2001

Towards adaptive balanced computing (ABC) using reconfigurable functional caches (RFCs)

Hue-Sung Kim
Iowa State University

INFORMATION TO USERS

This manuscript has been reproduced from the microfilm master. UMI films the text directly from the original or copy submitted. Thus, some thesis and dissertation copies are in typewriter face, while others may be from any type of computer printer.

The quality of this reproduction is dependent upon the quality of the copy submitted. Broken or indistinct print, colored or poor quality illustrations and photographs, print bleedthrough, substandard margins, and improper alignment can adversely affect reproduction..

In the unlikely event that the author did not send UMI a complete manuscript and there are missing pages, these will be noted. Also, if unauthorized copyright material had to be removed, a note will indicate the deletion.

Oversize materials (e.g., maps, drawings, charts) are reproduced by sectioning the original, beginning at the upper left-hand corner and continuing from left to right in equal sections with small overlaps.

Photographs included in the original manuscript have been reproduced xerographically in this copy. Higher quality 6" x 9" black and white photographic prints are available for any photographs or illustrations appearing in this copy for an additional charge. Contact UMI directly to order.

ProQuest Information and Learning
300 North Zeeb Road, Ann Arbor, MI 48106-1346 USA
800-521-0600

Towards adaptive balanced computing (ABC) using reconfigurable functional caches (RFCs)

by

Hue-Sung Kim

A dissertation submitted to the graduate faculty

in partial fulfillment of the requirements for the degree of

DOCTOR OF PHILOSOPHY

Major: Computer Engineering

Major Professors: Arun K. Somani and Akhilesh Tyagi

Iowa State University

Ames, Iowa

2001

UMI Number: 3016717

UMI Microform 3016717

Bell & Howell Information and Learning Company
300 North Zeeb Road
P.O. Box 1346
Ann Arbor, MI 48106-1346

Graduate College
Iowa State University

This is to certify that the Doctoral dissertation of

Hue-Sung Kim

has met the dissertation requirements of Iowa State University

Signature was redacted for privacy.

Co-major Professor

Signature was redacted for privacy.

Co-major Professor

Signature was redacted for privacy.

For the Major Program

Signature was redacted for privacy.

For the Graduate College

DEDICATION

I would like to dedicate this thesis to my wife, Eunjung, and to my daughter, Haerin. Without their incredible support I would not have been able to complete this work. They encouraged me to follow the right direction at a moment when I was discouraged by failure. This work was completed not only by my own enthusiasm, but also by their endless attention and love. In addition, I would like to thank my friends and parents for their emotional support during the writing of this work.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

ABSTRACT

The general-purpose computing processor performs a wide range of functions. Although the performance of general-purpose processors has been steadily increasing, certain software technologies like multimedia and digital signal processing applications demand ever more computing power. Reconfigurable computing has emerged to combine the versatility of general-purpose processors with the customization ability of ASICs. The basic premise of reconfigurability is to provide better performance and higher computing density than fixed configuration processors. Most of the research in reconfigurable computing is dedicated to on-chip functional logic. If computing resources are adaptable to the computing requirement, the maximum performance can be achieved. To overcome the gap between processor and memory technology, the size of on-chip cache memory has been consistently increasing. The larger cache memory capacity, though beneficial in general, does not guarantee a higher performance for all the applications as they may not utilize all of the cache efficiently. To utilize on-chip resources effectively and to accelerate the performance of multimedia applications specifically, we propose a new architecture - Adaptive Balanced Computing (ABC). ABC uses dynamic resource configuration of on-chip cache memory by integrating Reconfigurable Functional Caches (RFC). RFC can work as a conventional cache or as a specialized computing unit when necessary. In order to convert a cache memory to a computing unit, we include additional logic to embed multi-bit output LUTs into the cache structure. We add the reconfigurability of cache memory to a conventional processor with minimal modification to the load/store microarchitecture and with minimal compiler assistance. ABC architecture utilizes resources more efficiently by reconfiguring the cache memory to computing units dynamically. The area penalty for this reconfiguration is about 50-60% of the memory cell cache array-only area with faster cache

access time. In a base array cache (parallel decoding caches), the area penalty is 10-20% of the data array with 1-2% increase in the cache access time. However, we save 27% for FIR and 44% for DCT/IDCT in area with respect to memory cell array cache and about 80% for both applications with respect to base array cache if we were to implement all these units separately (such as ASICs). The simulations with multimedia and DSP applications (DCT/IDCT and FIR/IIR) show that the resource configuration with the RFC speedups ranging from 1.04X to 3.94X in overall applications and from 2.61X to 27.4X in the core computations. The simulations with various parameters indicate that the impact of reconfiguration can be minimized if an appropriate cache organization is selected.

CHAPTER 1. INTRODUCTION

The world's first electronic digital computer – the Atanasoff-Berry Computer – was developed at Iowa State University during 1937-42 [1]. After the advent of this first digital computer, new technologies have been developing dramatically. This high level of improvement and innovation in technologies has made computer systems one of the most essential pieces of equipment in the world. The central component in computer technology is the microprocessor, which executes tasks and controls the utilization of other components. Industry and academia have been building high performance microprocessors with significant enhancements in architecture and physical design. In this thesis, we propose a new architecture – Adaptive Balanced Computing (ABC) using a Reconfigurable Functional Cache (RFC) as one possible solution to attain high performance in current and future microprocessors. First, we compare the reconfigurable computing with general-purpose computing in Section 1.1. Then, in Section 1.2, we describe the concept of adaptive balanced computing. Finally, the motivation and approaches of our work are presented in Section 1.3.

1.1 General-purpose computing vs. reconfigurable computing

The general-purpose computing processor performs a wide range of functions. This versatility makes the general-purpose processor one of the most flexible devices in machinery; it is also cost-effective for the versatility provided. The trade-off for this versatility, however, is performance. Although the performance of general-purpose processors has been steadily increasing, certain software technologies like multimedia and digital signal processing applications demand ever more computing power. These computations can be accelerated by embedded processors and/or Application-Specific Integrated-Chips (ASICs). The general-purpose pro-

cessor is assisted by ASIC-like co-processors to provide a specialized/customized computing unit. However, it is quite expensive to integrate an ASIC-like specialized computing unit into a general-purpose processor due to the chip area constraint. In addition, the general purpose processor's versatility requirement limits the level of customization.

Reconfigurable computing has emerged to combine the versatility of general-purpose processors with the customization ability of ASICs. The reconfiguration in logic functionality allows for a number of computations to be accelerated using specialized/customized units for the computing requirements. In a given chip area, the density of computing power in reconfigurable logic is higher than in general-purpose processors and ASICs [2]. Another feature of reconfigurable computing is the ability to execute computations at task (or function) level. Unlike instruction-level execution in general-purpose processors, an entire function can be mapped and executed on the reconfigurable logic. This provides a function-level (coarse-grained) optimization to increase the performance of applications. The reconfigurable logic that is integrated into general-purpose processors can reduce the number of instructions by replacing the instructions with function-level customized logic. Moreover, the on-chip reconfigurable logic provides the conventional processor with an opportunity to exploit the parallelism by unloading a heavy computation (an entire function) from the core processor to the logic. Given these benefits, the major problem with reconfigurable logic is the configuration time and the dominating chip area for the interconnects compared to logic-cell area. However, we expect the configuration time to be relatively small in comparison to the computing time; otherwise, the acceleration of computation would be counteracted. Because the programmable interconnections among the logic cells are the basis for reconfigurability, the larger chip area for the interconnects is unavoidable. However, the programmability of interconnection could be adjusted and customized for a range of applications if an area optimization is highly desired.

1.2 Adaptive balanced computing

Computing performance can be characterized by the balance [3] between the computing and memory bandwidths of a processor. If the memory bandwidth matches the demand from the computing bandwidth, or vice versa, the highest performance can be achieved. Nevertheless, such an ideal computing/memory balance is unrealistic unless infinite amount of resources are available and adjustable.

The current general-purpose processors may not be able to offer such a dynamic resource balance because of the fixed resource configuration. A static allocation of resources was attempted in [4], where processor, cache and bus resources were balanced. However, a noteworthy feature of current processors is the large amount of on-chip real-estate dedicated to storage (such as caches, registers, and buffers). In conventional processors, more than half of an entire chip area is cache memory [5]. The high capacity of on-chip storage provides the instructions and data at peak speeds by reducing the stall time caused by off-chip communications. A certain threshold of performance can be achieved with a large size of memory in a wide range of applications. However, a large amount of fixed memory does not always guarantee better performance for all applications because of different memory and computing requirements for applications.

The basic premise of reconfigurability is to provide better performance and higher computing density [2] than the fixed configuration processors. Currently, most of the research in reconfigurable computing is dedicated to on-chip functional logic [6, 7, 8, 9, 10, 11, 12, 13]. The logic reconfiguration delivers greater performance by providing highly specialized computing resources. However, the on-chip resources are under-utilized if few computations exploit the logic. This results in a low balance between computing and memory in various applications.

If a variable type of computing resources is provided dynamically, the maximum performance over a variety of applications can be achieved. This dynamic resource configuration offers an adaptive balanced computing environment to the application. In this thesis, we present the benefits of adaptive balanced computing for multimedia and digital signal processing applications.

1.3 Motivation and approach

There are several challenges in the design of next generation microprocessors, such as instruction-level parallelism, compiler optimizations, higher reliability, adaptability, and memory capacity/performance [14] with a billion transistors [15, 16]. To overcome the gap between processor and memory technology, the size of on-chip cache memory has been constantly increasing. This trend is likely to continue even in deep-submicron technology. In a modern microprocessor, more than half of the transistors (80% of the total transistor and up to 50% of die area [17]) are used for cache memories. For example, Hewlett-Packard PA-8500 [18] contains 1.5MB cache as 1MB for D-cache and 0.5MB for I-cache. The larger cache memory capacity, though beneficial in general, does not guarantee a higher performance for all the applications as they may not utilize the full cache efficiently. Moreover, a larger cache memory hardly increases the performance of multimedia applications due to the streaming nature and lack of temporal locality in media data [19, 20, 21].

Merged DRAM Logic (MDL) [22, 23] has been introduced to provide data processing at peak speeds with faster accesses to data storage from computing units. This is done by integrating static logic and DRAM onto the same die. Intelligent Memory (IRAM) [24, 25] has also been integrated into on-chip microprocessors to reduce the bottleneck of off-chip communication. This is accomplished by increasing the capacity of on-chip storage using DRAM with a high density instead of SRAM. However, the integration of static logic and DRAM generates design difficulties due to the different fabrication process technologies. This causes degradation in terms of area and time for both logic and DRAM. For example, the performance of logic gates in MDL is degraded due to the slower transistor switching and the area is also increased due to the fewer number of routing layers under the DRAM process.

In deep-submicron technology, only a relatively small region can be reached in one processor clock cycle due to the delay of interconnects. For instance, only 16% of the die length (in billion transistors) can be reached within one cycle period (at 1.2 GHz) at $0.1\mu m$ technology [26]. This implies that a core control unit in a microprocessor may not reach all the resources in a short cycle time. This results in more complicated signal propagation structure. For example, more

driver buffers are needed between segments of wire to reduce the effect of relatively long wire delay. It would be beneficial to have high-demand resources within a single cycle distance. The resources that can perform independent of the processor core could be placed farther away. Any time-intensive computation that can be carried out independently may be scheduled on such resources. This allows other resources to remain physically closer to the core processor within one cycle distance.

To provide resources variable to the computing requirements, we propose a new architecture – Adaptive Balanced Computing (ABC). ABC uses a dynamic resource configuration of on-chip cache memory by converting the cache into a specialized computing unit, which is able to carry out independent computations. A reconfigurable functional cache (RFC) has the ability to operate as conventional cache memory or as a specialized computing unit [27, 28, 29]. With a small amount of additional logic and a slightly modified microarchitecture, a part of the cache memory can be configured to perform specialized computations in a conventional processor.

Several researchers have studied the use of reconfigurable logic for on-chip coprocessors [6, 7, 8, 10, 11]. They have shown that such logic can speed up many applications. An on-chip coprocessor improves the performance of the applications and reduces the bottleneck of off-chip communications. In Garp architecture [6], programmable logic resides on the processor-chip to accelerate some computations. The frequently used computations are mapped to the programmable logic. If an application does not need the logic, these functions remain idle. PipeRench [30] reconfigures the hardware every cycle to overcome the limitation of hardware resources.

Xilinx Virtex FPGA family [31] allows partial reconfiguration. However, the dynamic partial reconfiguration can only be done at the granularity of a configurable logic block consisting of four 4-input look-up tables (described in Section 2.1). An advantage of this architecture is that a number of smaller configuration memory blocks can be combined to obtain a larger memory. However, a fine-grained memory cannot be synthesized efficiently in terms of area and time. In particular, providing a large number of decoders for small chunks of memory is expensive.

These observations motivate ABC to use a reconfigurable module which works as a function unit as well as a cache memory. Our goal is to develop such a reconfigurable cache/function unit module and integrate it into the existing microarchitecture to improve the overall performance with low area and time overhead. The reconfigurable cache/function unit module can be implemented using multi-bit output LUTs in the cache memory array, which is similar to FPGA-like logic. The expectation is that significant logic sharing between the cache and function unit would lead to relatively low logic overhead for an RFC. If the area overhead of an RFC exceeds the area of the dedicated logic for that function, or if the time overhead of cache is significant (if the time increases more than 5% – 10% - commonly treated as a significant increase), this is too big a compromise.

