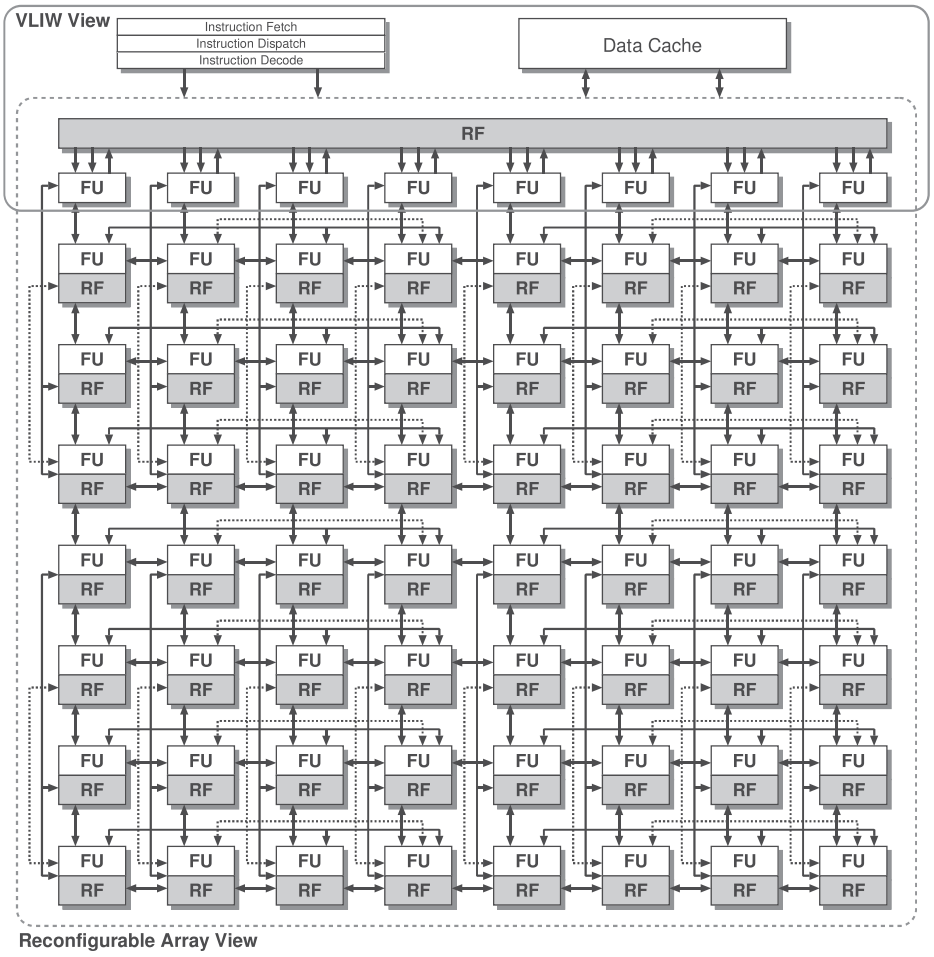


Fig. 3.27: ADRES Core Architecture

input and output, 9% for a register file, and 19% for an ALU. About 15% of the area of an RC are consumed by the interconnect between the components.

The maximum operational frequency of the RC is 60MHz or 16-ns cycle time. At a target frequency of 100 MHz, the power consumption for one RC is about 1.7mW. For performance evaluations, an H.264/AVC video decoder has been implemented on the ADRES architecture. The evaluation results show that the decoding cycle for the array view is about 88% faster than the VLIW view. The required clock frequency is 184MHz in the VLIW view and 98 MHz in the array view. The average power consumption at 100MHz would be around 46.3mW.

3.2.9 Stretch S5/S6 SCP Engine

The S6000 family configurable processors [76] are powered by Stretch S6 Software Configurable Processor (SCP) Engine. As shown in Fig. 3.29, they incorporate a Tensilica Xtensa LX dual issue VLIW processor core and a second-generation Instruction Set Extension Fabric (ISEF). The ISEF is a software-configurable computing fabric that contains 64KB of embedded RAM (IRAM). Using the ISEF, system designers can extend the processor instruction set and define new instructions using

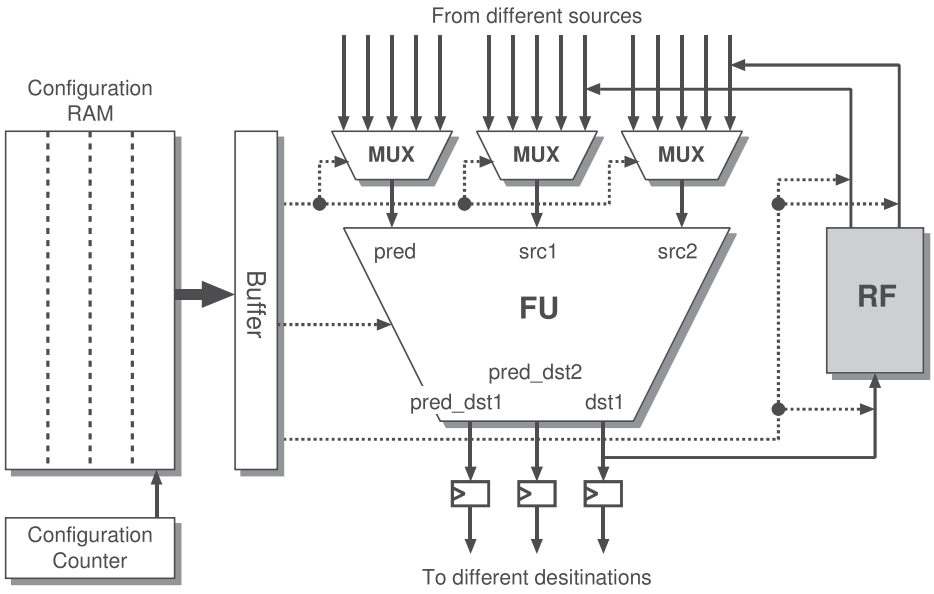


Fig. 3.28: ADRES Reconfigurable Cell

only their C/C++ code. As a result, developers get the performance of custom hardware with C/C++ development simplicity.

The S6 SCP Engine within the S6000 family, described in Fig. 3.30, contains a Xtensa LX VLIW core and an ISEF. It is the ISEF that provides the dramatic application acceleration by allowing user algorithms to be instantiated in hardware and called by the processor as single instructions. The Stretch ISEF, being tightly coupled to the processor, only needs to host compute-based and logic functions.

The S6000 ISEF contains 4096 ALU-based PEs that, in addition to traditional ALU functions, can be configured to perform 2 x 4 multipliers and cascaded for larger data width. In addition, there are 64 dedicated multipliers capable of 8 x 16 operations that can be cascaded to increase data width. Distributed state registers provide local storage for intermediate values and coefficients. Connectivity of the PEs is enhanced with distributed multiplexers, priority encoders, and shifters.

For computation-intensive applications, the S6 ISEF is fed by the same 32 128-bit wide registers carried over from previous generations of Stretch devices [77]. These registers are used for loading data into the ISEF, and their presence in the S6000 ensures maximum compatibility and code reuse from previous software configurable processor designs. The S6000 ISEF also contains 64KB of embedded ISEF RAM (IRAM) distributed throughout the fabric in 32 banks of 2KB each. The IRAM is the memory mapped into the S6SCP address space, so can be loaded directly by the processor.

The ISEF supports dynamic reconfiguration based on the configuration delivery scheme. If a fetched opcode corresponds to an extension instruction that is not resident in the ISEF, an instruction fault is raised. The operating system will then save the contents of any internal state registers, find the extension instruction group containing the missing instruction, and initiate an ISEF reconfiguration

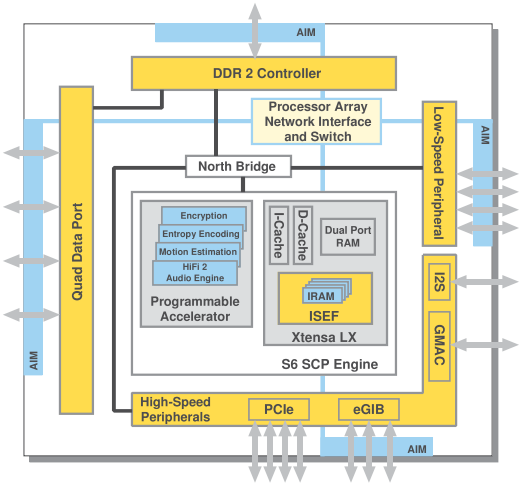


Fig. 3.29: S6000 Architecture

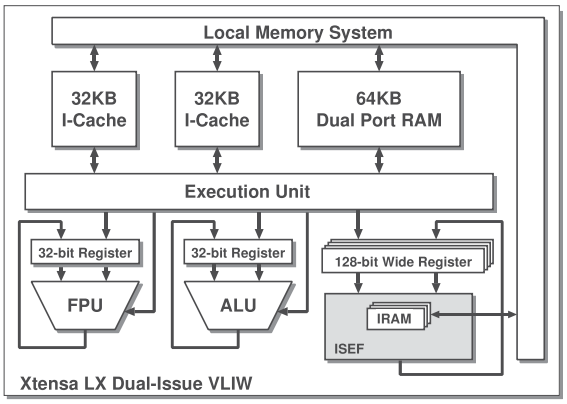


Fig. 3.30: S6 SCP Engine

before resuming the application program. The ISEF can be completely reconfigured in 27μs.

3.2.10 Fujitsu Cluster Architecture

Fujitsu has proposed a coarse-grained multicontext dynamically reconfigurable processors suitable for wireless communications [78]. The architecture consists of a hierarchical PE array called a cluster.

Fig. 3.31 shows a block diagram of the cluster. It contains several kinds of PEs, inter-PE networks, and a sequencer. To increase the cluster's area-efficiency, heterogeneous PEs are used in a cluster. The PE can be a 16-bit ALU, a MAC, a selector, a shifter, an address generator (counter), a one-port memory, a two-port memory, a register file, and variable delay line. The ALU supports logical, arithmetic, compare, and shift instructions, similar to DSPs. The "compare" instruction generates a signal that is used as a select signal by the selector and/or a transition signal by the sequencer. The MACs support 16 x 16 bit multiplication and accumulation.

The topology of inter-PE networks is a kind of indirect three-cube network and consists of three selector levels. The number of logical steps is the same in all paths. In the first implementation, the network corresponding to a 64 x 64 switch that consists of 4 x 4 switches. The sequencer controls the dynamic reconfiguration of the PE and inter-PE networks. The sequencer consists of a sequence control program memory, program counter (PC), and branch control unit. The value of the program counter is identical to the address of context memories in the PEs.