We integrate the RFC into a RISC superscalar microprocessor to build a computer which uses Adaptive Balanced Computing. To implement the RFC in the existing cache structure, we use two types of cache organizations, a multiple-set associative cache and a cache memory further partitioned into sub-cache blocks. Some of these cache blocks can be configured as specialized computing units. This will reduce the cache memory capacity when such a reconfiguration is in effect. However, we propose to organize the cache partition in such a way that the performance penalty is minimal. The entire cache organization with the RFCs is described in Section 5.2.2. Using various cache mapping organizations, we study the overall impact on the performance of selected benchmarks, such as multimedia and DSP applications.

There are some potential shortcomings with integrating RFC into a conventional microprocessor. Reconfiguring resources on-the-fly may affect the existing architectural behavior. For example, if a portion of the cache is converted into a functional unit, it increases the miss rate due to the smaller cache size. It may also require significant context switching due to the cache sharing between configurations and regular program data. However, we are targeting applications in which a large cache is not used effectively (e.g. multimedia applications). This implies that the smaller cache size by the reconfiguration does not affect the overall cache behavior significantly. In addition, our microarchitecture does not require any large amount of context switching. Loading configuration data initially is the only context switching required

when the RFC is configured as a computing unit. Other small amounts of configuration data loaded partially at run-time are not likely to be an outstanding problem. By using the existing microarchitecture, the integration of RFC does not modify the entire architecture significantly. In cache design, the size of cache memory increases due to the additional logic and interconnections. For high performance computing, additional chip area would not be a significant problem in the microprocessor design. The increased cache area may increase the cache access time in a conventional cache. In another cache model, the proposed new cache structure may decrease the time. The estimations for both models are presented in Section 4.4.

1.4 Thesis organization

The background and related work to our research is described in Chapter 2. In Chapter 3, we describe the problem statement of our work. Chapter 4 describes the architecture and design of a reconfigurable functional cache (RFC) with the functional unit and cache operations with multi-bit output LUTs. The ABC microarchitecture is described in Chapter 5. Chapter 6 presents examples of functions (Multiply-and-Accumulator and Distributed Arithmetic) mapped to the RFC. In Chapter 7, we present the performance results of our architecture. Finally, we conclude the thesis in Chapter 8.

CHAPTER 2. BACKGROUND

To better understand the problem presented in this thesis, we describe the characteristics and architectures for FPGAs, reconfigurable logic in a processor, memories with computation, SIMD extension in microprocessors, and a conventional cache memory structure in Section 2.1, 2.2, 2.3, 2.4, 2.5, respectively. Characteristics of cache memory in media applications is also presented. A typical superscalar microprocessor is described in Section 2.6. Finally, we discuss about the trends of future microprocessor in Section 2.7.

2.1 FPGAs

We use Xilinx [32] terminology in describing an FPGA (Field Programmable Gate Array) architecture. FPGAs can be viewed as a two-dimensional array of CLBs (Configurable Logic Blocks) - CLB is a primitive PE (programmable element) - with interspersed routing channels as shown in Figure 2.1. Each CLB consists of configurable gates realized through LUTs (Look-Up Tables). The CLBs are connected and communicate through flip-flops and programmable interconnections implemented using programmable pass gates and multiplexors.

LUTs are the essential component to implement any logic function on FPGAs. An n input LUT depicted in Figure 2.2 can represent 2^{2^n} number of functions. An LUT usually has four inputs and one output out of an SRAM-based memory to keep the overall operation and routing efficient [33]. However, the one-bit output granularity of each LUT results in a large interconnect area - even larger than the area of LUTs - and delay due to a number of switches for the programmability [34].

Using these programmable elements (PEs), user-defined functions can be implemented on FPGAs. Initial configuration is done by loading all the necessary configuration data into the

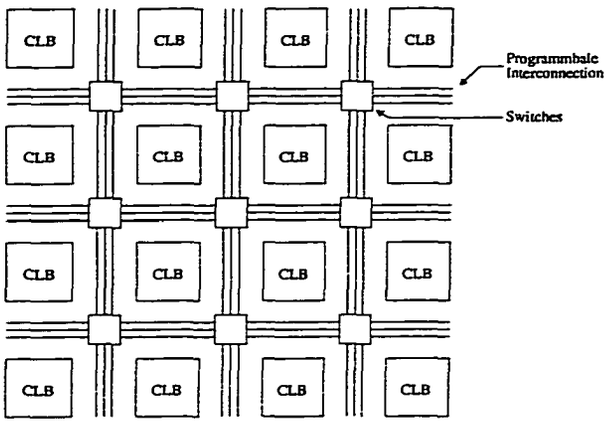


Figure 2.1 FAGA structure

PEs and configuring the programmable interconnection. Once configured, it performs the static function until reloading other configurations. In addition, configuration and computation cannot be performed concurrently due to technology difficulty between configuring and computing on FPGAs. The configuration of Xilinx Virtex FPGAs [31] is processed in three phases. First, the configuration memory is cleared. Then, the configuration data is loaded. Finally, the logic is activated by a start-up process. They also support readback of the contents for all the flip-flops/latches along with the configuration data in the configuration memory for verification and real-time debugging.

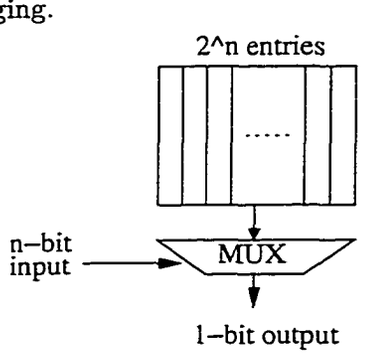


Figure 2.2 An n input Look-Up Table (LUT) with one-bit output

FPGAs map and execute virtually any kind of applications by writing the contents of LUTs based on the applications to represent various logic functions and configuring programmable interconnection to propagate data/control signals. Most of the common applications performed in FPGAs are DSP applications [35, 36].

2.2 Integration of processor and reconfigurable logic

Several researchers have investigated the issues associated with coupling processors and reconfigurable logic on a single die [12]. One motivation behind the idea is to give additional resources for computing-bound programs. Most of these approaches are targeting embedded and compute-intensive applications, such as video/audio processing, DSP, encryption, sequence matching applications, etc. in on-chip processor. In addition to the additional physical hardware, the architectural integration of reconfigurable logic into processors may require either a new instruction set (compiler-driven) or hardware/software co-design to exploit the logic.

2.2.1 Garp Architecture: Reconfigurable logic in a processor

Garp [6] architecture extended to MIPS-II instruction set incorporates an on-chip programmable logic - array of logic blocks. The reconfigurable array consists of control and logic blocks. The basic quantity of data in the array is 2 bits. Each logic block similar to CLBs in Xilinx 4000 series [37] can implement a function of up to four 2-bit inputs. The arrays connected through local/global wires vertically and horizontally carry the 2-bit quantities grouped in pairs. The loading and execution of array configuration is under the control of the main processor.

The logic block is controlled by the processor with a number of new instructions to configure the reconfigurable array as well as to move the data between the array and the processor's own registers. Garp also allows partial array configurations at a minimum of one row. The reconfigurable logic also has direct access to main memory through additional data buses between the logic and off-chip memory. Various structured computations are mapped and executed in the programmable logic and interconnection, but there are no resource reconfigurations.

2.2.2 DPGA-coupled microprocessor

DPGAs (Dynamically Programmable Gate Arrays) have constructed a hybrid architecture of FPGAs and SIMD arrays by reconfiguring cached configurations and performing different operations simultaneously, respectively [38, 39]. Array elements consist of 4-LUTs with a

4-context memory using DRAM depicted in Figure 2.3. The multi-context of configuration data is provided by on-chip memory (context DRAM) for multiple array functionalities with high/local on-chip bandwidth to reconfigure rapidly among different computations in each array element. 4×4 array elements are grouped into a sub-array with local interconnects. Each sub-array is connected and routed via global crossbar interconnections.

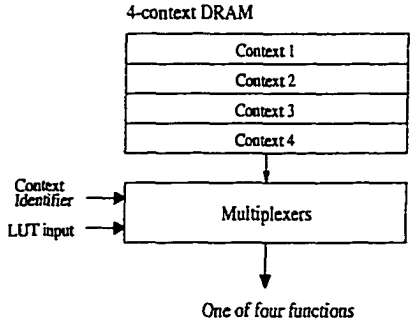


Figure 2.3 4-LUT with 4-context DRAM

The computational power and flexibility of DPGA allow conventional microprocessors to accelerate compute-intensive and special-purpose applications through a coprocessor [7]. The integration of DPGA into microprocessors requires additional instructions for the computational cooperation and communications between the processor core (fixed logic) and reconfigurable logic. Pre-defined subroutines for the reconfigurable logic - which would be assisted by hardware synthesis tools - would be used in high-level programming languages as library routines for the acceleration of applications. Tightly coupled DPGA processing arrays - reconfigurable logic - reduce the limitations of the communication between the processor and reconfigurable logic as well as the overheads for the reconfiguration.

2.2.3 ConCISe

ConCISe proposes a smart compilation chain with a hardware synthesis tool which generates application-specific custom instructions to support CPLD-based RFU (Reconfigurable Functional Unit) in RISC micro-architecture [40]. Thus, the RFU implemented using PAL

(Programmable Array Logic) and PLA(Programmable Logic Array) executes the different customized instructions generated at compile-time.

In the microarchitecture, the RFU is placed in the execution stage of a standard RISC pipeline to provide an extra functional unit. The custom instructions to be executed in the RFU contain the RFU configuration for decoding a specific configuration and register numbers for the register-register operations. The main features of the microarchitecture are as follows. It avoids partial reconfiguration to reduce the reconfiguration latency by adding more resources. It also minimizes logic complexity and optimizes resource utilization at compile-time with a logic synthesis tool. The compilation chain makes it easier for the application programmers to exploit the RFU in their applications.

The compilation chain reduces/eliminates the reconfiguration overheads by encoding multiple custom instructions in a single RFU configuration. First, it detects/selects the data-flow sequences mapped to RFU potentially in a conventional programming code. The possible candidates are limited to arithmetic and logic instructions. Then, a translator converts the candidates into hardware description language. A logic synthesis tool verifies the timing and the possibility of mapping the function. This procedure is repeated many times to find suitable functions to the RFU. It also generates the corresponding configuration data. Finally, assemble/link and generate the executable. This procedure generates no change in a conventional program design flow.

2.3 Memory systems with computations

Various memory systems enhanced for computations to overcome the performance gap between processor and memory systems have been presented in the last decade. Most of the computational memory system models attach logic to a conventional memory system (particularly for DRAM) for a faster and easier communication between computing unit and memory system referred to as Merged DRAM Logic (MDL) unlike Intelligent Memory (IRAM) [24, 25]. IRAM integrates only Dynamic RAM (DRAM) into on-chip microprocessor to reduce the off-chip bottleneck. This also gives a distribution of tasks by off-loading data-intensive applications

from the core processor to the memory systems. Another possible direction is to reconfigure memory itself for the resource utilization. One approach is presented in [42], which uses part of cache memory for different purposes of memory (such as buffers and look-up tables) on demand from applications. Reconfigurable cache addressing [43] is developed to provide data to computations without thrashing cache blocks.

2.3.1 Active pages

Active Pages [22] is a computation model which shifts data-intensive applications into the memory system. An implementation of Active Pages on RADram (Reconfigurable Architecture DRAM) is based on the integration of reconfigurable logic (FPGA-like logic) with DRAM to keep the processor at peak speeds by off-loading applications to the memory system. To achieve the proper transfer of computations between processor and memory system, Active Pages partitions an application into processor-centric and memory-centric tasks. The processor-centric partitioning is for complex computations while the memory-centric is for for data manipulation and integer arithmetic.

In the integration with a microprocessor, the interface to Active Pages is similar to a conventional virtual memory system. The processor controls Active Pages and communicates with them through a memory reference-like functions (like a series of memory-mapped operations) in a code sequence to write/read operands and results, set the particular pages, allocate/bind the group of pages for inter-page references. They are also synchronized by issuing synchronization variables.

Active Pages can exploit high parallelism by executing applications in both processor and memory systems simultaneously. This is accomplished by loading simple, application-specific operations in the memory system. One problem is the fabrication of Merged DRAM Logic (MDL). This integration may result in the performance degradation of logic and poor density of DRAM.

2.3.2 FlexRAM

FlexRAM architecture [23] places simple-compute engines in DRAM arrays to use as general purpose processing unit or otherwise as plain DRAM. In addition, to control and increase the usability of those engines a narrow-issue superscalar RISC core with small instruction and data caches is included on each memory chip based on Processor-In-Memory [41]. The RISC core also coordinates the compute engines with the host processor. Each chip has 64 memory arrays, which contain their own 32-bit fixed-point RISC engine.

To initiate tasks in FlexRAM, the host processor should send a signal to the small RISC processors with a write to a special memory-mapped location. The host also passes the address of the routines to be executed in the memory system. The FlexRAM processor in memory informs the host processor the completion of the tasks. The host processor and the FlexRAM processor share the virtual memory. For the inter-chip network, each FlexRAM chip communicates with other chips through an additional interconnection controlled by each FlexRAM processor.

2.3.3 Reconfigurable caches

Another type of reconfigurable cache design proposed in [42] enables cache SRAM arrays partitioned dynamically to be used for different processor activities that can benefit from extra resources instead of conventional cache memories. Some of the potential applications could use the partitions of reconfigurable cache as look-up tables/buffers for instruction reuse and hardware prefetching, or as compiler-controlled memory.

The new reconfigurable cache structure is achieved by partitioning (isolating) the physical data bank for one way out of a set associative cache memory. The partitions for the other activities and normal cache operations can be addressed and differentiated by multiplexing the corresponding addresses and signals. A special register called *cache status register* tracks the number and size of the partitions, and controls the signals for appropriate partitions. Overall, the new cache organization requires few modifications to a conventional cache design with a small increase on cache access time (less than 6%). The detection mechanism used

to reconfigure the cache memory in a code sequence can be software or hardware controlled. When reconfiguring, the current data in the cache is moved between partitions or written back to lower memory levels, called *cache scrubbing*. They expect the frequency of *cache scrubbing* to be low (only once at the start of applications).

2.3.4 Cache tiling

A new cache architecture for windowed image processing is developed as *cache tiling* in [43]. Processing data in large structuring elements in small caches maps input data to the same cache locations. This results in the repeated replacement of data in the same location in a cache, called trashing. Predictable memory access patterns for image processing are exploited to eliminate the cache trashing. This is done by the linearization of data accesses in memory. The data access linearization is achieved using a fast address translator to exchange the address bits in a cache address. This increases the cache efficiency. It eventually improves the overall execution time by reducing the number of memory accesses.

Cache tiling allows dramatic improvement in caching efficiency for small caches independent of compiler optimizations. Programs are not affected providing a transparent solution to improve caching. System code, compilers, or profiling programs can determine the blocking necessary for the best performance.

2.4 SIMD extension in microprocessor

The demand of multimedia-rich applications has been and will be dominating applications in PCs. Higher performance for these applications is preferred in general-purpose microprocessors, which results in architectural extensions. The media applications are compute-intensive with localized recurring loop operations involving small native data types. In addition, they have large working sets and are streaming applications, which need an efficient data caching mechanism. These applications will get more benefit from well-structured/specialized units and instructions.

This motivates a microarchitecture to incorporate SIMD (single-instruction, multiple-data)

extension to exploit the parallelism at instruction-level. To achieve instruction-level parallel execution, the microprocessor packs the small native data with at most 8 data elements for the same operation in one cycle with new media instructions (without special purpose processor or dedicated hardware). The common operations in SIMD media instructions are *arithmetic/logical operations, data movement/reorganization, and type conversion* with the packed data types in 8(one-byte), 4(word), 2(doubleword), and 1(quadword) data elements. The packed data operation is handled by saturating instructions which truncate the result in case of over/underflow. Since the multimedia data has less temporal locality, which implies that the data is processed once and discarded, it may cause the cache trashing problem. This motivates the use of special instructions to prefetch the data for faster processing and less interference with the cache behavior.

By sharing the existing processor resources, the SIMD extensions are integrated with a minimal microarchitecture modification and a small amount of additional units. For example, the media instructions share the FP-registers with the floating-point instructions in Intel MMX [44]. Thus, the advantage of SIMD extension is to accelerate the media applications on general-purpose microprocessor without the aid of special purpose processor or dedicated hardware. Another main factor of the SIMD extension is the compatibility with the existing Instruction Set Architecture (ISA) and Operating Systems.

There is a trade-off in the SIMD extension between embedded and exclusive architectural extension/support for the media applications. We describe the trade-off and difference in two microprocessors for the extension with respect to compatibility and additional resources in Section 2.4.1 for Intel MMX/SSE and Section 2.4.2 for PowerPC's AltiVec.

2.4.1 Intel Pentium MMX/SSE ISA Extension for Multimedia

Intel MMX/SSE [44, 45] has small modifications in the part of existing datapath and shares the common contexts - the core pipeline stages (datapath) - and switches the contexts between the SIMD extension and conventional architecture.

MMX: In MMX, no new states (such as register sets, control registers, and new condition codes) are added for the compatibility with existing IA (Intel Architecture) and OS. This makes the extension *embedded* to the Intel x86 architecture. The MMX media instructions share the existing 64-bit FP (Floating Point) register set as the MMX registers with FP instructions for the compatibility. The sharing restricts a code sequence to be partitioned into FP and MMX codes to make infrequent and simple context switching. Those two types of code should not be executed simultaneously. This results in the full IA compatibility. With the same techniques for FP interface to OS, such as FSAVE and FSTORE, the context switching is done by saving and restoring the data in the shared FP registers before executing FP/MMX data. The context switching from MMX to FP is performed by EMMS (empty MMX state) operation. Using FP reg tab bits (status of each register), FP data can be also protected from the MMX instructions.

As mentioned above, MMX supports parallel operations on multiple small packed data elements with 57 specialized and enhanced instructions for media applications. MMX also has similar instructions like other microprocessor's extension described above. For instance, it supports packed shift instructions to realign the misaligned memory access similar to AltiVec's permute instruction. Data transfer to memory for the packed data is done by the new special instructions (MOVQ: move quadword (64b), MOVD: move packed-word (32b)). MMX instructions use 8 MMX(FP) registers for packed data and 8 integer registers for loop control, addressing, etc..

The MMX instructions can be programmed using the assembly extension and optimized at compile-time. A code sequence may contain both conventional code and MMX code for applications to be executed conditionally on detection of MMX technology.

SSE: Streaming SIMD Extension (SSE) has additional features which supplement MMX technology. SSE adds SIMD-FP execution units and 70 new media instructions for floating-point computations in media applications. For the streaming nature of data access from/to memory in media applications, SSE employs prefetching and streaming load/store instructions. It reduces the cache misses by keeping only basic/common data through the entire data set

in a cache memory. The different contexts and resources from the MMX technology are as follows.

- new SIMD-FP instructions and media instructions.
- new conversion instructions: from SIMD-FP to MMX for packed data and from scalar-FP to IA-32 integer for scalar data.
- Adding a new state - SIMD-FP FU and eight 128-bit registers for SIMD-FP.

  This reduces the implementation complexity because of no sharing between contexts. There is no context switching between SIMD-FP and MMX/X87. Thus, SIMD-FP and MMX/X87 can be executed concurrently with separate exception handling. It also provides wide and fast FP resources. The 128-bit processing (4-wide micro-instruction) is accomplished using two 64-bit micro-instructions (2-wide micro-instruction) for the compatibility without 128-bit datapath.
- memory streaming instructions to prefetch instructions and wider bus systems to memory - which results in high memory throughput.
- decoupling memory prefetch from retirement of subsequent instructions for the concurrent execution of the computational stream and the prefetched memory access.

Other major microprocessor vendors have introduced a similar SIMD architectural extension, AMD 3DNOW! Technology [46], HP MAX-2 [47], and Sun Sparc's VIS [48] as well.

2.4.2 AltiVec in PowerPC

AltiVec [49] adds exclusive and independent datapaths, such as the 128-bit vector processing unit, from the existing microarchitecture to free up the core datapath for other conventional application sequence flows. This results in less context conflict/switching between the SIMD extension and conventional architecture compared to Intel's MMX/SSE. In addition, AltiVec allows x86 and SIMD extension instructions to be executed in parallel without interfering with each other. However, it occupies a larger portion of die area and needs more complicated

management due to two sub-states in one global state. The independent architectural support in AltiVec requires a different optimization algorithm at instruction-level from the conventional scalar ISA.

AltiVec implements fully pipelined functional units which have low latency - one cycle per computation in most cases. Simple and compound (merged with two or three instructions) operations can be performed by two issues per cycle for ALU-class and permute-class instructions. Other media instructions are similar to other SIMD extensions described in Section 2.4.1. AltiVec also minimizes the number of memory accesses using the special permute instruction for load/shift for misaligned memory accesses.

The architectural support for the entire range of multimedia processing in AltiVec is achieved through the following additions - a large vector register file, full-range data-type support, four operand non-destructive instruction format, permute capability, powerful SIMD instruction set, and enhanced and adapted data-prefetch streams to the media applications.

2.5 Cache memory

In Section 2.5.1, we explain how a cache memory works and is organized in architecture and design. This describes the basic concept of conventional cache memories, such as addressing, mapping, and policies. Then, we describe the characteristics of cache memory in media applications in Section 2.5.2.

2.5.1 Cache memory architecture and design

Cache is the first level of memory hierarchy in a microprocessor and a subset of lower memory hierarchy, such as main memory and hard disk. According to the principle of temporal and spatial locality, the cache memory contains data or instructions to be accessed by the processor activities in the near future. There are three types of cache organizations, *direct-mapped, set associative, and fully associative* caches. In a *direct mapped* organization, each data block in lower-level memories can be placed only into one location in the cache. If a block can be placed anywhere in the cache, the cache is *fully associative.* If a block of lower-level

memories can be placed in a n number of locations in the cache, it is referred to as n-way set associative. Direct-mapped cache is just 1-way set associative and fully associative cache is n-way associative for n blocks. Since a cache memory maps a number of blocks in lower-level memories - more than the number of blocks in the cache, the blocks to be placed into a same location need to be identified. Each block is identified by an address tag, which is a part of block address. The tag is the number of block frames in a lower-level memory partitioned by the number of sets of cache memory (in direct mapped or one way). Another part of block address is an *index* that is used to select an appropriate set. The remaining part of an address is the *block offset* which can point out the desired data within a block. When a miss occurs, an appropriate block in the cache will be replaced by the replacement policy, such as random or Least-Recently-Used (LRU). Data can be written to a cache memory in two policies, write-through or write-back. The write-through policy is commonly used in a multi-processor system for the memory consistency while the write-back policy is used in a uniprocessor system to reduce the off-chip memory traffic.

There are three categories of the causes of cache misses : compulsory, capacity and conflict. The first access of data will always be missed, called compulsory or first reference misses. This may often occur in a large cache memory due to the misses of all the blocks in the cache at the very beginning. A cache memory cannot contain all data in lower memories due to the lack of capacity. This results in blocks of data to be mapped to the same location in a cache memory. This conflict requires a replacement of block. If too many blocks are mapped to the same location, for example low-associativity, more conflict misses occur.

Conventional cache memory consists of an array of SRAM-based memory cells, row/column decoders, multiplexers, sense amplifiers, and comparators depicted in Figure 2.4. Data and tag memory banks have the same organization described above with a different column size. The arrays consist of the storage cells (SRAM cell implemented using two cross-coupled inverters), horizontal wordlines, and vertical bitlines. A cache memory read operates as follows. The index and the block offset of an address are propagated to the row decoder and the column decoder, respectively. One signal from the row decoder selects one cell-row in the arrays for

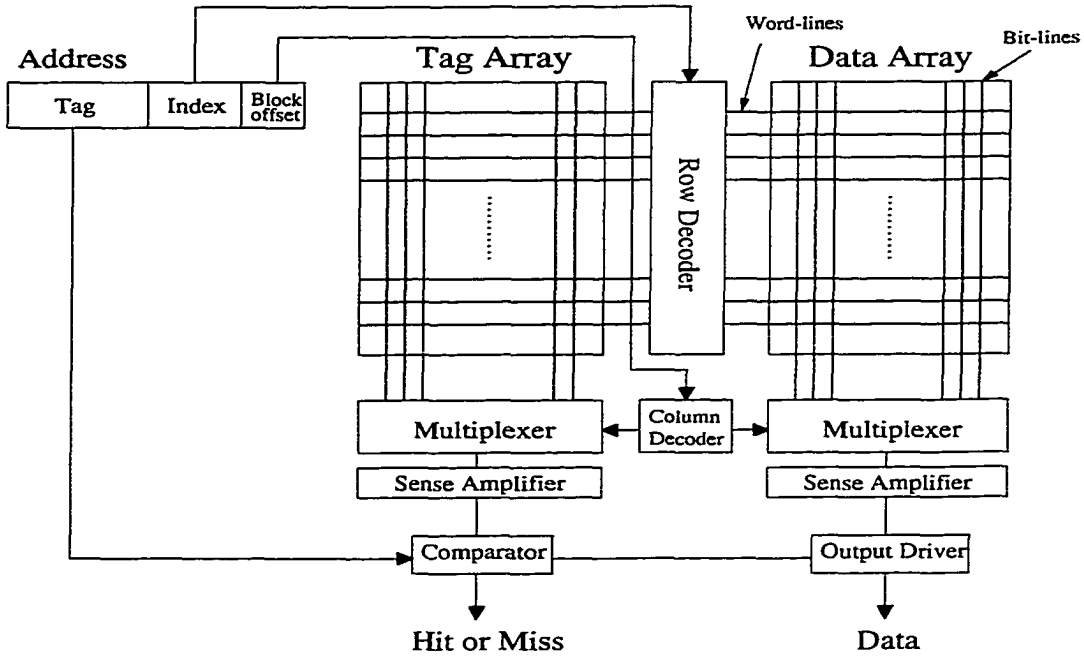


Figure 2.4 Conventional Cache memory structure

tag and data. The selected block is transfered through the bitlines. A desired data word in the block is selected by the column decoder through the multiplexer. For a faster propagation of data from the array, sense amplifiers are used at the end of multiplexers. The tag part is compared with the data from the tag array to check if the memory access is a hit or miss. If a hit occurs, the data is passed to the processor core. If a miss, the address is passed to the lower-level memory and the miss signal is issued. Therefore, there are several components which determine the cache access time - decoder, wordline, bitline, column multiplexor, sense amplifier, comparator, and select data output delays - as modeled by CACTI [50]. The access time of tag array is often higher than that of data array due to the additional tag comparison.

A cache memory can work as a large number of input LUTs (like CLBs in FPGAs described in Section 2.1), for example, the addresses as input and the data words as logic functions. However, only one logic function is implemented in the cache memory because the LUT is controlled by one set of input-bits. While one large LUT could represent multiple stages of logic as one memory read by synthesizing a number of small LUTs (logic), it requires more effort from logic synthesis (and makes it less effective).

2.5.2 Cache memory characteristics in media applications

Experiment results on cache effectiveness for media applications in [19, 20, 21, 43] show that a larger cache size hardly increases the performance in selected media applications due to streaming data nature and low temporal locality. However, data in media applications has higher spatial locality. In addition, all the data of multimedia applications cannot fit into a cache memory because the working sets of these applications are very large and are accessed in a streamed fashion. This directs the microprocessor to prefetch the streaming data without losing frequently used data in the cache [45]. Instead of caching the data on a processor, streaming load and store are more convenient and faster than the conventional caching strategy since streaming operations can remove the complicated cache operations for the memory hierarchy consistency. This results in a media processor to be a kind of vector processor.