In this architecture, a hierarchical structure of the cluster as the cluster group is adopted to process larger algorithms that do not fit within a cluster. Fig. 3.32 illustrates the cluster group. Inter-cluster networks are organized through crossbars. The sequencer in each cluster also controls the context switch of each crossbar which has five direction inputs/outputs (from/to upper, lower, left, right, and into/out of the cluster). There are no restrictions on the data transfer direction between the clusters,

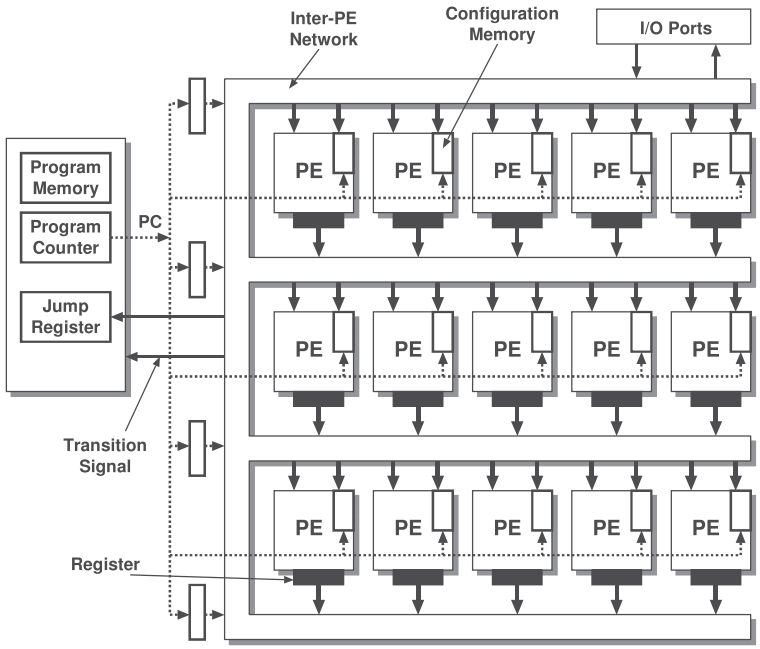


Fig. 3.31: Fujitsu Cluster Architecture

so the feedback paths that often appear in wireless communication baseband signal processing can be used. Each cluster can operate both independently and cooperatively in arbitrary combinations. Inter-cluster data transfer needs two or three additional cycles.

The shared resources located on the inter-cluster network are used by each cluster. There are two kinds of shared resources. One is a memory that can be shared by plural clusters. The other is dedicated hardware to accelerate signal processing.

The Fujitsu cluster architecture has been evaluated with algorithms that appear in the physical layer of IEEE802.11a and b wireless LAN processing. The evaluation results review that the cluster architecture provides a short latency compared to the DAPDNA-2 introduced in Section 3.2.4.

3.2.11 MuCCRA platform

The Multi-Core Configurable Reconfigurable Architecture (MuCCRA) project aims at developing a methodology and a framework to design highly configurable DRPAs toward varieties of target applications [79]. Because MuCCRA is developed for examining an optimized architecture for a given set of applications, several prototypes toward different needs have been designed and evaluated. Namely, prototypes introduced in this section are MuCCRA-1, MuCCRA-2 and MuCCRA-D.

3.2.11.1 MuCCRA-1

MuCCRA-1 is the first prototype chip of MUCCRA project adopting an island-style interconnection architecture for connect PEs [79, 39]. The MuCCRA-1 architecture shown on Fig. 3.33 consists of a 4×4 24-bit PE array, four 24-bit dedicated multipliers (MULTs) on the left side and four 24-bit

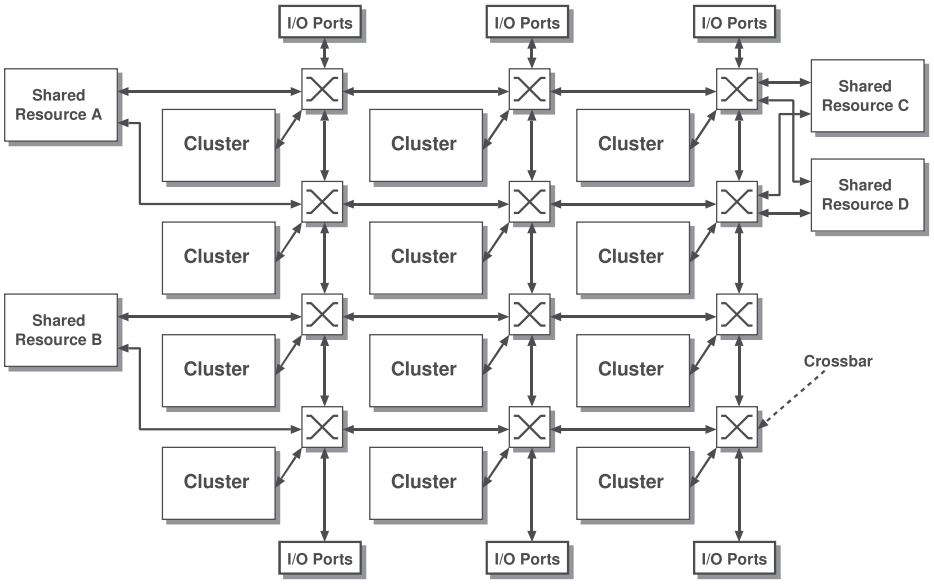


Fig. 3.32: Fujitsu Cluster Group

distributed memory modules (MEMs) at the bottom of the PE array. Each MULT unit can multiply two 24-bit data words. Each MEM is 24-bit $\times$ 256-word and 2-ported module.

A PE consists of a shift and mask unit (SMU), an arithmetic logic unit (ALU) and a register file unit (RFU) as shown on Fig. 3.34. A SMU supports various types of shift and mask operations; and, an ALU provides basic arithmetic operations, comparisons, and logical operations. Both a SMU and an ALU can perform half-word operations, in which a 24-bit datum is treated as two 12-bit data. A RFU is a 24-bit wide and 8-entry deep register file with a read/write port and a read-only port. For avoiding combinatorial loops, an output of the ALU can be connected to an input of the SMU, but the opposite connection is not allowed. On the other hand, each RFU can connect with all of inputs and outputs of the ALU and the SMU.

Each PE has a 64-entry context memory to provide multiple sets of configuration data for configuring SMA, ALU, RFU and interconnection. Being a multicontext device, context pointers are provided to every PE to control context switching. In every clock cycle, a configuration data pointed to by a context pointer is read out from the context memory. Doing this allows hardware context to be changed in a clock cycle.

MuCCRA-1 uses an island-style interconnection network. The input and output of each PE are connected to a near channel via a connection block. Switching Elements (SEs) are placed at the intersections of channels to route data to different directions according to configuration data provided.

3.2.11.2 MuCCRA-2

In order to reduce the size of the chip, MuCCRA-2 has the bit width and the context size smaller than correspondent parameters in MuCCRA-1 [79, 80]. Specifically, MuCCRA-2 uses a 16-bit architecture and allows only 16 contexts for context memories. In addition, instead of providing separated

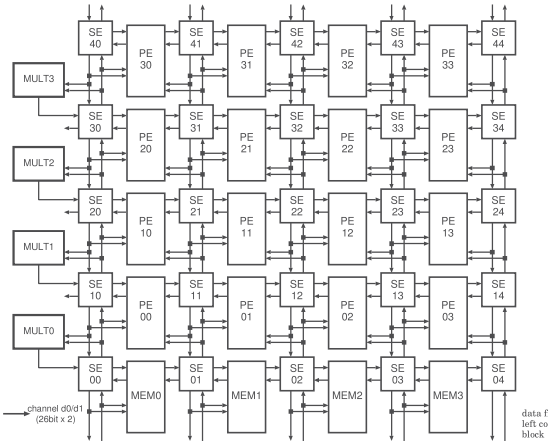


Fig. 3.33: MuCCRA-1 Architecture

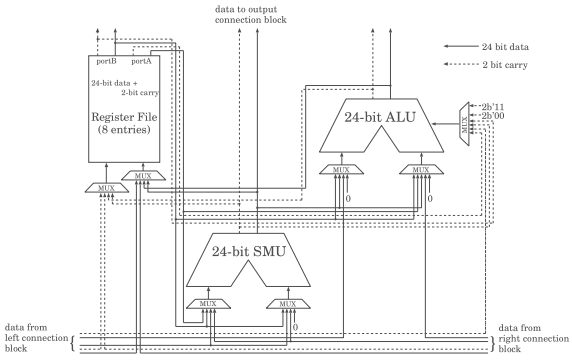


Fig. 3.34: PE architecture of MuCCRA-1

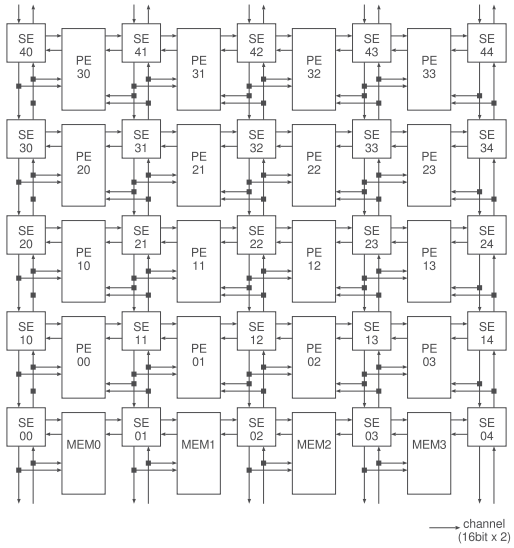


Fig. 3.35: MuCCRA-2 Architecture

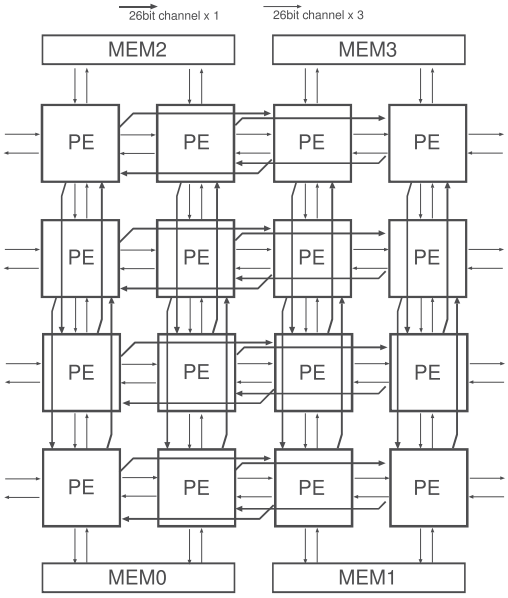


Fig. 3.36: MuCCRA-D Architecture

multipliers as in MuCCRA-1, multiply operations can be performed in every PE. Apart from that, the architecture of a PE in MuCCRA-2 is the same that in MuCCRA-1. The architecture of MuCCRA-2 is described in Fig. 3.35.