2.6 Superscalar microprocessor

Superscalar microarchitectures [52, 53] issue and execute multiple instructions every cycle to exploit the instruction-level parallelism. An additional feature in a superscalar microprocessor is an out-of-order issue and execution by resolving the control and data dependencies. A typical superscalar microprocessor depicted in Figure 2.5 operates as follows. Multiple instructions are fetched from the instruction cache every cycle in the fetch stage and then decoded in the dispatch stage. The artificial data dependency, such as write-after-write and read-after-write, is checked and resolved by the renaming registers in the dispatch stage and the data forwarding mechanism between the stages. The instructions are dispatched to the issuing reservation stations. In the centralized/distributed reservation stations, the instructions are issued, then executed in a functional unit and, depending upon the operation of instructions, may access the memory. In the write-back stage, the instructions update the computed or loaded data into the register file. The issue and execute can be done in out-of-order; however, the instructions are committed in-order in the commit stage to avoid wrong operations in the instruction stream by mis-speculation (for example, by branch mis-prediction) and to support precise exception handling.

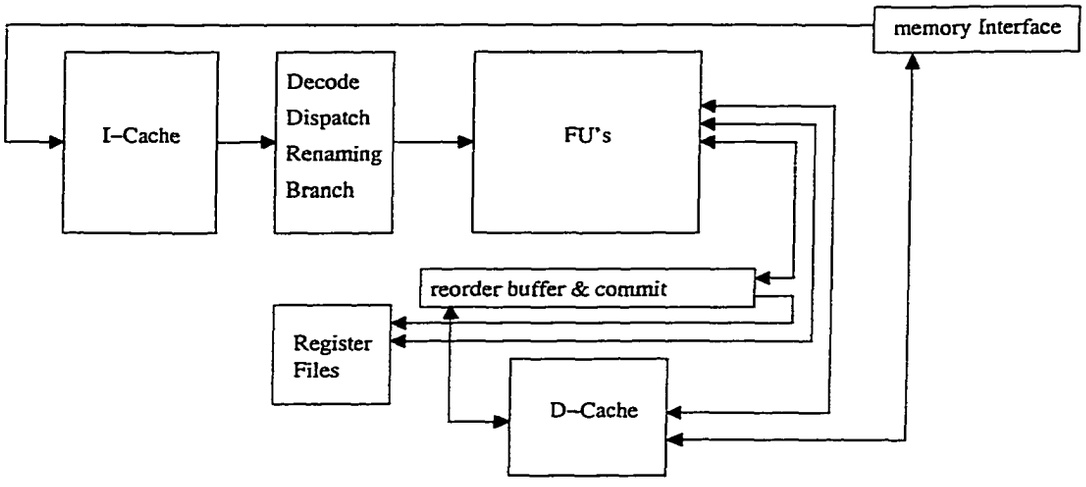


Figure 2.5 A typical superscalar microprocessor

2.7 Trends of future microprocessors

In deep-submicron technology, we can have billions of transistors running at gigahertz frequencies [14, 15, 16]. This trend promises a lot of challenges and possibilities in microprocessor architecture and design. However, certain complexities and constraints inherent to deep-submicron technology will limit the performance of future microprocessors accordingly.

Microprocessor performance has been improved mainly by technology scaling. As the feature size scales down, the device sizes at the transistor-level shrinks as well. This allows for more devices and a smaller clock cycle time on a chip. However, the interconnect delay, especially, for a global interconnect, does not scale down as much as the gate (device) delay because it remains unchanged due to the RC (resistance and capacitance) time constant (i.e., as feature size reduces, the capacitance of the wires also reduces, but the resistance increases due to the narrower width and thinner height in the wire.). This is the cause of the main physical constraint in microprocessor design for high performance. In addition, the power and cost will grow as the number of gates increases. Another behavioral constraint is program control and data dependencies in a code sequence, for example, branches and WAR (write-after-read) data dependencies, (even though more resources are available on the chip).

One temporary solution to the interconnect delay is to make the height of interconnect thicker to reduce the resistance and use copper metalization. However, this will not be sufficient in extremely small feature sizes. It also causes cross talk between adjacent wires, which increases wire capacitance and power consumption since they are very close due to feature size shrinkage. In the previous technology, we have used the RC time model to identify the characteristics of timing problems in design. Now, we need to consider inductance (electromagnetics) that is caused by the very narrow interconnect, which results in the RLC model. Due to the interconnect delay, the signal drive region (in which signal propagation can be reached within one clock cycle) will have less number of gates and relatively smaller area than larger feature size technology. For example, only 16% of die area (in billion transistors) could be reached within one cycle period (at 1.2 GHz) at $0.1\mu m$ technology [26]. The impact on VLSI design is that the interconnect delay and complexity will be the dominating factor for delay, which motivates a new design paradigm and flow of microprocessor, such as interconnect-driven design. For example, we need to avoid the congestion of wires in critical paths. This mismatch between gates and interconnects gives a gap between gate delay and propagation delay via interconnects. To match the wire delay to the faster gate delay, more driver buffers between segments of wire are necessary. For instance, long distance travel of a signal requires a large number of buffers and registers. However, adding more buffers and registers may not be a solution since the region (or the number of gates) that can be reached in one CPU clock cycle would be smaller and smaller as the feature size decreases. This is another factor which architecture needs to account for.

The smaller feature size reduces the clock cycle time by reducing the corresponding capacitance, which gives higher performance. Since the gate (logic) delay largely decreases, the clock overhead - setup time, clock to output delay, and clock skew - takes a significant fraction of the cycle time. Therefore, a very careful clock distribution across a chip and other circuit techniques to improve cycle times are necessary. Here are some of the possible techniques to improve the cycle time and reduce the effect of clock overhead: sense-amplifier-based, hybrid latch, semi-dynamic FFs, asynchronous logic to eliminate the global clock dependency, and

dynamic logic to reduce and hide clock overhead with higher area/power.

A deeply pipelined microarchitecture may not produce high performance due to the significant fraction of clock overhead in pipeline cycle time. In addition to the clock overhead, the unexpected control dependencies and exceptions in a code may nullify the high throughput of pipeline by flushing all the current pipeline stages and executing all of them again. To increase the ILP (Instruction-Level Parallelism), we may increase the instruction window to be issued in one cycle. However, this also generates an additional overhead. For example, a larger instruction window needs more time to check and verify the content/name of registers and the result of execution in the renaming stage in a superscalar microprocessor. Both deeper pipeline and wider instruction window for higher ILP do not produce higher performance linearly. Therefore, this requires an optimal number of pipeline stages and instruction window size.

The capacity of computation and memory in a microprocessor is increasing dramatically in deep-submicron technology. However, this trend may restrict a highly localized placement of resources on a chip to reduce the interconnect delay and power dissipation because of the small signal drive region. As we observed above, the deeper pipeline and larger instruction window specially in deep-submicron era do not produce high performance due to the significant time/area overhead to support such a microarchitecture. This implies that a centralized processor control and execution may not be a good solution because it adds a huge interconnect delay. This indicates that the "locality" of complex architectures distributed over the die is an important design factor.

CHAPTER 3. PROBLEM STATEMENT

Single-Instruction Multiple-Data (SIMD) extensions for multimedia applications are incorporated in conventional microarchitecture with a small amount of architectural change and design modification. Thus, SIMD extensions integrate the SIMD parallel computation model at instruction-level to the current general-purpose processor. However, the main restriction of SIMD extensions are the compatibility with the existing Instruction Set Architecture (ISA) and Operating Systems (OS). This causes significant context switching and compiler assistance. Another problem is the balance between computing and memory bandwidth. To match the balance, a streamed load/store of data from/to memory is employed. Moreover, the additional resources to support the media instructions should be as minimal as possible for effective use of die size and removing frequency impact [45, 49].

This kind of fine-grained architectural support (instruction-level) for multimedia applications needs less physical design effort and is easier to integrate into existing architecture. However, the fine-grained support may require low level optimizations at the instruction-level to exploit the benefits. For instance, a great deal of coding optimizations and significant compiler assistance may be required. It also needs unnecessary architectural support because of the compatibility between the media instructions and the conventional instructions as described above. Due to the fine-grained support requirement, the control/execution units for the SIMD extensions may need to be centralized to the core processor.

These shortcomings of fine-grained architectural support motivate exploration of coarse-grained architectural support (function-level) for media applications. For example, we may add function-level resources like ASICs to accelerate those applications. However, coarse-grained support requires highly customized units which are not cost-effective in a general-purpose

microprocessor. Instead, we have developed a dynamically reconfigurable functional cache (RFC), which works as memory or as LUT-based computing units. The RFC can support the coarse-grained architecture feature by mapping and executing an entire function with small additional logic and modifications in the existing architecture and design. One problem, however, is that the compile time optimization would have to operate at function-level. For example, we may not control or modify the function's behavior freely if the computing unit is already specialized to the functions. Conversely, the coarse-grained support could simplify coding and optimization for programmers and afford specialized/customized computing units for media applications in a general-purpose microprocessor. The simple compilation shown in Section 5.2.5 can be realized with pre-defined function calls in high-level languages to be performed by slightly modified load/store instructions. More pre-defined function calls may be added in the future through the co-design of hardware and software. Moreover, highly structured and intensive computation could be decoupled from the core processor to reduce the overheads of centralized architecture and design described in Section 2.7.

SIMD multimedia applications with large working streamed data sets, in which data is used once and then discarded [45], can be accelerated by a specialized computing unit. A larger on-chip cache hardly helps these applications due to the streaming nature and lack of temporal locality as mentioned earlier. Since SIMD applications need less reconfiguration at run-time (by the nature of SIMD), the run-time reconfiguration does not affect the overall execution time significantly once we configure the RFC as a function unit. The Multiply-and-Accumulation (MAC - core of FIR) and ROM-based Distributed Arithmetic (DA core of DCT/IDCT) functions are good examples of such SIMD applications. Such structured computations are more easily targeted for a reconfigurable functional cache especially within the low area and time overhead constraints. The LUT-based computing unit can be organized with unique and common configuration data. For instance, different types of computation using MAC and DA can be mapped to the RFC by changing contents of LUTs.

In this thesis, we propose a microarchitecture as a combination of RISC-type and CISC-type microarchitecture to support Adaptive Balanced Computing (ABC) through exploiting

Reconfigurable Functional Caches (RFCs). The basic concept of this combination is as follows. The interface and instructions are as simple as a RISC-type architecture while the operation is performed in multiple processes as a CISC-type architecture in a dense format of instruction. Thus, in the proposed architecture, the actual computation, which is initiated by the RISC-type instruction, is processed using specialized computing units as common primitives for DSP and multimedia applications. On demand from the applications, we simply add more functions to a conventional processor with with minimal amount of additional logic and time penalty. Unlike the SIMD extensions, the ABC microarchitecture involves minimal context switching between conventional instructions and the instructions supporting RFCs with a little modification of compiler and existing programs. This results in the combination of RISC-type instruction set with CISC-type computations. The proposed microarchitecture may also form a basis for a dynamic distributed microarchitecture.

CHAPTER 4. RECONFIGURABLE FUNCTIONAL CACHE (RFC)

In this section, we describe how the proposed reconfigurable cache module architecture (RCMA) is organized and how it works. First, we introduce multi-bit output LUTs to be used in the reconfigurable functional cache (RFC) in Section 4.1. Second, we show constant coefficient multipliers using multi-bit output LUTs in Section 4.2. Third, we describe the core design of RFC architecture, such as how to partition the memory blocks and connect them, and how it operates as a cache memory and a special function unit in Section 4.3. In Section 4.4, we compare and estimate the cache access time of RFC with respect to memory cell array cache (with one memory cell array) and base array cache (with a number of partitioned memory cell arrays of a conventional cache structure for a faster access time). The configuration and scheduling of the module are described in Section 4.5.

4.1 Multi-bit output LUTs

In most FPGA architectures, a Look-up table (LUT) usually has four inputs and one output to keep the overall operation and routing efficient [33]. However, an SRAM-based single output LUT does not fit well with a cache memory architecture because of a large area overhead for the decoders in a cache with a large memory block size. Instead of using a single output LUT, we propose to use a structure with multi-bit output LUTs. Such LUTs produce multiple output bits for a single combination of inputs and are better suited for a cache than the single output LUTs. Since a multi-bit output LUT has the same inputs for all output bits, it is less flexible in implementing functions. However, this is rather inconsequential for our problem domain. A 2-bit carry select adder and a 2-bit multiplier or a 4×2 constant coefficient multiplier (all need the same size, up to 6-bit output, of LUT) are depicted in Figure 4.1(a) and (b), respectively.