Similar to MuCCRA-1, an island-style interconnection network is exploited in MuCCRA-2 to connect PEs. However three routing channels are available on MuCCRA-2 instead of two routing channels on MuCCRA-1.

3.2.11.3 MuCCRA-D

Using an island-style interconnection network in the PE array of MuCCRA-1 and MuCCRA-2 causes two problems: the area overhead for routing wires and switches is large, and the operating frequency is reduced because of long delay caused by switches. To solve these issues, a direct interconnection

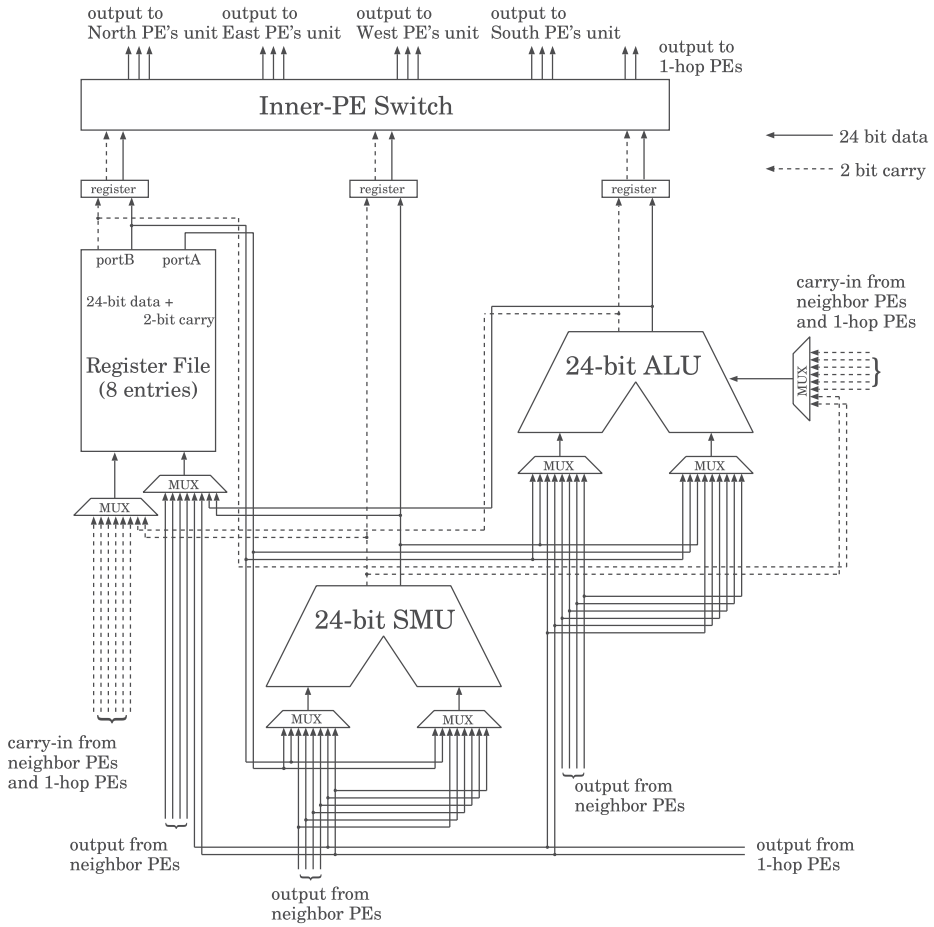


Fig. 3.37: PE architecture of MuCCRA-D

is applied in MuCCRA-D [39].

The architecture of MuCCRA-D is presented on Fig. 3.36. It composes of a 4 ×4 PE-array and 4 distributed memory modules on the top and bottom of the array. Distributed memory modules are the same as those in MuCCRA-1. Each PE is directly connected to the nearest neighboring PEs. Moreover, connections to the other PEs in the same row and column are also provided to reduce the delay for transferring data to remote PEs. Three independent channels are provided for the the nearest neighboring PEs while there is only one channel for one-hop distant PE.

Each PE of MuCCRA-D consists of a PE core, a 64-entry context memory and an inner-PE switch for selecting the destination of output data. Similar to MuCCRA-2, PEs in MuCCRA-D are equipped with multiply capability. The architecture of a PE is presented on Fig. 3.37.

3.3 Summary

Table 3.2 summarizes the characteristics of the coarse-grained dynamically reconfigurable architectures described in this chapter. As shown in this table, a wide variety of architectures have been

Table 3.2: Summary features of surveyed Dynamically Reconfigurable Processors

Number	Device	Vendor	Ref.	*G*	*PE*	*R*	*CX*	*I*
1	CS2112	Chameleon	[61]	16/32	108	M	8	Tiled, 2D Bus
2	XPP-III	PACT	[36]	16	64	C	-	2D Bus, Direct
3	DRP-1	NEC/NECEL	[63]	8	512	M	16	Tiled, 2D Bus
4	DAPDNA-2	IPFlex	[64]	32	376	M	4	Tiled, 2D Bus
5	DAPDNA-IMX	IPFlex	[66]	16	955	M	4	Tiled, 2D Bus
6	FE-GA	Hitachi	[67]	16	32	M	4	2D-mesh Direct
7	D-Fabrix	Elixent	[68]	4	576	C	-	Chessboard
8	KC256	Rapport	[73]	8	256	C	-	Crossbar for Rows
9	ADRES	IMEC	[75]	16	64	M	32	2D-mesh Direct
10	S6 SCP Engine	Stretch	[76]	4/8	4096	C	-	-
11	Cluster machine	Fujitsu	[78]	16	15/Cluster	M	-	3-Stage Switch
12	MuCCRA-1	Keio University	[79, 39]	24	16	M	64	Island style
13	MuCCRA-2	Keio University	[79, 80]	16	16	M	16	Island style
14	MuCCRA-D	Keio University	[39]	24	16	M	64	Direct

G = Granularity [bits],
PE = PE-Array Size (number of PEs),
R = Reconfiguration Method,
CX = Number of Contexts,
M = Multicontext Scheme,
C = Configuration Delivery Scheme,
I = Inter-PE Connection Network

proposed and designed, but they seem to be designed for a particular purpose such as digital signal processing devices, multimedia or wireless communications. The main reason is that the dynamically reconfigurable processors are expected to provide high area- and power-efficiency for the target application even with decreasing flexibility or programmability compared to FPGAs. Consequently, I can find various kinds of architectures, and the straightforward architectural classification has become complicated [14].

Dynamically reconfigurable processors have a very wide design space including PE structures, channel bit widths, intra/inter-PE connection networks, the number of available multipliers and distributed memory modules, dynamic reconfiguration schemes, context memory capacity, PE array sizes, and so on. Thus, unlike conventional FPGAs [3, 81], it is difficult to explore a general-purpose and preferable architecture among the dynamically reconfigurable processors. Although they are just being in practical use, designers need to perform a time-consuming and inefficient architectural exploration for its target application at design time. The configurable processors, in which several optimizations or customizations including instruction set extensions and memory size scaling can be performed, are fully exploited for embedded microprocessor designs. Also for dynamically reconfigurable processors, a configurable architecture design methodology is needed in order to provide an efficient design space exploration to SoC designers.

The variety of architectures for DRPAs also leads to a challenge in developing computer-aided designs and compilation tools that can map an application to a target reconfigurable device. This involves determining which parts of the application should be mapped to the reconfigurable array and which should be mapped to the attached processor, determining when and how often the reconfigurable device should be reconfigured, which changes the functional units implemented in hardware,

as well as the specification of algorithms for efficient mappings to the reconfigurable system. To date, a standard interface to DRPAs has not yet been solved, and each company still has its own set of tools tailored to its device.

Chapter 4

Hardware Task Mapping

4.1 Problem

To date, a large number of researches in the area of reconfigurable computing have resulted in a number of academic and commercial DRPAs. These devices play an important role in balancing high performance demands and low power consumptions, especially in embedded devices. One of the trends in developing reconfigurable devices is the dynamic reconfigurability based on a multicontext mechanism such as DRP, DAPDNA-2, FE-GA and ADRES introduced in Chapter 3 in order to minimize the reconfiguration overhead and greatly improve the performance of reconfigurable systems. The datapath mapped to a piece of physical hardware is called a *context*. A target application is divided into a set of different contexts, and a multicontext DRPA executes them by changing contexts with each clock cycle. Basically, with such multicontext DRPAs, an application is designed and mapped into hardware as a single thread of control. At any time, only one required context is activated and executed. In order to increase the throughput, some techniques such as software pipelining and loop unrolling can be applied to exploit more parallelism.

For a particular hardware architecture, the performance improvement depends on the inherent parallelism of a target application. According to Hasegawa [82], the optimal size of the PE array in a DRPA is fixed related to the target application, and PEs exceeding the optimal number are not efficiently used. In many cases, the parallelism of an application is smaller than the number of PE array in a context. Accordingly, a large number of PEs is not used efficiently.

One of the methods to improve the performance in such a condition is making the best use of stream-level pipelined execution. Most target applications of such devices are stream processing, that is, data blocks to be processed are iteratively received in a certain interval. By dividing an application into small independent sequential tasks, and executing in a pipelined manner, a large number of PEs can be used efficiently.

Beside the popular *single-process* execution, where an application is designed and implemented as the only one thread of control for being assigned and executed on a reconfigurable array, some architectures like DRP-1 [63] by NEC Electronics support the "multi-process execution", which

allows multiple threads of control to run concurrently. An application is divided into several *tasks*; a large reconfigurable array is partitioned into some small arrays, each of which is called Tile; and each *task* is assigned to different Tiles of DRPAs, and executed in parallel. An inter-task communication mechanism using internal memories is defined for exchanging data between tasks. Each Tile, to which a task is assigned, is independently controlled for executing and exchanging configuration data.

Although the introduction of the *multi-process* execution may lead to an effective way of partitioning applications in order to improve throughput and energy efficiency, systematic method for efficiently mapping tasks into hardware execution units has not been well researched. This chapter proposes a systematic mapping method to map target *tasks* into Tiles, and shows examples applying it for DRP-1.

4.2 Related Work

There are a lot of research efforts aimed at models of computation such as Synchronous Dataflow (SDF) [83], Dataflow Process Networks [84], and Kahn Process Networks [85]. In the Ptolemy project [86], several models have been combined together to create a structure for the Ptolemy framework. However, this framework mainly focuses on application modeling and simulation without supporting the real mapping of application models onto models of architectures.

Partitioning applications is a well-known technique that has been thoroughly researched. Hardware/Software partitioning specifies which parts of an application should be mapped to hardware or software components [87]. Partitioning applications between reconfigurable hardware blocks of different granularity tries to map some parts of an application into fine-grain reconfigurable units and others into coarse-grain reconfigurable arrays for exploiting the advantages of various granularity reconfigurable modules [88]. In a multiprocessor environment, partitioning applications into multiple tasks and threads, each of which could be mapped to an execution unit for running, is a well researched topic.

There are also a number of researches for job mapping and scheduling into the partitioned area of FPGAs with partial reconfigurable capabilities. In these studies, the area and execution time requiring for a job are fixed, and scheduling algorithms decide the order and place where multiple jobs should be mapped. Since a three-dimensional (x, y, and temporal) placement problem must be solved for efficient placement, a number of theoretical researches have been done [89] [90] [91].

However, my target problem has the following differences, which prevent previous researches from being directly applied.

- In the *multi-process* execution treated here, each task being a part of a single job works in a pipelined manner, and handles a large number of streaming data arriving continuously. Thus, the throughput of a single job with multiple tasks is a target for optimization.

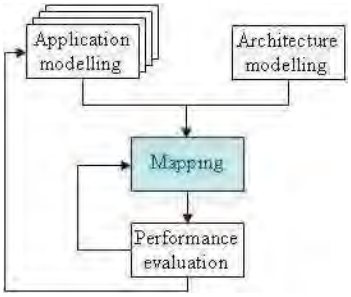


Fig. 4.1: General design flow

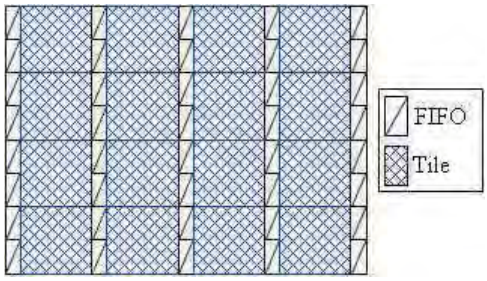


Fig. 4.2: Target DRPA

- By a multicontext execution, a Tile can execute multiple *tasks* sequentially as far as the number of contexts is sufficient.
- Multiple Tiles can be assigned to a *single task* for enhancing the execution speed by using parallel execution with a large number of PEs.

4.3 Target Architecture and Application Model

Fig. 4.1 shows a general design flow for a target DRPA. By explicitly specifying a mapping step, multiple target applications could be mapped onto a candidate architecture for evaluation. Typically, a set of applications is proposed for a certain architecture, then, they are mapped onto such an architecture. Through the performance analysis of each application-architecture-mapping combination, some improvements could be made for application implementation or mapping algorithm.

This paper will focus on a method for efficiently mapping an application modeled as a Kahn Process Network onto shown below target reconfigurable architectures.

4.3.1 Target architecture

In this paper, a two-dimensional tile-based DRPA consisting of identical $M \times N$ hardware execution units is assumed to be the target architecture. This type of target architectures can be considered as a homogeneous structure, which allows more flexible in mapping.

As an example, a target DRPA (4×4) shown in Fig. 4.2 consists of a number of PE array units, each of which is called a Tile. A Tile has a certain size, distributed shared memory modules and its own controller for dynamic reconfiguration. The total number of Tiles in the target DRPA is referred as N_{all}. Here, a multicontext DRPA is assumed, that is, each Tile has a certain size of context memory and changes its hardware context according to a context pointer. The number of contexts that can be stored in the context memory (C_{max}) is limited in a certain number. Some examples of a candidate DRPA with correspondent parameters can be seen in Table 4.1, where PEs and $Tiles$ show the total number of PEs and Tiles in each DRPA respectively; $PEs/Tile$ denotes the number of PEs per a Tile; and $Contexts$ is the maximum number of contexts that can be stored in a PE.

Table 4.1: Example of possible target architectures

Parameter	DAPDNA-2 [92]	ADRES [75]	DRP-1 [63]
PEs	376	64	512
Tiles	6	4	8
PEs/Tile	56	16	64
Contexts	4	32	16

Multiple neighboring Tiles can be joined to form a Tile group for executing a single task together. In this case, joined Tiles work synchronized with a single controller, and the number of available PEs becomes the sum of Tiles in the Tile group. Neighboring Tile groups can be communicated with each other through FIFOs formed with the distributed memory modules. That is, streaming data can be transferred between Tiles with a simple handshake mechanism.

4.3.2 Target application model

I assume that a target application can be represented with a Kahn Process Network (KPN), which is similar to a model proposed for streaming processors [93]. In this model, a total job is represented with multiple processes tasks that can be executed in a pipelined manner. That is, data streams continuously arrive at a certain interval, and the results of a task are transferred to adjacent tasks. A KPN has following characteristics:

- The execution of a KPN is deterministic, or in other words, for a given input, the same output is always produced. This model is suitable for stream-based multimedia and signal processing applications since it allows to model such types of applications, and guarantees that no data is lost.
- According to the model, tasks are monotonic, that is, they require only a partial information from input for generating a partial result to output. Therefore, the model allows parallelism and pipelined processing, which are suitable to be implemented on reconfigurable arrays.
- The model makes it easier to partition an application into a set of parallel communicating tasks.

For example, the upper part of Fig. 4.3 shows the KPN graph of a JPEG encoder. In this case, the graph becomes a linear structure. Media processing programs could be easily translated into KPNs [93], and here, I assume that the KPN corresponding to a target application has been already formed.

4.3.3 Goal of mapping

Each task of a target KPN (p_i) can be mapped into a Tile Group TG_j of the target DRPA, and executed in a pipelined manner. The lower part of Fig. 4.3 shows an example of mapping for a JPEG

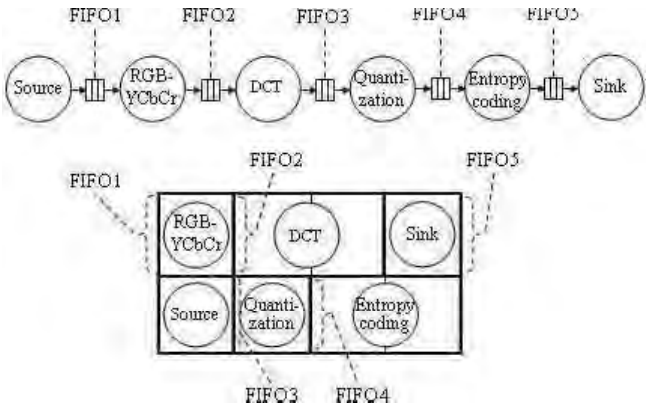


Fig. 4.3: Task mapping for JPEG encoder

encoder, in which, tasks mapped into each Tile Group get data stream from the input FIFO, execute their own computation, and produce results to the output FIFO. If there is no data in the input FIFO or the output FIFO is full, the task execution is stalled. The data stream is assumed to arrive in a certain interval corresponding to the total throughput of the DRPA.

In order to improve the total throughput and the execution time of an application, it is critical to balance computation stages in the interrelation with other tasks in a pipelined chain. In a pipelined processing model, the total throughput is bottlenecked with the most time consuming task. Here, by increasing the size of TG_js, the throughput can be enhanced by parallel processing with more number of processing elements. If, for example, the task *DCT* is the bottleneck of the JPEG encoder in Fig. 4.3, the total throughput can be improved by mapping *DCT* into a *TG* with a large size because the number of execution clock cycles of the bottleneck task (DCT) could be reduced.

The goal of mapping is to find the best combination of tasks and Tile Groups in order to improve the throughput and to reduce the execution time of each pipeline stage while preserving system limitations: (1) the total number of Tiles used in TG_j must be smaller than or equal to the number of Tiles supported in the target DRPA, and (2) the sum of contexts required for tasks mapped in a *TG* must be smaller than or equal to the number of contexts supported in the target DRPA.

4.4 Mapping Algorithm

4.4.1 Target task graphs

In this research, target task graphs are limited in a simple unidirectional linear graph with a fork-join structure. As shown in Fig. 4.4, a task can send a data stream to multiple tasks (fork) and the results are gathered in the next task (join). Each task is connected with a FIFO, and can work independently. Stream data arrive in a certain interval to the starting task, and the total tasks can be executed in a pipelined manner. Although complicated graphs cannot be represented because of the above limitations, most graphs of streaming processing are rather simple and fall into this limitation.

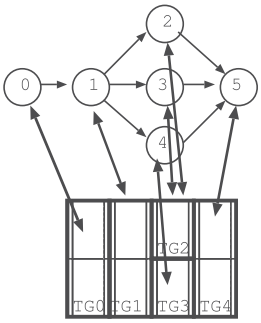


Fig. 4.4: Target task graph

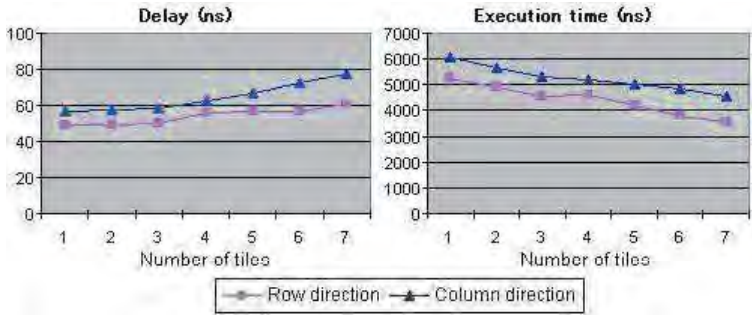


Fig. 4.5: Delay and execution time vs. number of Tiles

Here, task number p_i is assigned into each task from the starting task to the terminal one. Parallel execution tasks (tasks 2, 3, and 4 in Fig. 4.4) can be assigned in any order.

4.4.2 Target architecture and task mapping

With the target architecture introduced in Section 4.3.1, neighboring Tiles can form a Tile Group TG_j, and neighboring TGs can communicate via FIFOs provided on the edge of a Tile. The number of Tiles in a TG_j is denoted with $Size(TG_j)$.

When a task p_i is mapped on a TG_j, different assignment strategies could affect the quality of an implementation. Namely, mapping depends on three following factors: 1) The number of Tiles allocated to each task. 2) Absolute position of Tiles. 3) And, absolute position of FIFOs for exchanging data between tasks. Of which, the most important factor is the number of Tiles assigned to a task. In general, the execution time represented with $FT(p_i, Size(TG_j))$ is reduced when the task is executed with TG formed by a large number of Tiles. However, the relationship between the execution time and the number of Tiles is not simple. For example, Fig. 4.5 shows the relationship between the number of execution clock cycles and the delay with various numbers of Tiles when two main computation steps of two dimensional Discrete Cosine Transform (DCT), "Row direction" and "Column direction", are separately implemented on NEC Electronics DRP-1. While the number of execution clock cycles is decreased with a large number of Tiles, the delay tends to be stretched. So, the total execution time is reduced but not relational to the size of TGs. Moreover, if a task does not have enough degree of parallelism, the execution time is not improved at all by using Tiles larger than a certain size.