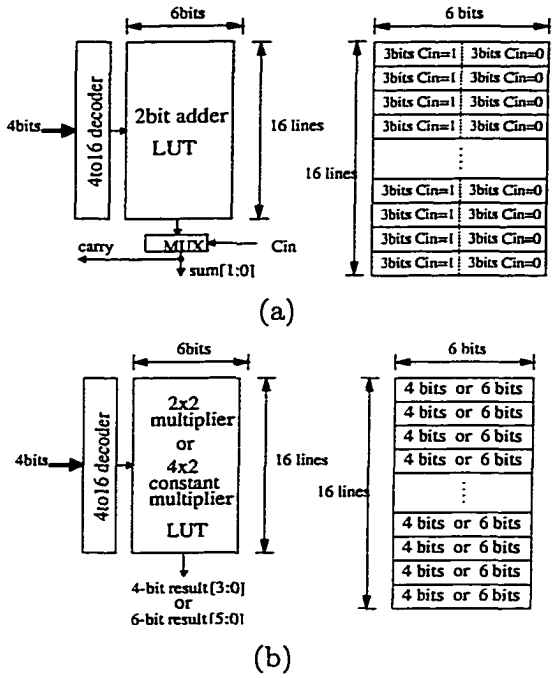


Figure 4.1 Multi-output LUTs : (a) A 2-bit adder ; (b) A 2x2 or a 4x2 constant coefficient multiplier

If a multi-bit output LUT is large enough for a computation, no interconnection (for example, to propagate a carry for an adder) may be required since all possible outputs can be stored into the large memory. In addition, unlike a single output LUT, a multi-bit LUT requires only one decoder or a multiplexer with multiple inputs. Thus, the area for decoders reduces. However, the overall memory requirement to realize a function increases. The required memory size increases exponentially with the number of inputs. Therefore, multi-bit LUTs may not be area-efficient in all situations. Also, in this case, the computing time may not reduce much due to the complex memory block and the increased capacitance on long bit lines for reading.

Instead of using one large LUT, we show implementations of an 8-bit adder with a number of smaller multi-bit output LUTs in Figure 4.1. Figure 4.2(a) depicts an 8-bit adder consisting of two 9-input LUTs. Each 9-LUT has two 4-bit inputs, one 1-bit carry in, and a 5-bit output for a 4-bit addition. Thus, total memory requirement is $2 \times 2^9 \times 5 = 5120\ bits$. The carry is propagated to the next 9-LUT after the previous 4-bit addition in one LUT is completed (i.e.

a ripple carry). Since each LUT must be read sequentially, this adder takes a longer time to finish an addition. By employing the concept of carry select adder as depicted in Figure 4.2(b), a faster adder using 8-LUTs can be realized as the reading of the LUTs does not depend on the previous carry. In this case, the actual result of each 4-bit addition is selected using a carry propagation scheme. However, all the LUTs are read in parallel. The total time for the modified adder is the sum of the read time for one 8-LUT and the propagation time for two multiplexers. Thus, it is faster. This adder also requires the same amount of memory (i.e. $4 \times 2^8 \times 5 \ = \ 5120 \ bits$).

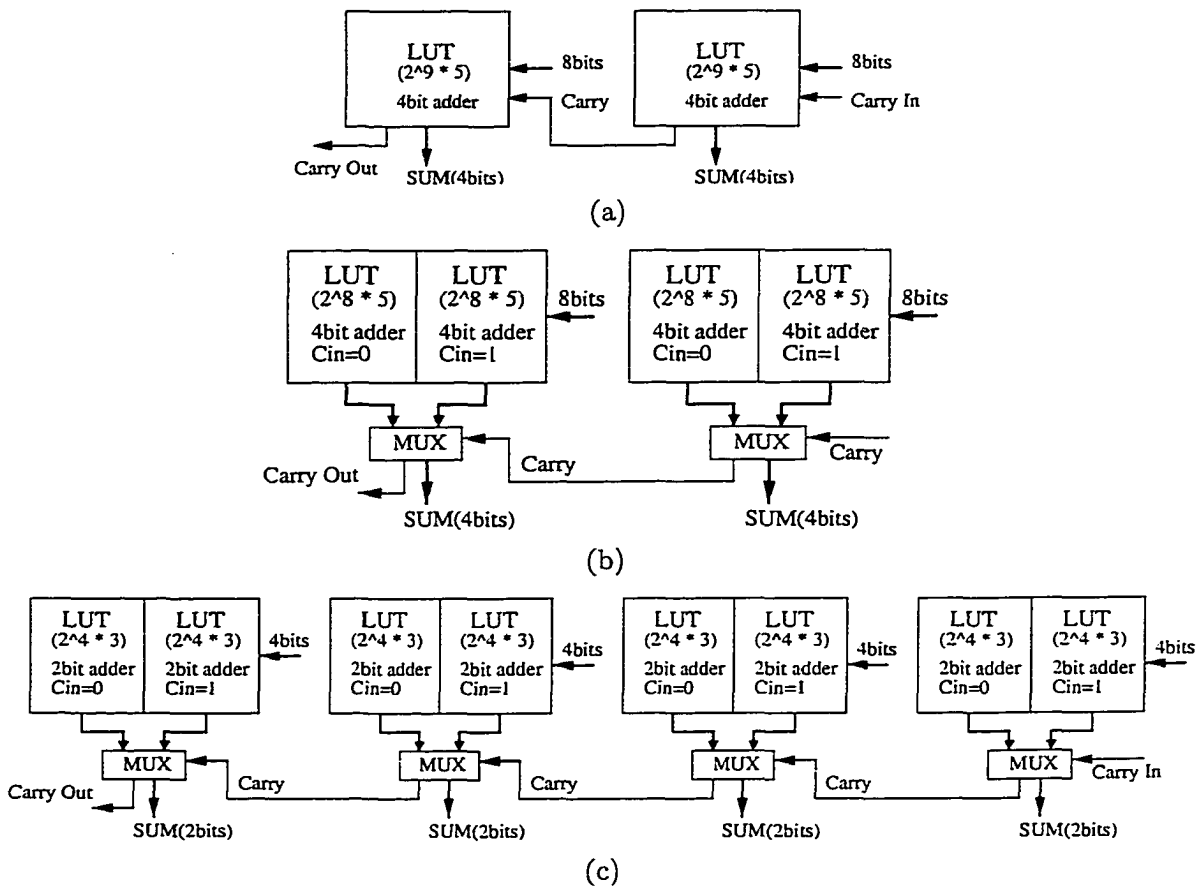


Figure 4.2 8bit adder using (a) two 9-LUTs ; (b) two 8-LUTs; (c) four 4-LUTs

To make an area efficient adder, a 4-LUT with 6-bit outputs can be employed (Figure 4.2(c)). The same carry propagation scheme as in Figure 4.2(b) is applied to the 4-LUTs to implement an 8-bit adder, but four 4-LUTs are used. The total time of the adder using the 4-LUTs might be higher than that using the 8-LUTs because it has twice the number of multiplexers to be propagated. However, the read time for a 4-LUT is faster than for an 8-LUT since it has a smaller decoder and shorter data lines for memory reading. We, therefore, recommend the design in Figure 4.2(c).

4.2 Constant coefficient multipliers using multi-bit output LUTs

An 8×8 multiplier is presented using single-bit output LUTs in [54]. As mentioned above, the implementation in single-bit output LUTs may require larger area for each dedicated decoder to LUTs and the interconnects between LUTs than multi-bit LUTs. Using the multi-bit output LUTs shown in the previous section, an area/time efficient constant coefficient multiplier can be implemented. Many memory cells reside on a cache memory. All of these memory cells can be converted to multi-output LUTs with an appropriate decoding scheme. Multi-bit output LUTs (4×8, 4×16, 4×32, 4×64, etc.) can reduce a significant amount of area by removing the interconnects between single-bit output LUTs and the decoders. Figure 4.3 shows a hierarchical structure of 16×16 constant coefficient multiplier using multi-bit output LUTs. The hierarchical constant coefficient multipliers using multi-bit output LUTs may be less flexible than single-output LUTs with respect to the programmability in functions. However, the area and time for larger constant coefficient multipliers using multi-bit LUTs is linearly scaled with the number of bits. The reason for the linear scale is the variable width of multi-bit LUTs, which implements a large constant coefficient multiplier in one LUT. For example, 4×8, 4×16, 4×32, and 4×64 multipliers are implemented in 4-input LUT with the corresponding width of output. Therefore, the multi-bit LUTs fits well to the conventional cache structure with minimal area and time overhead.

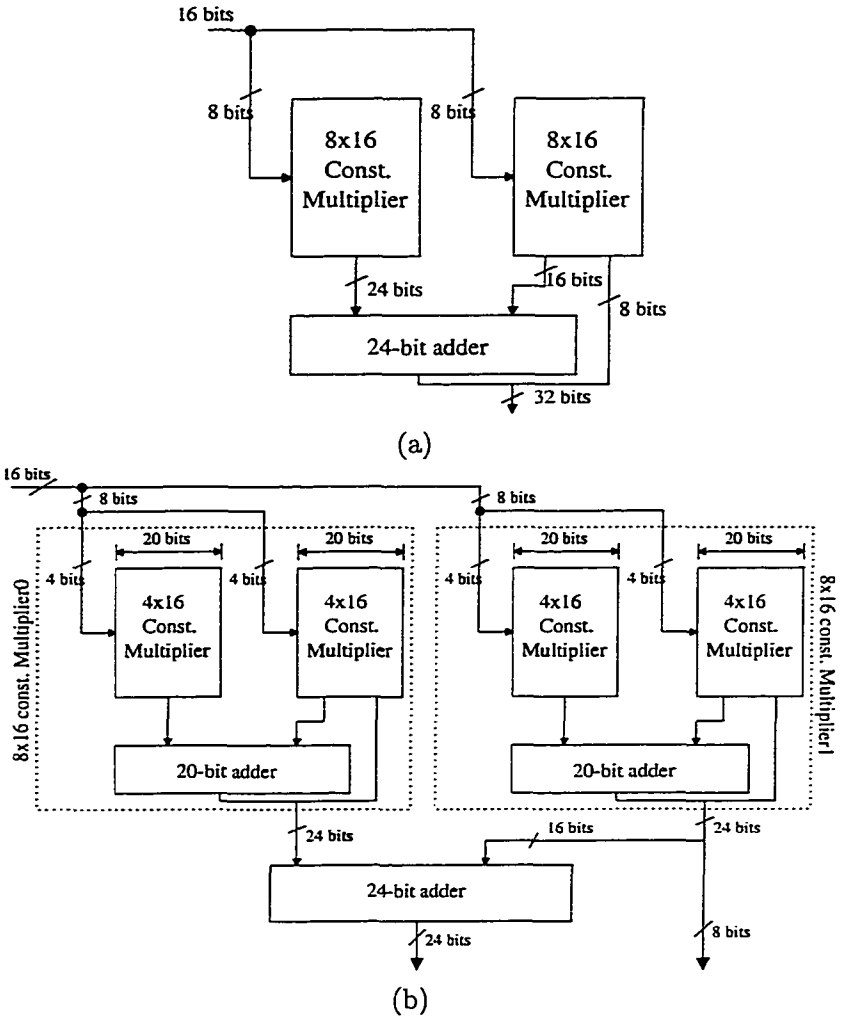


Figure 4.3 A 16×16 multiplier : (a) top level of 16×16 multiplier ; (b) hierarchical structure

4.3 Organization and operation of a reconfigurable cache module

Since we target compute-intensive applications with a regular structure, such as DSP and image applications (FIR, DCT/IDCT, Cjpeg, Mpeg, etc.) as mentioned in Section 1, we first partition them at coarse-level into repeated basic computations. A function in each level can be implemented using the multi-bit output LUTs as described in Section 4.1. We only add pipeline registers to each coarse-level stage, which contains a number of LUTs, to make the entire function unit efficient. All these registers are enabled by the same global clock.

Therefore, a number of coarse-level computations can be performed in a pipelined fashion.

Figure 4.4 shows a coarse template for a module. The cache can be viewed as a two-dimensional matrix of LUTs. Each LUT has 16 rows to support 4-LUT function and as many multi-bits in each row as required to implement a particular function. In the function unit mode - in which the RFC works as a special function unit, the output of each row of LUTs is manipulated to become inputs for the next row of LUTs in a pipelined fashion. In the cache memory mode - in which the RFC works as a conventional cache memory, the least significant 4 bits of the address lines are connected to the row decoders dedicated to each LUT. The rest of the address lines are connected to a decoder for the entire cache in the figure. In the cache memory mode, the LUTs take the 4-bit address as their inputs selected by the enable signal for the memory mode. Therefore, regardless of the value of the upper bits in the address, the dedicated row decoder selects a word line in each row of LUTs. This means one word is selected in each LUT row according to the least significant 4 bits.

Each LUT thus produces as many bits as the width of the LUT. These are local outputs of the LUTs. These outputs are available on the local bit lines of each LUT row. For a normal cache operation, one of the local outputs needs to become the global output of the cache. This selection is made based on the decoding of the remaining $(n-4)$ address bits decoded by the higher-bit decoder. The local outputs of the selected row of LUTs are connected to the global bit lines. The cache output is carried on the global bit lines as shown in Figure 4.4. Thus, output of any row of LUTs can be read/written as a memory block through global lines. We propose that these global lines be implemented using an additional metal layer. The global bit lines are the same as the bit lines in a normal cache.

Both decodings can be done in parallel. After a row is selected by both the decoders, one word is selected through a column decoder at the end of the global bit line as in a normal cache operation. In the figure, the tag part of a cache is not shown and a direct-mapped cache is assumed for the module. However, the concept of reconfigurable cache can be easily extended to any set-associativity cache because the tag logic is independent of the function unit's operations.