Fortunately, by using design tools like Musketeer for DRP-1 [59], the relationship between the size of TGs and the execution time can be evaluated in advance. Here, I assume that the execution time with the various size of TG has been evaluated and the result represented with $FT(p_i, Size(TG_j))$ can be obtained by table reference. Exactly speaking, $FT(p_i, Size(TG_j))$ depends not only on the size but also on the connected shape of Tiles. For example, when four Tiles are connected, the execution time slightly differs depending on the patterns shown in Fig. 4.6. However, implementation experience on DRP-1 shows that the difference is small when the number of

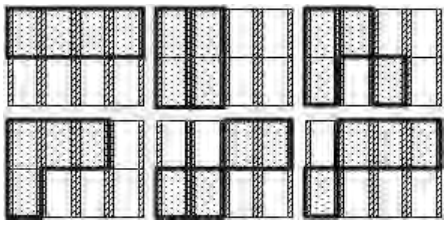


Fig. 4.6: Tile connection patterns for a TG

connected Tiles are less than eight. So, I will ignore the difference here.

When multiple tasks $p_i \ldots p_{i+k}$ are assigned into a TG_j, they are executed sequentially, (for example, task 2 and 3 mapped in TG2 in Fig. 4.4), that is, the total execution time T_{ik} becomes $T_{ik} = \sum_i^{i+k} FT(p_i, Size(TG_j))$ $(k > 0)$

Also, the number of contexts required for executing a task is depending on the number of Tiles in TG_j, and represented with $FC(p_i, Size(TG_j))$. It can be also analyzed before scheduling by the design tools. When multiple tasks are assigned into a TG_j, total required contexts becomes $C_{ik} = \sum_i^{i+k} FC(p_i, Size(TG_j))$

Here, a task fork can be included in the graph (e.g. tasks 2, 3 and 4 in Fig. 4.4). Such forked tasks can be mapped into the same TG even if they have non-neighboring numbers (e.g. tasks 2 and 4 in Fig. 4.4). However, if the graph does not have any fork, mapping tasks with non-neighboring numbers will increase the communication between two tasks, and this often requires the communication between non-neighboring Tiles. So, in order to make the algorithm simple, I only map the neighboring tasks into a TG.

4.4.3 Mapping algorithm

The proposed algorithm consists of three steps: task grouping, adjusting and topological mapping.

4.4.3.1 Task grouping

Task grouping clusters the tasks of an implementation in the specification model to n groups in a way that the computation amount in each group is balanced. Neighboring numbered tasks are mapped into a Tile as possible, keeping the limitation represented with $\sum_i^{i+k} FC(p_i, 1) < C_{max}$ for each Tile. Here, the number of mapped Tiles is called N_{used} and so the number of unused Tiles is $N_{all} - N_{used}$. When there are multiple candidates for mapping, the one with less N_{used} is selected. If $N_{used} > N_{all}$, the total job is too large to implement on the target DRPA. In this case, some of light-weight tasks must be moved to the software executed on the embedded CPU with the DRPA. In the initial mapping, every mapped Tile Group TG_j has only a Tile.

4.4.3.2 Adjusting

This step aims at adjusting the size of Tile groups assigned in the previous step toward reducing the execution time of those taking a large time. First of all, the execution time of each TG_j, $\sum_i^{i+k} FT(p_i, Size(TG_j))$, is evaluated in order to find TG_js whose execution time is the largest. To improve the execution time of such TG_j, unused n Tiles are assigned. Here, the limitation of the target DRPAs is considered. If the size of TG must be 1 or even numbers (1, 2, 4, 6 and 8), the assigned number of Tiles must be the smallest one keeping the limitation.

There are three possibilities:

- All tasks in TG_j are executed sequentially with a larger size of Tile $Size(TG_j)+n$. In this case, the execution time is $\sum_i^{i+k} FT(p_i, Size(TG_j)+n)$ if the execution time is smaller than those of the other two possibilities, add n Tiles to TG_j to increase the size.

- The first x tasks $p_i, \ldots p_{i+x-1}$ are executed with additional n Tiles and the other tasks are executed with TG_j. In this case, the execution time is $FT(p_i, n)+\cdots+FT(p_{i+x-1}, n)+FT(p_{i+x}+Size(TG_j))+\cdots+FT(p_{i+k}, Size(TG_j))$. If the execution time is smaller than those of the other two possibilities, generate a new Tile Group and move $p_i, \ldots p_{i+x-1}$ from TG_j.

- The last x tasks $p_{i+k-x}, \ldots p_{i+k}$ are executed with separated n Tiles and the other tasks are executed with TG_j. In this case, the execution time is $FT(p_i, Size(TG_j))+\cdots+FT(p_{i+k-x}, Size(TG_j))+FT(p_{i+k-x+1}, n)+\cdots+FT(p_{i+k}, n)$. If the execution time is smaller than those of the other two possibilities, generate new Tile Group and move $p_{i+k-x}, \ldots, p_{i+k}$ from TG_j.

After that, increase N_{used} and iterate the above steps until all Tiles are used; that is N_{used} is equal to N_{all}.

4.4.3.3 Topological mapping

With the previous described steps, tasks are mapped into TG_j, and the last step is to fit TG_j into the physical shape of an $M \times N$ structure. Two approaches can be applied in this step.

All possible mapping exploration (APME) Since the number of Tiles in a system is limited into small numbers (for example, eight in DRP as seen in Table 4.1), choosing the best topological mapping by this approach is possible in a reasonable amount of time by searching the complete solution space to retrieve all possible mapping variants.

In order to limit the search space, I decide the best allocation of Tiles for each list ($Size(TG_0)$, $Size(TG_1), \ldots, Size(TG_k)$), which consists of the sizes of TGs after adjusting where k denotes the number of used TGs. There are multiple possibilities of Tile assignment for each list. For example, the allocations in Fig. 4.7 are all corresponding to the list (2,1,1,2,2), since the arbitrary combination of Tiles can be allowed in DRP-1. However, allocating a TG into separating Tiles increases long wires

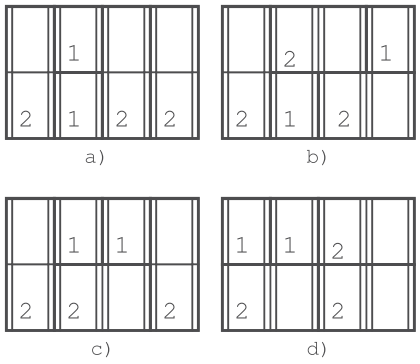


Fig. 4.7: Tile assignment

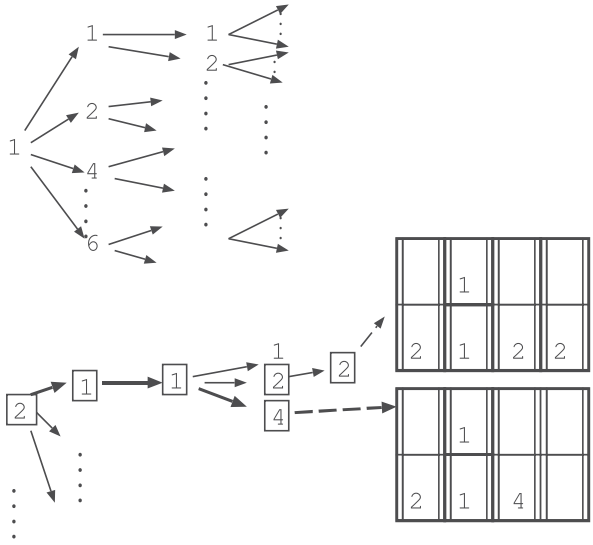


Fig. 4.8: Example of APME approach

which connect distant Tiles, and degrades the operational frequency. Moreover, the communication between TGs is done through the FIFOs allocating edges of a Tile, and so the neighboring TGs that need communication should be mapped into neighboring Tiles. Considering them, I selected only one allocation for all possible patters in each list beforehand. For example, Fig. 4.7 (a) was selected for the combination (2,1,1,2,2). It is called *prepared pattern* in this study.

Fig. 4.8 shows an example of the list (2,1,1,4) and another example of (2,1,1,2,2). In this method, for every branch of the list, a pattern is pre-assigned. Apparently, this method will cause the explosion of the possible patterns if the number of Tiles becomes large. This approach has been proved to be NP-complete, and it requires exponential time in order to find an optimal solution. However, for many up-to-date DPRAs with a realistic size, the number of patterns are reasonable.

Dynamic programming approach In the possible solution space, the same sequence of physical Tiles for a specific sequence of TG_j often appears as a part of many mapping solutions. Instead of searching the whole solution space, the dynamic programming technique can be utilized to find the optimal topological mapping for the complete sequence of TG_j by using solutions for smaller subsequences. Once an optimal solution for mapping up to TG_i is determined, the execution time for executing up to TG_{i+1} can be determined. This step is applied recursively to compute the final optimal mapping.

Given a sequence of Tile Groups ($Size(TG_0)$, $Size(TG_1)$, ..., $Size(TG_k)$), the minimum execution time for executing up to task i in a TG_j, E_{ij} can be computed using the following recursive expression.

$$E_{ij} = t_{ij} + min(E_{i-1})$$

In the expression, t_{ij} is the execution time of the task i in the physical TG_j, and $min(E_{i-1})$ is the total of the minimum execution time of other tasks up to task $(i-1)$. The expression shows that all possible ways of mapping task i are examined once the mapping of task $(i-1)$ has completed.

4.5 Target Device

4.5.1 Device

In this research, for evaluation, I used a real DRPA named DRP-1 from NEC Electronics. The architecture of DRP-1 has been described in Section 3.2.3. The *multi-process* execution in DRP-1 is almost the same as the model introduced in Section 4.3. A task is assigned manually to a basic execution unit called a Tile. Since DRP-1 contains eight Tiles (Fig. 3.15), the maximum number of *tasks* that can be concurrently executed is eight. The DRP architecture is flexible enough to allow a task to be allocated to a group of connected Tiles so as to provide a greater possibility for large tasks, which cannot be assigned to a single Tile, to be implemented. STCs inside Tiles can be operated and controlled independently to provide different instruction pointers for each task, so this allows tasks to run in parallel. A first-in first-out (FIFO) memory mechanism, which employs VMEMs between Tiles, is used as an inter-task communication method. A FIFO is for one-way communication and acts like a pipe. Writing to and reading from a FIFO are blocking, that is, a task needs to be stalled because of the data shortage.

4.5.2 Mapping applications onto DRP-1

After an application is partitioned and modeled as a KPN, it is mapped into the 8-Tile architecture of DRP-1 according to the mapping algorithm described in Section 4.4 with following constraints taking into account the specific features of DRP-1.