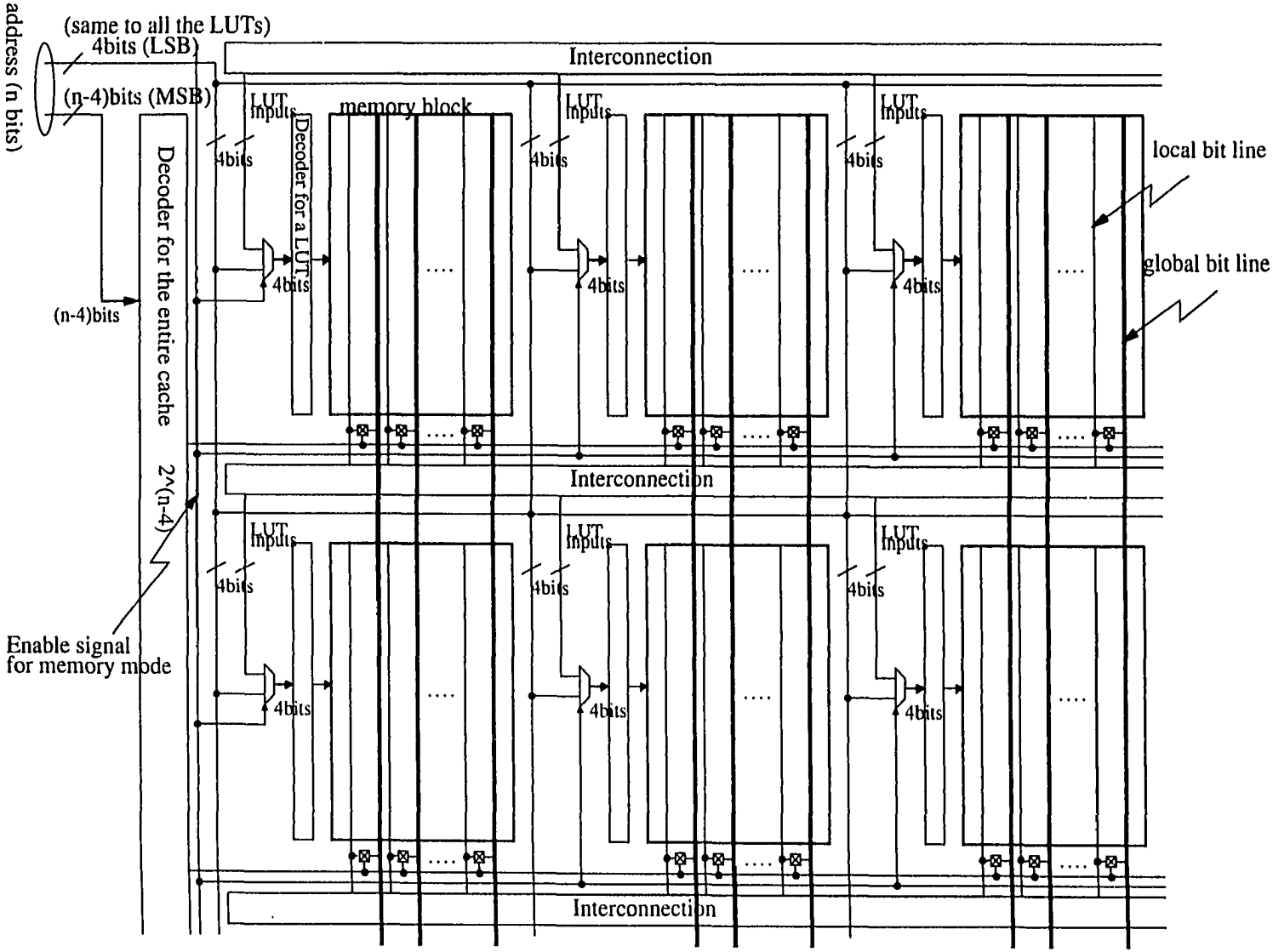


Figure 4.4 Cache architecture in the reconfigurable module

4.4 Access time for cache operations

We compare the access times for the reconfigurable functional cache (RFC) with the access time for a fixed cache module of comparable size. The base fixed cache module from which a reconfigurable cache is derived comes in two flavors. The first is a memory cell-only array cache with one address decoder and one data array. The second is a parallel decoding cache with segmented-bit lines and partitioned-word lines. The segmented-bit lines are divided every 16 cache blocks and enabled by the decoder for the high-order address bits with switches like the global bit lines in Figure 4.4. The partitioned-word lines are divided into the decoding lines from the the high-order address decoder and local word lines in a sub-memory blocks from each dedicated decoder in Figure 4.4. The local word lines select one block in every 16 cache blocks and one of them is selected by the high-order address decoder. The base array cache is shown in Figure 4.5.

The memory-cell only array cache has single-level decoding leading to low area and slow access time. An RFC based on this design reduces access time by introducing hierarchical decoding at a cost of large area overhead. A base array cache structure, however, already incorporates access time advantage of hierarchical decoding, and hence also needs more area. An RFC based on this design, hence, shows a slight degradation in access time with a very small area overhead. We analyze the RFC access time for cache operations in terms of address decoding time and word/bit line propagation time. Other components of access time, such as sense amplifier and column decoding, do not differ over the two cache organizations. The access times for an RFC based on a memory cell array cache and the base array cache are estimated below, respectively.

Memory cell array cache. The cache with the reconfigurable structure may have a faster address decoder than a memory cell array cache, which contains one main address decoder and a bunch of adjacent memory cells. Since each LUT, with its own row decoder for addressing in the reconfigurable module, is much smaller than a large synthesized memory cell array in a conventional cache, the decoding time of a LUT is faster than the decoding time of

a large cache. As mentioned earlier, since two decoders can decode in parallel, possible word lines in a cache according to the least significant 4 bits may be ready to be read or written before the main row decoder even finishes decoding an address. The assumption here is that the main decoder has a larger number of address bits. Since the two decoding operations are independent, the delay of decoders is the maximum of two decoding times in the reconfigurable module. If there are many LUTs which take the same lower 4 bits in the module, we have to consider the increased capacitance due to the fan-out of the lower address bits. If the delay of decoding is higher, we may need a larger driver for the least significant 4 bits to reduce the delay. However, the drivers will not affect the size of the reconfigurable module much as we can put a driver into the space saved by reducing the size of the high-order address decoder.

Each bit line in a normal cache is replaced by the global line in the proposed architecture. Since the global line does not drive any gates (only the drain connections of the switches placed in an interleaved fashion - every sixteen cache blocks), the reconfigurable module does not have a higher delay due to the global lines. Although the global bit line in RFC is stretched by inserting the interconnection between LUT rows, the number of drains - dominant capacitance in the bit line - is reduced by a factor of 16. Thus, the segmented global bit line in the RFC has less capacitance than the bit line of a conventional cache. Additionally, the local bit line discharge can be done in parallel with the higher address bit decoding and word line propagation. This indicates that a data signal from a memory cell through the bit line in the module is propagated faster than a normal cache.

The word line in the reconfigurable cache is longer than in a memory cell array cache due to additional row decoders for each LUT. Therefore, the propagation delay of a signal from the higher-bit decoder through the word line in the module is slightly higher than in a normal cache. However, the sum of two propagation times, word and bit lines, is smaller than in a conventional cache since the local bit line in RC starts discharging before the word line finishes the propagation.

As mentioned earlier, other delays are similar in both the memory cell array cache and the RFC. In summary, the cache access time of RFC is faster in decoding time and bit/word line

propagation time. Therefore, the RFC is faster than a conventional memory cell array cache in read and write cache operations.

To estimate the access time, we used a cache simulator, CACTI with $0.8\mu m$ technology [50]. They partitioned the access time into five parts, Decoder, Word line, Bit line, Sense Amplifier, and Data out driver (for column decoder) for tag and data parts. We modified the simulator slightly to suit our structure, such as the parallel decoding and longer lines in the data array. We computed the access time of data part for 8KB cache with 128 bits block size, which is used to implement our example functions in Section 6. Table 4.1 shows the access time of a normal cache and the reconfigurable module. Due to the reason described above, it turns out that the overall decoding time has decreased and the delays of word and bit lines have increased. Note that CACTI simulator implemented in software cannot take account into the overlapped propagation time between the local and global word lines as described above. If the overlapped time is included, the sum of propagation times for word and bit lines would be smaller than the time shown in the table. However, since we reduce the decoding time significantly, the total cache access time, sum of the five factors, in the reconfigurable module is less than in a normal cache.

Table 4.1 Comparison of access time for an 8KB cache with 128bit-wide block

	Normal Cache (ns)	RFC Module (ns)	Comments
Decoder	2.88	Max(2.38, 2.28) = 2.38	decreased
Word line	1.14	1.24	increased
Bit line	0.46	0.55	increased
Sense Amplifier	0.58	0.58	same
Data out driver	0.60	0.60	same
Total time	5.66	5.35	decreased

Base array cache. Recall that the base array cache performs parallel decoding with segmented-bit and partitioned-word lines. Cache implementations may have a similar or more efficient parallel decoding structure with segmented bit lines and partitioned word lines (vertical partition in HP PA-RISC [55] and horizontal partition in Divided Word Line cache architec-

ture [56]). Unlike the RFC cache organization with vertical and horizontal partitions, some partitioned caches might employ only the vertical partition of cache blocks for less capacitance on the segmented bit lines because the stretched word line causes more delay than a synthesized sub-block. However, if we consider the word line propagation time with the discharging time of local bit lines, the horizontal partition with the dedicated decoders to each LUT (sub-memory module) can make the word line propagation faster. As described earlier, discharging the local bit line can start with charging the word line in RFC. If we partition a cache block only vertically for segmented bit lines, one bit line of each bit line pair in a cache block cannot be discharged unless the entire word line is fully charged (decoded) from the higher address-bit decoder. Although the entire stretched word line propagation in RFC is slightly slower due to insertion of the dedicated LUT decoders, the parallel discharge/charge of the local bit line/word line compensates the stretched word line (or makes it even faster). Therefore, we compare the access time for RFC to the base array cache partitioned vertically and horizontally with the segmented bit lines and partitioned word lines.

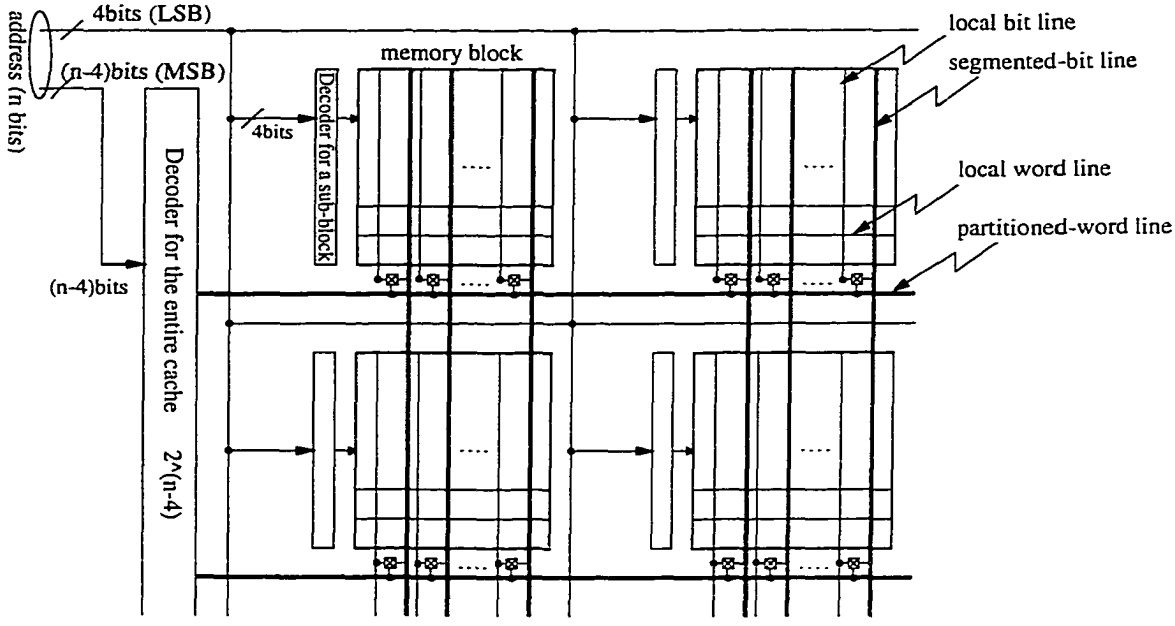


Figure 4.5 Parallel decode cache architecture (Base array cache) for faster cache access time

The access time of reconfigurable cache is slightly slower than that of a plain cache due to the stretched bit lines caused by the interconnections between LUT rows in the RFC. Based on the SPICE model parameters for $0.5\mu m$ technology in [57] the capacitance of the stretched bit line in the RFC is increased by 11% over the segmented bit line in the caches. Since the bit line access time constitutes 8% of the overall cache access time (estimated in [50]), the access time overhead due to the stretched line is about 1% of the overall cache access time. Since the word line propagation time, the decoding time, and other components in the RFC are similar to those in the base array cache, the overall cache access time in the RFC is slower than the base array cache by about 1%. The area overheads for FIR and DCT/IDCT function modules are given in Section 6.2 with respect to both cache models (memory cell array cache and base array cache). Using those faster decoding caches to implement reconfigurable modules, we can easily convert memory sub-blocks into LUTs without adding significant dedicated logic, such as decoders and address lines. Therefore, we build the reconfigurable caches into the caches based on the parallel decoding architecture.

4.5 Configuration and scheduling

In Section 4.5.1, we explain how to store and place the configuration data in a cache memory based on a conventional cache architecture. Then, in Section 4.5.2, we describe a way to load the configuration data initially and to load partial configuration data at run-time. The scheduling and controlling data flow for RFC is described in Section 4.5.3. The conditions for the efficient number and size of LUTs to build an RFC is described in Section 4.5.4.

4.5.1 Configuration of a computing unit

To reduce the complexity of column decoding in normal cache memory, data words are stored in an interleaved fashion within a block. The distance between two consecutive bits of a word is equal to the number of words in a block. However for LUT application, we need to use multiple bits for a single LUT. Due to the interleaved placement of data words in a cache block, we cannot write one entry of a multi-bit output LUT by writing one word in a cache.

This implies that we can only write one bit into a LUT if the width of LUT is the same as the number of words in a cache block *or* we can write two bits simultaneously into a LUT if the width of LUT is half the number of words in a cache block. For example, if a 4-LUT produces an n bit-wide output for a function and the number of words in a cache block is n, $16 \times n$ words - 16 for the number of entries and n for the width of LUT output (one bit from each word) - are required to be written to the LUT in the cache. However, since other LUTs placed in the same cache blocks (LUT row) can also be programmed simultaneously, no more than $16 \times n$ words are required to fill up the contents of all LUTs in the entire LUT row. In addition, if the width of a LUT is larger than the number of words in a cache block, multi-bit writing is performed into each LUT in a LUT row (as mentioned above). This restricts the width of the multi-bit output LUT to be an integral multiple of the number of words in a cache block. This allows for an efficient reconfiguration of all LUTs in a row. The number of LUTs in a column - placed vertically - for a pipeline stage may also be required to be a power of 2. Since all cache structures are based on a power of 2, it is more convenient to make all LUT parameters (length and width) a power of 2 to avoid a complicated controller and an arbitrary address generator. This may result in under-utilization of memory. However, the idle memory blocks for LUTs are not likely to be a problem when the module is used as a function unit due to availability of sufficient memory size in a cache.

4.5.2 Initial/partial reconfiguration

Initial configuration converts a cache into a specific function unit by writing all the entries of LUTs in the cache. The configuration data to program a cache into a function unit may be either available in an on-chip cache or an off-chip memory. Loading time for the configuration data in the latter case will be larger than in the former case. The configuration data may be prefetched by the host processor to reduce the loading time from off-chip memory. Using normal cache operations, multiple writes of configuration data to the LUTs are easily achieved.

An RFC operating as a function unit can also be partially reconfigured at run-time using write operations to the cache. When a partial reconfiguration occurs, the function unit must

wait for the reconfiguration to complete before feeding the inputs. Since computation data (input and output) and reconfiguration data (contents of LUTs) for a function unit share the global lines for data buses, we cannot perform both computing and partial reconfiguration at the same time. It is possible to perform both computations and reconfigurations simultaneously if we have separate data lines for computation data and configuration data. To process a large number of data elements, we do not need to reconfigure often. For example, in a Convolution application with 256 taps, we need to reconfigure a module implementing 8 taps 32 times.

The time to configure initially from the normal cache memory mode to a function unit mode or to reconfigure a part of a function unit depends on the number of cycles to write words into a cache. Initial configuration time dominates the total configuration/reconfiguration time. The partial reconfiguration at run-time usually loads a small part of configuration. The targeted SIMD applications require small initial and partial configurations and hence configuration has a small affect on overall execution time. The configuration time in our simulation (including initial and partial) for FIR and DCT with various function parameters are shown in Section 7.3. With a smaller number of data elements, the configuration time dominates the total execution time. However, the total execution time is not dominated by the configuration time when the number of input data elements exceeds a threshold (which is true for SIMD applications).

4.5.3 Scheduling and controlling data flow

A cache module can also be used to implement a function with a larger number of stages than what can be realized by the reconfigurable functional cache in one pass. In this case, we divide the function into multiple steps. That is, S stages required for a function can be split into sets, $S_1, S_2, \ldots, S_k$, such that each set S_i can be realized by a cache module. If all S_i's are similar, then we can adapt data caching as described in [58] to store the partial results of the previous stage as input for the processing by the next configuration. The 'similar' here means that the LUT contents may change, but the interconnection between stages is the same. This happens, for example, in Convolution applications. By changing the contents of LUTs, we can convert a stage in the cache block to carry out the operation of a different set of pipeline stages.

In general, Multiply-and-Accumulation (MAC) is a very common function in many DSP and image processing problems. The MAC for FIR and DCT/IDCT is implemented in this thesis in two ways using multiplier-adder and distributed arithmetic unit as shown in Section 6. The applications computing with MAC will have the same interconnection for all the computing stages with different LUT contents.

In data caching scheme, we place all input data in a cache and process it for the first set of stages, S_1. Following this, the cache module is configured for stages, S_2. We have to store the intermediate results from the current set of stages into another cache and then reload them for the next set of computations. To provide data without any stall, two other cache modules may be used to store input and intermediate data, respectively. These modules are address-mapped to provide efficient data caching for intermediate results. The role of the two caches can be swapped during the next step when a computation requires the intermediate results as inputs and generates another set of intermediate results. If both an input and an intermediate result are required by all the computations, the two caches cannot be swapped. The two caches must be large enough to hold input and intermediate results, respectively. Moreover, the reconfigurable functional cache must be able to accept an input and an intermediate result as its inputs.

4.5.4 Number and size of LUTs in RFC

The following conditions are used to determine the efficient number and size of LUTs with the parameters described in Table 4.2. The size and number of LUTs must be a power of 2 for a convenient control of partitioned memory blocks. By determining the most efficient number of LUTs in a row, the area overhead for the dedicated decoders to the LUTs could be reduced.

- Condition1:

  total # of bits/line $\geq$ (# of LUTs)$\times$(# of bits required per decoded entry in a LUT)

  $\Longrightarrow N_{b/w} \times N_{w/B} \geq x \times a$

- Condition2:

  # LUTs required for a function in a row $\leq \frac{total\ \#\ bits/line}{the\ width\ of\ LUT}$

  $\Longrightarrow x \leq \frac{n_{b/w} \times N_{w/B}}{m \times N_{w/B}}$ *or* $m \leq \frac{n_{b/w}}{x}$

These conditions imply that we can write $m \leq \frac{N_{b/w}}{x}$ configuration bits into each LUT in a row by writing one word into the cache. Also, the actual number of LUTs implemented in a row is equal to $\frac{n_{b/w}}{m}$.

Table 4.2 Parameters to determine the number and size of LUTs

a	Number of bits required in a row of LUT
x	Number of LUTs required in a LUT row
m	Number of bits to be written into a LUT by one word (power of 2)
$n_{b/w}$	Number of bits per word
$N_{w/B}$	Number of words per cache line

CHAPTER 5. ABC MICROPROCESSOR

5.1 Overview of microprocessor

In a reconfigurable cache module architecture (RCMA), we assume that the data cache is physically partitioned into n cache modules. Some of these cache modules are dedicated caches. The rest are reconfigurable modules. A processor is likely to have 256KB to 1MB Level-1 data cache within the next 5-10 years. Each cache module in our design is 8KB giving us 32 - 128 cache modules. A reconfigurable cache module can behave as a regular cache module or as a special purpose function unit.

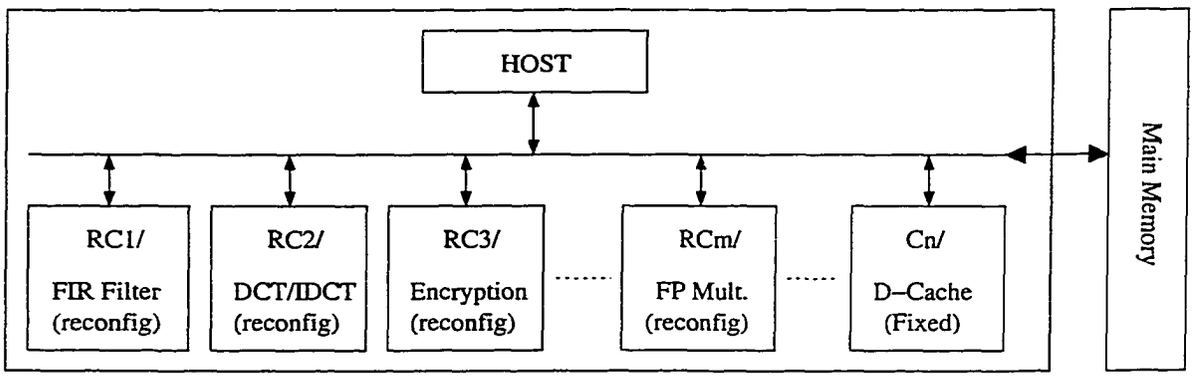


Figure 5.1 Overview of a processor with multiple reconfigurable cache modules

Figure 5.1 shows the overview of the processor with reconfigurable functional caches (RFCs). In an extreme case, these n cache modules can provide an n-way set associative cache. m modules out of n cache modules are reconfigurable. Whenever one of these cache modules is converted into a computing unit, the associativity of the cache drops or vice versa. Alternatively, the address space can be partitioned dynamically between the active cache modules

with the use of address bound registers to specify the cached address range. RFC*1*, RFC*2*, RFC*3*,..., RFC*m* in Figure 5.1 can be converted to function units, for example, to carry out functions such as FIR filter, DCT/IDCT (MPEG encoding/decoding function), encryption, and general computation unit like a floating point multiplier, respectively. When some subset of the m RFCs are used as function units, the other caches continue to operate as memory cache units as usual. It is also possible to configure some cache modules to become data input and output buffers for a function unit. The RFCs are configured by the processor in response to special instructions.

In this thesis, we propose that each cache module be designed to be reconfigurable into one of several specific function units. Since each reconfigurable module can be converted into a small set of functions with similar communication needs, interconnections for each RFC are fixed to be a super set of the communication needs of the supported functions. The advantages of fixed interconnection are as follows. The fixed interconnection is less complex, takes less area, and allows faster communication than a programmable interconnection. Moreover, our experience demonstrates the feasibility of merging several functions into one RFC with fixed interconnections. More discussion with respect to the fixed set of applications supported is presented in Section 7.4.

5.2 Microarchitecture with RFCs

We integrate the reconfigurable functional cache (RFC) described in Section 4 into a superscalar processor architecture to build the adaptive balanced computing (ABC) microprocessor with the use of RFC as a conventional data cache storage or a specialized computing unit on demand. First, we show a partitioned cache memory to provide a larger number of memory modules in Section 5.2.1. Second, the cache organization with RFCs is described in 5.2.2. Third, we describe a new instruction set to utilize the RFCs in Section 5.2.3. Finally, we show the proposed microarchitecture to exploit the RFCs and compiling requirements with the new instructions in Section 5.2.4 and 5.2.5, respectively.

5.2.1 Partitioned cache design

When one of the cache memory modules implemented as an RFC is reconfigured as a computing unit, the capacity of cache memory reduces by the fraction of the module to be configured. To reduce the effect of the reduction in memory storage capacity, we partitioned a cache memory into a number of smaller-sized memory modules. Then, each smaller module could be used to implement an RFC.

To partition a large cache memory, we apply a similar address decoding organization (design) in each cache module for an RFC as shown in Chapter 4. The decoder for each module is divided further to make a hierarchical decoding in higher address bits. This gives sixteen smaller-sized cache modules which can be built as an RFC. When one of the modules is configured as a computing unit, it should be excluded from the cache operations. This is done by disabling the dedicated decoder to the specific data array through ANDing the decoding line from the 2-to-4 decoder and an RFC flag for computing mode. More details are given in the following sections.

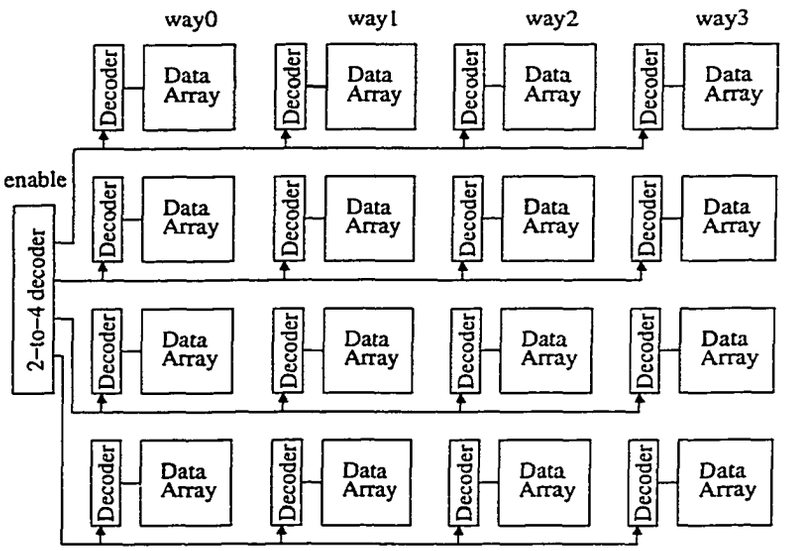


Figure 5.2 Partitioned cache for multiple modules

5.2.2 Cache organization with reconfigurable functional caches

We employ a multiple-way set associative data cache memory in a RISC superscalar microprocessor to support the RFCs. Modules in a set associative cache can be built as RFCs. Each RFC can be configured to a specialized computing function or used as a normal cache memory module. If the size of a cache module for one-way in a multiple-way set associativity is large enough to be mapped to more than one RFC, we partition them into multiple smaller-sized cache modules as described in Section 5.2.1.

Two possible cache organizations with the address mapping including RFCs are shown in Figure 5.3. To prevent the memory address space from becoming non-cacheable (they cannot be stored in L-1 data cache due to removal of the blocks from the cache operations), not all of the cache modules are configured as computing units at the same time in Figure 5.3(a). RFCs can be implemented with a minimal cache modification in this organization. In addition, one module can be easily excluded from the cache operation because the cache partition is already provided in a multiple-way set associative cache memory. The cache memory capacity is reduced when an RFC converts it into a computing unit. This results in a full dynamic associativity of cache memory when configuring. For example, if one out of four cache modules is configured to a computing unit, only 3-way blocks in all sets are left to map the address space in lower-level memory. This may cause more cache misses, which results in degradation of the performance. However, this can be compensated adequately by accelerating the computations.

A further partition of cache memory within a module (way) is shown in Figure 5.3(b). The structure of further cache partition is described in Section 5.2.1. The size of each module is based on the minimal size of RFC - 8KB as shown in Chapter 4. In this organization, when an RFC converts into a computing unit, the sets containing the RFC are less-cacheable (low associativity compared to other sets) while the other sets retain the same caching capacity. This scheme retains more storage than the full dynamic associativity organization of Figure 5.3(a) (each way corresponding to RFC) by converting a smaller portion of cache memory. This could reduce the impact of the RFC reconfiguration when the RFC works as a computing unit with the partial dynamic associativity.