- The limit number of Tiles that can be allocated to an application no matter how many tasks it is partitioned into is eight.
- Separate tasks must be mapped into different Tiles, or in other words, the scheduling of two or more tasks sharing the same Tile is forbidden in this implementation. This is because NEC's DRP-1, which is used as the target device for evaluation, does not support different tasks to execute on the same Tile sequentially. In order to solve this problem using the current design tool, all possible combinations of merged tasks must be manually prepared in advance. It is not practical especially when the number of case studies becomes large. With devices providing a task controll mechanism (for example MuCCRA-1 or MuCCRA-2 [79]), this problem can be solved easily.
- A task can be mapped to a Tile Group formed in any shape (like the 4-Tile example in Fig. 4.6).

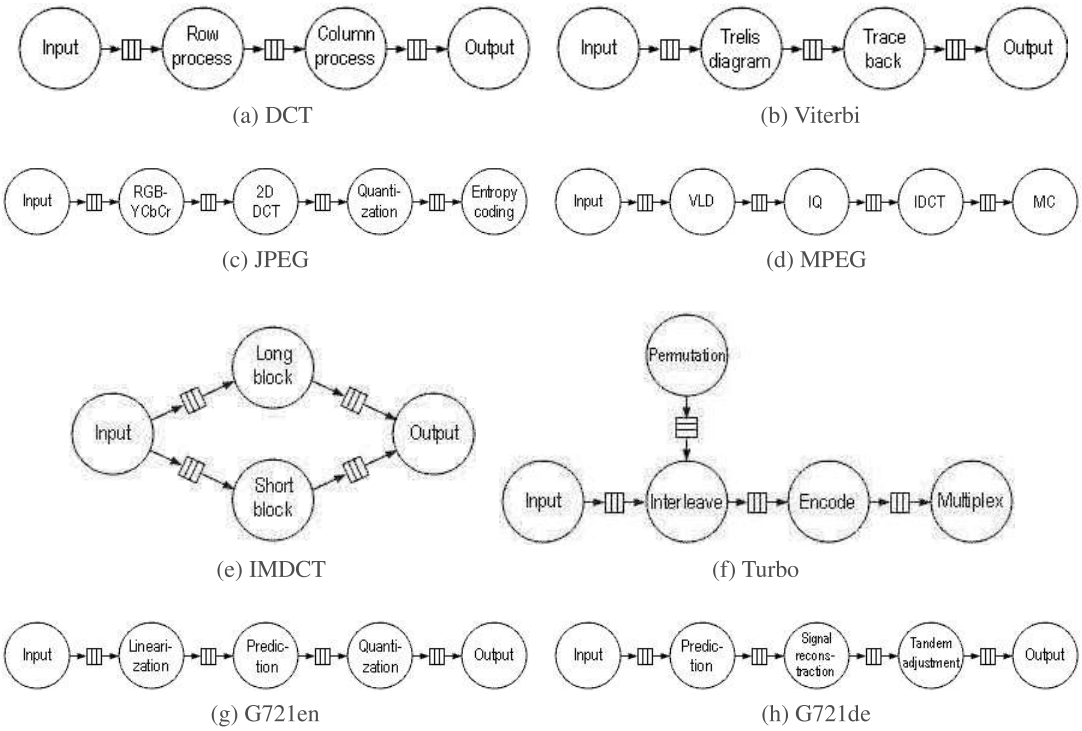


Fig. 4.9: PKN models of target applications

An application is mapped onto the DRP architecture in a way that each task is mapped onto a Tile or a Tile Group according to the mapping algorithm and each task's FIFO is mapped one-to-one onto an DRP's FIFO. Moreover, tasks that need to communicate are mapped to adjacent Tile Groups. An example mapping of the JPEG encoder is shown in Fig. 4.3.

4.6 Evaluation

4.6.1 Target applications

Target applications listed in Table 4.2 have been implemented on DRP-1. Column "#" denotes the number of tasks each application is partitioned into. Column "Variant" shows different mapping versions for an application. Of which, the first version with the name of the application without number added is the one generated by the proposed mapping algorithm. Column "Mapping" shows the number of Tiles each task in a KPN is mapped onto. For example, for the DCT, the column "Mapping" of the first line contains 1, 3, 3, 1. That means the DCT is modeled with four tasks that are mapped to groups of 1, 3, 3, and 1 Tile(s) respectively.

Target applications are partitioned into the tasks of KPNs shown in Fig. 4.9 as follows.

- DCT: input, row-direction computation, column-direction computation, and output
- IMDCT: input, long-block computation, short-block computation, and output

Table 4.2: Implementation results of target applications

Application (Abbr.)	#	Variant	Mapping	Delay [ns]	Clock cycle	Throughput [Mbps]	PEs/ context	Memories/ context
2D DCT [20] (DCT)	1	Single		67.3	95	160.2	98.1	11.0
	4	DCT	1, 3, 3, 1	46.8	112	341.9	141.3	49.3
		DCT1	1, 1, 5, 1	63.7	182	143.5	142.5	48.2
		DCT2	1, 1, 3, 1	46.9	184	194.9	115.0	35.7
		DCT3	1, 2, 4, 1	59.5	136	195.6	139.1	46.7
		DCT4	1, 2, 2, 1	45.1	166	258.0	111.8	45.1
		DCT5	1, 1, 1, 1	44.3	238	180.6	93.7	26.4
IMDCT [20] (IMDCT)	1	Single		140.6	1349	97.2	158.5	36.0
	4	IMDCT	1, 3, 3, 1	94.4	1452	182.3	185.9	39.0
		IMDCT1	1, 4, 2, 1	97.2	1421	178.6	188.0	40.1
		IMDCT2	1, 2, 2, 1	89.5	1592	169.8	178.3	39.0
Viterbi decoder [20] (Viterbi)	1	Single		30.4	11	3.0	52.7	0
	4	Viterbi	1, 2, 2, 1	25.9	21	4.3	73.0	10.8
		Viterbi1	1, 3, 3, 1	33.5	18	3.7	84.0	10.8
		Viterbi2	1, 4, 2, 1	37.2	18	3.8	87.7	10.8
		Viterbi3	1, 5, 1, 1	48.1	17	3.5	85.2	10.8
		Viterbi4	1, 1, 1, 1	25.2	27	2.6	42.1	10.8
JPEG encoder [94] (JPEG)	1	Single		56.3	636	42.9	136.3	19.8
	5	JPEG	1, 2, 2, 1, 2	53.1	894	106.3	196.4	65.2
		JPEG1	1, 1, 2, 1, 2	53.0	956	97.3	178.7	56.3
		JPEG2	1, 1, 3, 1, 2	58.6	882	100.8	214.5	65.2
Turbo encoder [95] (Turbo)	1	Single		81.3	16065	1.4	32.5	12.4
	5	Turbo	1, 1, 2, 2, 2	58.3	17642	4.0	48.8	39.5
		Turbo1	1, 1, 1, 3, 2	69.3	17498	3.9	54.2	39.5
		Turbo2	1, 1, 1, 2, 1	58.1	18312	3.4	36.9	32.4
		Turbo3	1, 1, 2, 2, 1	58.3	17936	3.9	31.1	39.5
MPEG-2 decoder [96] (MPEG)	1	Single		73.9	1564	3.9	144.9	16.3
	5	MPEG	1, 2, 2, 1, 2	62.9	1680	10.4	275.1	45.1
	5	MPEG1	1, 2, 1, 1, 3	69.7	1744	7.6	289.0	52.3
G721 encoder (G721en)	1	Single		91.2	618	16.6	98.9	21.1
	5	G721en	1, 1, 2, 2, 2	78.8	754	45.1	126.0	52.1
	5	G721en1	1, 1, 3, 1, 2	85.7	766	25.2	115.3	47.2
	5	G721en2	1, 1, 2, 3, 1	86.1	812	29.5	119.1	48.2
	5	G721en3	1, 2, 2, 1, 2	77.2	784	34.3	131.4	51.3
	5	G721en4	1, 2, 2, 2, 1	77.5	788	37.7	132.0	51.1
G721 decoder (G721de)	1	Single		94.7	669	4.1	74.6	13.5
	5	G721de	1, 2, 2, 2, 1	77.8	818	10.3	113.1	37.7
	5	G721de1	1, 3, 2, 1, 1	87.0	796	5.9	105.7	32.3
	5	G721de2	1, 3, 1, 2, 1	85.1	806	7.3	103.2	34.2
	5	G721de3	1, 2, 1, 2, 2	79.3	858	8.1	111.0	38.5
	5	G721de4	1, 2, 2, 1, 2	77.4	832	8.7	117.3	38.5

- Viterbi: input, trelis diagram, trace back, and output
- JPEG: input, RGB-YCbCr conversion, two-dimensional DCT, quantization, and entropy coding

- MPEG: input, variable length decode (VLD), inverse quantization (IQ), inverse discrete cosine transform (IDCT), and motion compensation (MC)
- Turbo: input, permutation, interleave, encode, and multiplex
- G721en: input, input data linearization, prediction, quantization, and output
- G721de: input, prediction, signal reconstruction, tandem adjustment, and output

4.6.2 Mapping versions

For comparison, the result of the *single-process* execution, where applications are not partitioned but the whole application is mapped into 8 Tiles of the target architecture as only one thread of control, is provided in Table 4.2 with the line "Single" of column "Variant". Also, in this execution, column "#" shows the value of 1 to clearly specify that applications are mapped as one task for executing.

For comparison, different mapping versions of applications shown in Table 4.2 with variety number of Tiles assigned to tasks are also evaluated. They are not the whole possible mapping space, but typical representatives. For demonstrating efficiency of the proposed algorithm with practical applications, there exist two problems: (1) there is no standard algorithm to be compared, and, (2) with simple algorithms (for example, random mapping), it is difficult to successfully complete the place-and-route phase, so it is impossible to get the designs to evaluate their quality. In order to address this problem, I stretched the exploration space by relaxing the following conditions.

- *Task grouping:* Tasks are grouped without taking into account balancing their computation amount. Other conditions are still kept to get the reasonable design.
- *Adjusting:* At this step, there are two options related to whether unused Tiles are exploited to improve the execution time of TG_j. In a version, groups TG_j are adjusted according to the proposed step in Section 4.4.3.2 with unused Tiles in order to reduce the execution time of target groups. In the other version, groups TG_j are not adjusted with unused Tiles.
- *Topological mapping:* this step is performed exactly the same as proposed in Section 4.4.3.3.

These steps were performed almost automatically, and I tried to get as many results as possible. If the mapping results can successfully pass the place-and-route phase, it is recorded and assigned a tag number consisting of an application name and a number. On the other hand, if the version fails at the place-and-route phase, it is not taken into consideration. For some applications, I could get a lot of mapping results, and Table 4.2 only shows results with good achievement either in the performance metrics: critical path, clock cycle and throughput. However, other applications like MPEG, only one version other than the proposed method could pass the place-and-routing phase.

4.6.3 Implementation results

Table 4.2 presents the performance result of target applications. In column "Variant", *Single* denotes the result of applications implemented in the *single-process* execution. Columns "PEs/context" and "Memories/context" show the average number of required PEs and used memories (both VMEMs and HMEMs) in a context.

In some applications like Viterbi, not all Tiles are assigned because the parallelism of these applications is not high enough. Usually, the execution time could be reduced by increasing the number of Tiles; but, for applications having low parallelism, only small number of PEs is required, so it is not expected to reduce execution clock cycles even with a larger number of Tiles. Moreover, since using larger numbers of Tiles with low usage will increase the leakage and the clock distribution power, an optimum number of Tiles should be carefully selected for each implementation [82]. In many cases, this causes the actual number of Tiles is not the largest one as shown in Table 4.2.

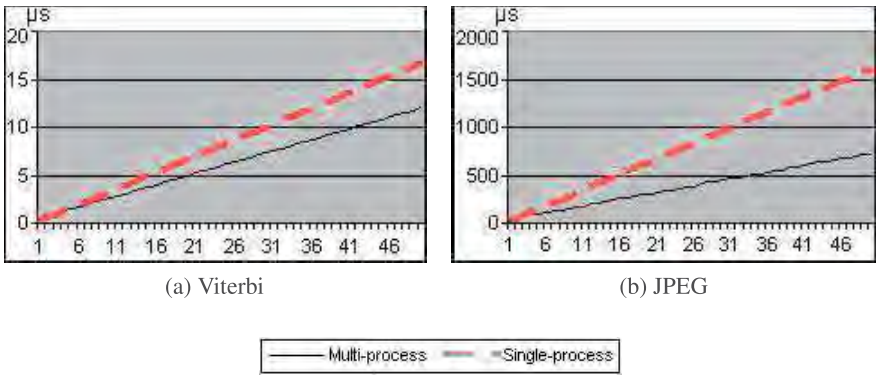
4.6.4 Throughput

The throughput of an implementation is represented with the amount of data processed in a second. Since all tasks are executed in a pipelined manner in the *multi-process* execution, the throughput of the total execution is limited by the stage with the largest execution time.

Table 4.2 shows that all implementations in the *multi-process* execution improve the throughput in a certain degree over the *single-process* execution up to nearly three time in JPEG implementation. In other words, by representing an application as a task network that are mapped into a DRPA as separate threads of control, the throughput could be improved. As the size of an input data block is the same, the throughput could be improved by either reducing the critical path or the number of execution clock cycles. In the *multi-process* execution, the critical path is usually shorter than that from the *single-process* execution. While a target application is mapped into the whole reconfigurable array in the *single-process* execution, in the *multi-process* execution, the critical path of an application is the longest one among child tasks, each of which is often mapped to one or several Tiles but not the whole reconfigurable array. For example, in DCT, the implementation in the *single-process* execution requires 8 Tiles of NEC's DRP, but in the *multi-process* execution, the largest task can only be mapped to 5 Tiles (version DCT1 in Table 4.2).

Moreover, the calculation of the throughput in the *multi-process* execution takes into account the largest number of execution clock cycles among tasks; and, by dividing an application into independent tasks, that number is often smaller than that of a big task executing in the *single-process* execution though the total number of clock cycles from all tasks is often larger due to task overhead. Since both the critical path and the number of execution clock cycles in the *multi-process* execution could be shorten, the throughput is likely to be improved.

The throughput could also be improved by taking advantages of the pipeline technique where multiple tasks are arranged to operate in a pipelined manner. In most cases, the output of a com-



(a) Viterbi (b) JPEG

Fig. 4.10: Execution time vs. Number of data blocks

putation step will be the input of the next step with no data or control hazard, so this is suitable for pipelining.

Among implementations in the *multi-process* execution mode, the one with my mapping algorithm achieves the best throughput. According to Hasegawa [82], each implementation has an optimum context size (the number of Tiles) where the performance becomes optimum. Other context sizes no matter whether they are smaller or larger than the optimum one result in performance degradation. The implementation according to the mapping algorithm is likely the one where constituent tasks are mapped with the optimum context sizes; therefore, the optimum throughput could be achieved. More importantly, in a pipelined environment, the throughput is greatly influenced by the balance of computation stages. The proposed mapping algorithm produces the most appropriate result in terms of task workload balancing. For example, balancing two tasks, row-direction computation and column-direction computation, in DCT is the most important factor for the throughput since they occupy the largest part of the total execution time.

Although the proposed mapping algorithm could improve the throughput in a certain degree, the main limitation is the number of available contexts. Since the number of required contexts becomes easily more than 16, the possibility of grouping tasks are strictly limited. This is the reason why the execution time of each task is still unbalanced.

4.6.5 Execution time

The execution time of an implementation for a given set of data can be computed as the product of the critical path and the number of execution clock cycles.

In the *single-process* execution, when multiple blocks of data are given, the number of execution clock cycles will be the multiple of that with one block of data and the number of data blocks. On the other hand, in the *multi-process* execution, because of the effect of pipelined processing, when more data are fetched, the number of execution clock cycles keeps increasing but less than the correspondence in the single-process execution. This is illustrated in Fig. 4.10, which shows the total

execution time for Viterbi and JPEG when processing N data blocks ($N = 1, 2..., n$). In the graph, the largest number of data blocks n equals to 50. Other target applications show similar behavior. When a number of data block is small, the execution time in the single-process execution is smaller because of the overhead in the multi-process execution. However, when there are enough input data, or in other words, when a number of data block provided is larger than a certain number, to keep the pipeline stages full, the execution time in the multi-process execution becomes smaller.

From Table 4.2, among implementations in the *multi-process* execution mode, the one with the proposed mapping algorithm achieves the smallest execution time since it has either the shortest critical path or the smallest execution clock cycle.

4.6.6 Area utilization

In order to investigate the mapping quality, I evaluate the area utilization in terms of the number of active PEs and on-chip memories. The overall area utilization could be presented by the number of consumed resources in every context. Since the number of contexts varies among applications, the average resource per a context will be used for examining the area utilization.

In Table 4.2, two columns *"PEs/context"* and *"Memories/context"* show the average number of active PEs and used memories in a context. They are calculated by dividing the total number of PEs and memories (both VMEMs and HMEMs) respectively for an implementation by the total number of contexts. From these two columns, it can be seen that, implementations in the *multi-process* execution demand more resources than correspondent implementations in the *single-process* execution. In terms of used PEs, the number of active PEs in the *multi-process* execution is often larger than that in the *single-process* execution. This is the result of having more than one task running at the same time.

Similarly, the requirement for active memories in the *multi-process* execution is also higher. One of the reasons is the use of FIFOs for inter-task communication. Even when requiring no memory for storing, an implementation still uses memories for communicating via FIFOs. This is illustrated in the implementation of a Viterbi decoder; while no memory for storing is demanded in the *single-process* execution, the *multi-process* execution still uses memories for FIFOs.

4.6.7 Two methods for topological mapping

As mentioned in Section 4.4.3.3, there are two methods for topological mapping in the proposed algorithm: all possible mapping exploration and dynamic programming approach. The difference between these two methods is the time they take to produce the final result. In order to find the difference, a program for topological mapping according to both methods is implemented, then the execution time is computed. The programs are compiled by gcc 3.3.6 with -O3 optimization option and executed on a Pentium 4 processor 3.2GHz with 512Mb internal RAM.

Table 4.3 shows the time needed for mapping each target application onto the target application

Table 4.3: Time for topological mapping

Application	All possible mapping exploration	Dynamic programming
DCT	65.4	2.3
IMDCT	65.3	2.3
Viterbi	65.3	2.3
JPEG	384.2	2.8
Turbo	384.2	2.8
MPEG	384.2	2.8

DRP-1. Columns "All" and "Dynamic" represent the methods of all possible mapping exploration and dynamic programming respectively.

Table 4.3 shows that the dynamic programming method greatly reduces time for topological mapping. Since target applications are not modeled with too many tasks, and the target architecture only has eight Tiles, the topological mapping step does not take too much time to produce the final result. Nonetheless, when the number of tasks an application is modeled increases, or when the target architecture has more number of Tiles, time for topological mapping will become a great concern. For example, taken time for applications modeled with four tasks (DCT, IMDCT and Viterbi) and applications modeled with five tasks (JPEG, Turbo and MPEG) on the same target architecture is considerably different when the method of all possible mapping exploration is applied.

Table 4.3 also shows that time for topological mapping depends on the number of tasks each application is divided into since applications with the same number of tasks take almost the same time. Moreover, the number of Tiles on the target DRPA influences time for mapping as well.

4.7 Conclusion

A systematic method for mapping an application modeled as a KPN onto a dynamically reconfigurable processing array is proposed. Using real applications and a real target architecture DRP-1, the impact of the proposed method on performance and area utilization is evaluated and analyzed. Evaluation results show that the throughput of the *multi-process* execution increases from two to three times compared with the *single-process* execution, while more area utilization is realized as a result of tasks being executed in parallel. In addition, my proposed mapping method results in the best throughput and execution time.

Chapter 5

Hardware Task Preemption

5.1 Problem

In order to further exploit the flexibility of reconfigurable devices, operating systems for managing task allocation, scheduling and configuration have been introduced. One of the focus areas of many studies is to build a multitasking environment to allow different tasks to efficiently share a piece of reconfigurable hardware. Among problems for realizing such an environment, a task preemption mechanism, where a higher priority task is allowed to interrupt an executing task, plays an important role. However, such a mechanism for tasks running on hardware circuits is not trivial involving the question of how to suspend and resume a hardware execution, and especially, how to capture the state data of a given task within a certain latency while guaranteeing a reasonable hardware overhead.

A considerable number of coarse-grained DRPAs such as DRP, DAPDNA-2, FE-GA and SAKE [14], which exploit a multicontext architecture to reduce configuration overhead, have been developed and commercialized. By providing storage for multiple configurations in each processing element, hardware configuration can be changed quickly often in a clock cycle. Compared with fine-grained FPGAs, since data and time for setting configuration from outside are small, task switching involving preemption in DRPAs seems to be more realistic than those for FPGAs. Nonetheless, while scheduling without preemption has been researched [97], and several methods to approach the problem of task preemption on FPGA-based devices have been proposed, only a few research efforts to implement such functions into DRPAs have been carried out.

In this chapter, I propose a method for preempting a hardware task and capturing its state data based on the analysis of resource usage at the design time. By modifying the state transition graph of applications implemented on dynamically reconfigurable systems at computation steps where the requirement of resources is small, the impact of preemption on performance and cost for task switching could be reduced.

5.2 Related Work and Research Contribution

5.2.1 Related work

Hardware multitasking on FPGAs have been a challenging subject for many researches. [29] deals with the support of concurrent applications in a multi-FPGA system by reconfiguring entire FPGAs. Storing the state information of a hardware task during interruption is discussed in [31]. The outline of a multitasking environment, which enables several tasks to run in parallel, is introduced in [98]. A hardware check-pointing approach for reconfigurable devices is proposed in [99] to divide a task into smaller modules to minimize the lost of computation when failures occur. In order to do so, the state of a task must be saved frequently at pre-defined checkpoints during execution.

One of the most important features to enable a preemptive multitasking environment is task preemption. In a such environment on reconfigurable devices, several solutions for saving and restoring the state data of a hardware task have been proposed. However, they mainly target fine-grained FPGAs.

- *Readback:* This solution is based on the configuration read-back capability, which allows to read the content of both registers and internal memory modules [31] [30] [100] [101]. Although demanding no extra hardware, the solution is slow due to a great amount of configuration data, and requires additional computation to filter out useless information. More importantly, the format of the configuration stream is crucial to extract useful data; so, this makes the solution depend on specific FPGAs.

- *Internal state supervision:* By adding extra interfaces to registers and internal memory modules, it is possible to access these elements when saving and restoring context data. This solution could be implemented as a scan chain, a memory-mapping structure, or a scan chain with shadow registers [101] [99] [102] [103]. Although achieving data efficiency as only needed information is saved, this approach demands extra resources and design efforts to implement interfaces to registers and memories. Additionally, hardware circuits should be modified, which is almost impossible for already available devices.

[104] proposes a systematic methodology for incorporating preemption constraints in application specific multi-task VLSI systems. By considering a predetermined set of applications, the method tries to insert preemption points taking into consideration both dedicated and shared registers in order to minimize the task switching overhead. This approach is suitable for fixed hardware platform like ASICs in which registers to be saved can be determined in advance.

5.2.2 Research contribution

The method proposed in this research differs from above-mentioned solutions in the following aspects.

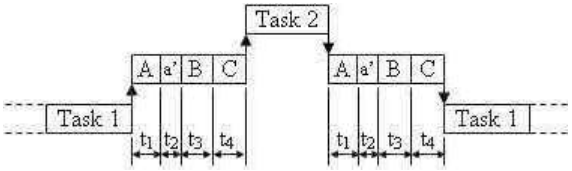


Fig. 5.1: Task switching

- By modifying the source program at high level languages (C and Verilog) during the design time to insert special states for saving and restoring state data, the method is independent from the details of the underlying hardware architecture such as which registers and memories are assigned to variables at the run time.
- The proposal targets coarse-grained DRPA devices, whose system design flow is tightly and automatically integrated with the necessary steps of the method in order for designers to examine and justify how the method affects implemented applications.
- The impact of the method on performance, hardware overhead and preemption latency can be measured at the design time.

The main contribution of this research includes:

- Algorithms to select and optimize the preemption points of a target application subject to given preemption latency constraints.
- Important steps to integrate the proposed method into the system design flow in order to automate the process of identifying and optimizing the list of preemption points for a target application during the design time.

5.3 Preemption Analysis

5.3.1 Task switching

In a preemptive multitasking environment, a typical task switching process can be illustrated on Fig. 5.1. While Task 1 is running, an interrupt signal, often caused by a system timer, indicating a possible task switch is generated. Before a new task (Task 2) can be executed, several preparing stages have to be done. First, an interrupt service is called to decide whether a task switch is necessary (Stage A). If it is, it might take some more times to wait because Task 1 may not be suspended immediately (Stage a'). Then, when Task 1 is ready to be stopped, the state data of Task 1 is saved in Stage B; and, that of Task 2, which had been preserved before, is loaded in Stage C.

(t_1) can be considered as interrupt latency; and, $\sum_{i=2}^{4} t_i$ is context switch latency. In many systems and depending on certain tasks, there is no Stage a', or in other words, tasks can be usually suspended immediately when receiving a preemption request. In such cases, time according to Stage a' (t_2) is

zero. On the other hand, some systems or certain tasks may not be interrupted right after receiving a request, for example, when the current running task is in a critical area, or when tasks only allow to be interrupted at certain points during execution. In this case, t_2 has a certain value. Generally speaking, the sum $\tau = \sum_{i=1}^{4} t_i$ can be considered as preemption latency. Usually, Stage A does not take a long time for modern processors and operating systems. In this study, I do not take Stage A or interrupt latency (t_1) into consideration, so preemption latency can be computed as follows:

$$\tau = \sum_{i=2}^{4} t_i \tag{5.1}$$

However, Stages B and C may take a considerably amount of time since the information representing the context of of a hardware task is specific for a given task implementation and may scatter on different state-holding elements. The amount of data captured when preempting a hardware task is often considerably large; so, the cost for a hardware task preemption mechanism should be minimized.

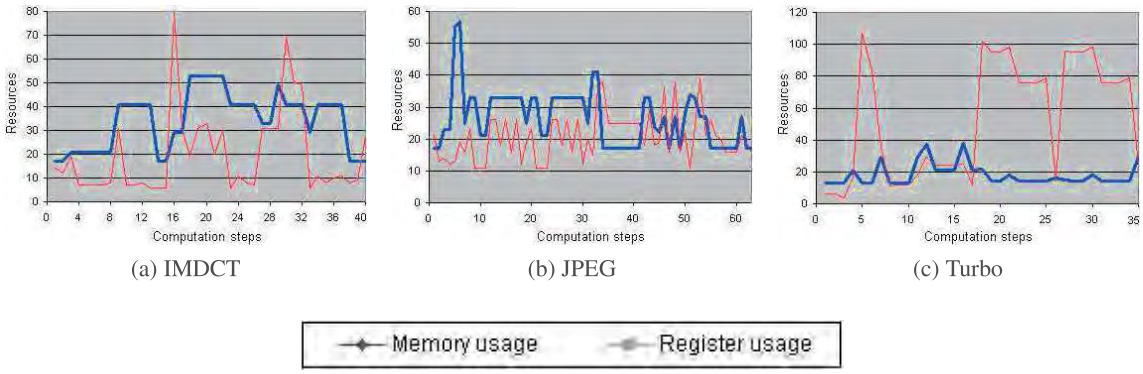
5.3.2 Approach

During execution, the amount of variables of a task mapped to internal registers and memories for storing intermediate results is considerably varying over time. Most target applications for dynamically reconfigurable processors are stream processing, that is, data blocks to be processed are iteratively received in a certain interval. Normally, between the processing of two data blocks, the amount of state data is relatively small. This fact can be applied to build a preemption mechanism that allows preemption only at predefined steps called *preemption points* [105] or *switchpoints* [106] during execution.

5.3.2.1 Preliminary evaluation

Fig. 5.2 shows requirement for resources in terms of memories and registers in each computation step when an IMDCT, a JPEG encoder and a Turbo encoder are implemented on NEC's DRP-1. X-axis of the figures shows the computation steps, and Y-axis shows the number of registers and memories. As shown in Fig. 5.2, the number of registers and memories for storing intermediate results and switching contexts greatly varies from step to step. For example, in IMDCT, steps 0, 1, 2, 3, 4, 6, 13, 19, 22, 29 and 34 do not require too many memories and registers; in addition, steps 10, 11, 23, 24 and 29 do not use a lot of memories though the number of registers is remarkable. Accordingly, by only allowing preemption at steps corresponding to steps where the requirement for memories and registers is minimized, the amount of data necessary to be saved can be dramatically reduced.

For a given task, in order to identify which data should be the target for capturing if the task is preempted at a certain computation step, it is critical to examine what resources are required and how they are used. The former can be done by source program analysis to identify variables and their life-time; and, the latter can be determined by an appropriate simulation. In a typical program,



(a) IMDCT (b) JPEG (c) Turbo

Fig. 5.2: Memory and Register usages vs. Computation steps

there are two types of variables: global and local, which specify the scope of variables. When a task is preempted, basically, all global variables should be saved; furthermore, local variables relating to the preempted place should also be captured. In addition, during execution, the use of variables is varying based on execution time and computation step.

5.3.2.2 Solution

In my approach, it is critical to evaluate the usage of variables in order to specify potential locations to be preemption points. Specifically, the proposed method is based on the state transition graph and the resource examination of a target application. This kind of resource evaluation can be done at the early stage of design by synthesis tools. The solution should be automatically done with a certain algorithm, since it must be combined into the design tool in the future. That is, the steps of the proposed method are automatically attached while an application is designed. The following policies are adopted.

- Steps where preemption is allowed are limited at predetermined points called *preemption points* where the demand of registers and memory modules is smaller than a certain limitation.
- At preemption points, special states tailored to current contexts are added for flushing and restoring state data.
- Performance degradation resulting from preemption is evaluated in order to optimize preemption points.
- Proposed algorithm is integrated into the system design flow of a target device.
- Algorithms for selecting and inserting preemption points are based on the design tools of a target architecture.

5.3.3 State transition graph

The homogeneous synchronous data flow [107] is used as the computational model for applications. The model iteratively processes semi-finite data blocks arriving in a certain interval. Stream applications, which are the main target of DRPAs, are suitable to be represented by this model. The behavior of a hardware task could be represented in the form of a State Transition Graph (STG) $G(N, E, START, END)$, where N is the set of nodes representing computation states; E is the set of edges showing the transition and data dependence between states; $START \in N$ is a distinguished start node without incoming edges; and $END \in N$ is a distinguished end node without outgoing edges. Fig. 5.5 shows the STG of two tasks, where (a_i, b_i) $(i = 0...n)$ shows numbered states, arrows represent possible transitions from states to states, (a_0, b_0) and (a_9, b_6) are start and end states. Transition can be switched conditionally as in state a_2.

5.4 Preemption Algorithms

5.4.1 System design flow

Fig. 5.3(a) shows a typical design flow for DRPAs, in which the compiler or the behavioral synthesis, the technology mapper, the place-and-route and the code generation tool are assumed to be design tools available for the target DRPA. First, a source C-based program and an architecture description are taken as inputs for the behavioral synthesis, which extracts control flows as well as data flows, allocates operation resources, and produces reports about required resources. Then, the technology mapper actually produces the code in the form of hardware description language (HDL) for processing elements, and, a functional simulation at the register transfer level (RTL) can be executed. The place-and-route tool compiles the HDL code into a netlist. Exact reports about resource usage and critical path can be obtained at this step. Finally, the code generator produces configuration code for the underlying reconfigurable hardware.

Fig. 5.3(b) presents a modified design flow with the proposed preemption algorithm. Since most current DRPAs do not support dynamic memory allocation, the resource report produced by the compiler could describe quite exactly how variables are allocated and which resources are necessary for task switch. This is the basic for preliminarily analyzing preemption points and inserting preemption states at the step *preemption insertion*. The *evaluation* step is based on the RTL simulation for evaluating how added preemption states affect the implementation. The last step is the *preemption refinement* where preemption points are modified according to the exact report of resource usages and the evaluation result. A modified source program will be fed back to the technology mapper for re-compiling and re-evaluating.

According to the modified design flow, three additional steps are inserted to support the proposed preemption mechanism. Each step requires different input files and produces correspondent results as shown on Fig. 5.4.