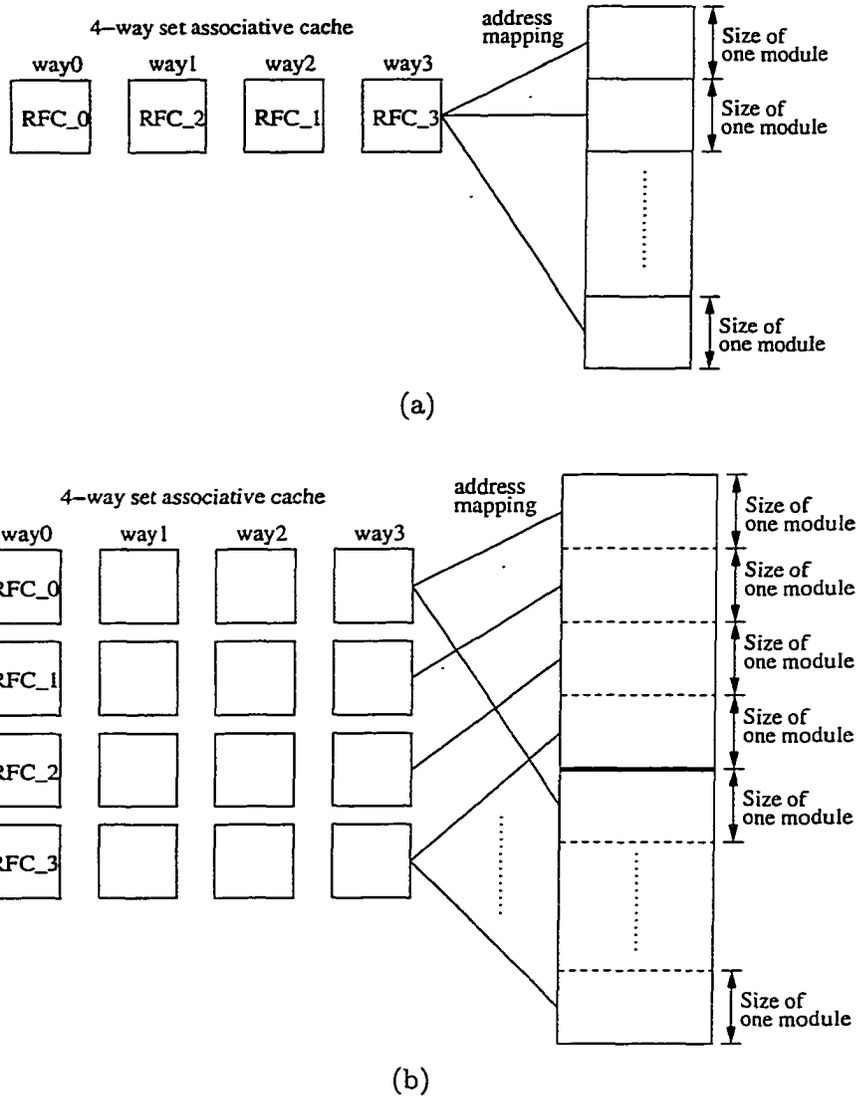
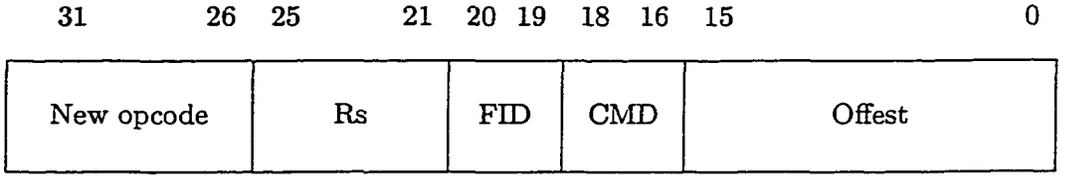


Figure 5.3 Cache organizations and address mapping with RFCs (a) 4 (b) 16 cache modules

5.2.3 Instructions to utilize RFC

It takes three steps to perform a computation in RFC. First, the RFC is configured as a specialized computing unit by loading the pre-defined configuration data into the specific cache memory (RFC). Second, the input data to be processed is loaded to the RFC. Third, the output data from the RFC is stored back into the memory. All these steps require only load/store class of instructions. To configure an RFC and then execute computations in the RFC in a conventional microarchitecture (RISC processor), new instructions - named rfc class of instructions - are added with a new opcode. The format of rfc instructions is the same as the conventional load/store instructions except for the target register field. The *rfc load/store* instructions need not have the target register field in the microcode as is the case for conventional load/store instructions. This format provides an address for access to memory hierarchy.

Three types of instructions, *rfc_load_conf*, *rfc_load_in*, *rfc_store_out*, and initialize / terminate instructions are added. The detailed format of the new class is described in Figure 5.4, where we show only word-data type load/store instructions (lw/sw). The instruction description is mostly self-explanatory. Individual instruction operations are explained briefly later in Section 5.2.4. Note that the function identifier (FID) field chooses the module that will be configured for a specific function. In this thesis, we use word-data type rfc instructions for the description purpose. For different types of data (for example, half-word or byte, etc.), the same instruction format in the figure can be used. Also, note that *rfc_lw_conf_end* and *rfc_partial_set* also perform a loading operation in addition to setting the mode. The *rfc_lw_conf_start* instruction is used for a special setting in a RFC, such as use of multiple input buffers (i.e. input and intermediate data). We assume that there are only four functions, but the concept can be easily extended. The special flags can be used for a two-level function identification of RFCs if more than four RFCs are implemented. For example, the flag with *rfc_lw_conf_start* sets a function class followed by sub-FID.



FID (function identifier) : 4 different functions to be implemented into RFCs

00 - Function 0

01 - Function 1

10 - Function 2

11 - Function 3

CMD (command) : type of operations

000 - start configuration and set the special state register and other required flags

001 - load configuration data (from reserved address space)

010 - end of loading configuration data and set to execution mode

011 - terminate an RFC use for computation and return back to the cache mode

100 - set flags and load input data to be processed for the computation

101 - load 2nd set of input data from memory hierarchy if applicable

110 - store output data to memory hierarchy

111 - set for partial reconfiguration process at the end of current step

RFC instructions for load-word based on the CMD

```
000 :  rfc_lw_conf_start   F_class   special_flags
001 :  rfc_lw_conf         fid       offset($r)
010 :  rfc_lw_conf_end     fid       offset($r)
011 :  rfc_terminate       fid
100 :  rfc_lw_in1          fid       offset($r)
101 :  rfc_lw_in2          fid       offset($r)
110 :  rfc_sw_out          fid       offset($r)
111 :  rfc_partial_set     fid       offset($r)
```

Figure 5.4 rfc instructions for loading and storing "word" type of data

5.2.4 Mechanism for the computation in RFC

5.2.4.1 Overall microarchitecture

With out-of-order issue in a superscalar processor, any instruction which does not have a dependency on preceding instructions can be issued and executed at any time if the required resources are available. In addition, in a speculative execution, the next instruction stream in a code sequence can be executed speculatively. The out-of-order issue and execution may also happen among rfc instructions because there is no explicit dependency between rfc instructions. However, the rfc_lw_in(1 *or* 2) and rfc_sw_out instructions must not be issued and executed until the RFC has been configured. From the microarchitecture viewpoint, a speculative execution mechanism may issue the rfc instructions in any order. To avoid this type of exception, we add a special RFC state register. In the register, two bits are reserved for each RFC module. The two-bit RFC state information is organized as follows.

- 00 : NON-RFC/END-RFC - normal mode; the RFC is not performing a computation, but functioning as a normal cache. $rfc\_lw\_conf\_start$ checks this to make sure that no execution in a designated RFC is performed and $rfc\_terminate$ sets this to notify the end of an execution

- 01 : CONF - configuration mode; $rfc\_lw\_conf\_start$ sets this to notify the configuration mode being performed and the following rfc_lw_conf instructions load configuration data with checking the configuration status

- 10 : CONF_DONE/START-RFC_EXE - end of configuration; $rfc\_lw\_conf\_end$ sets this to notify the configuration DONE and rfc_lw_in checks this to make sure that no more configuration is being performed

- 11 : RFC_EXE - execution mode; the first rfc_lw_in sets this to notify the execution being performed and the following rfc_lw_in and rfc_sw_in instructions process the input and output data

The state transition is controlled by the rfc instructions as depicted in Figure 5.5. All the rfc instructions must access the RFC state register according to the FID field in the microcode and then check the current state with its CMD field. If it is an allowed state, the rfc instructions can be issued. Otherwise, the rfc instructions are stalled until the corresponding state is resolved.

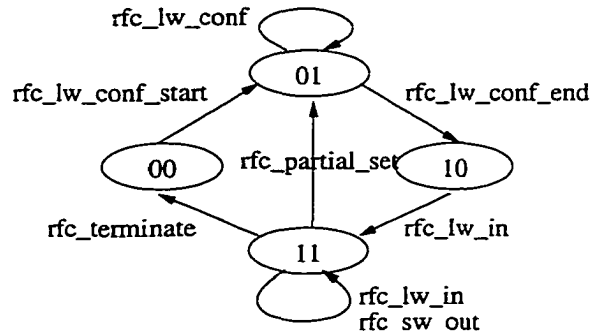


Figure 5.5 State transition for the RFC status

5.2.4.2 Configuration

The configuration of RFC from a cache module simply implies loading all the contents of LUTs required to construct a computing unit. A normal cache read with a small modification directs specific data into the designated cache line. The required configuration data for RFCs resides in a reserved memory (address) space in main memory. The configuration is loaded into main memory when the system boots up. The $rfc\_lw\_conf\_start$ instruction sets the corresponding RFC state register. The subsequent rfc_lw_conf instructions load the configuration lines to the specific RFC without changing the RFC state.

Since the configuration data is held in main memory, the data accesses would be cache misses if the same configuration had not been loaded previously. This cache miss will replace the current clean/dirty lines in a write-back cache. Thus, we do not require any separate cache flushing. A simple modification of cache replacement mechanism, such as LRU, is required to replace the data in the specific cache module (RFC) with the loaded configuration data in a set-associative cache organization. The modified LRU scheme, which is set by $rfc\_lw\_conf\_start$,

replaces lines in the RFC only if rfc_lw_conf accesses in $CONF$ mode. The configuration data in RFC should not be modified by other load/store instructions except the rfc_lw_conf instructions if the RFC state is not '00'. An additional operation in the modified LRU mechanism protects the configuration data by removing the lines in RFC from the replacement line list in LRU when conventional load/store instructions access the cache. Thus, the proposed LRU scheme consists of two operations, one for rfc_lw_conf to replace data in the specific RFC and the other for other load/store instructions (including $rfc\_lw\_in1/2$) to access the rest of cache memory.

A conventional load/store instruction may see a hit in the RFC block during the reconfiguration as old data, which have not been replaced yet, may still reside. This is acceptable since all old data will eventually be replaced by configuration data. If a computation is being performed in RFC (RFC_EXE mode), the read/write access to RFC must be blocked (using the state bits). Using the proposed LRU modification, the set associativity of cache memory is changed dynamically depending upon the use of RFC. During the configuration, one module out of multiple modules in a set associative cache is frozen out of cache operations. A write operation to RFC is prevented by disabling the write enable in the RFC during the execution mode.

5.2.4.3 Execution stage

The new instructions are decoded in the dispatch stage according to their specification described in Section 5.2.3. Since the format of new instructions is similar to that of conventional load/store instructions, the complexity of the additional decoding logic is not significant. The $rfc\_lw\_in1/2$ instruction, which loads input data to an RFC computing unit, is decoupled from LSQ (Load Store Queue) and queued into an input buffer (IBUF) dedicated to each RFC. This allows the $rfc\_lw\_in1/2$ instructions to be independent of the LSQ and dispatch of more rfc_sw_out instructions. In addition, the decoupled IBUF provides data in-order to the RFC computing unit. The rfc load/store instructions also process the input/output data independently in separate buffers. Otherwise, these instructions pass/receive data to/from

RFC serially in one buffer. The effective address for the $rfc\_lw\_in1/2$ instruction is calculated using the existing datapath and passed to the corresponding instruction in IBUF, not to LSQ. The details of IBUF are described later. If no slot in IBUF is available, the following instructions including conventional instructions fetched from memory are stalled until IBUF is again available. Otherwise, a complex mechanism may be required for a decoupled fetch queue between rfc load instructions and other instructions.

By queueing the $rfc\_lw\_in1/2$ instructions into IBUF in-order, the input data to be processed is provided to RFC in the correct order. This is like a reorder buffer mechanism for input data of the RFC unit to remove the impact of out-of-order execution. The input data from memory to the IBUF can be supplied out-of-order as in the conventional LSQ. Note that instead of providing IBUF with the data in writeback stage, the data may be directly loaded into the matched address slot in IBUF from data buses. Some applications may require two separate inputs. For example, in an FIR filter with many taps, an input data stream may be processed by a fixed number of taps and partial results are stored. In the second iteration, input and partial results are loaded (using rfc_lw_in1 and rfc_lw_in2, respectively) and processed together. That is why we provide two IBUFs. The computation in RFC is processed when both data elements are available.

The mechanism to support out-of-order update (load) of input data in IBUF dedicated to RFC is as follows. When the rfc instructions are issued and executed, the FID and CMD fields (5 bits) are propagated with the address to be accessed together. Each IBUF attached to RFCs keeps snooping these five signals and compares them with its own function identifier (FID) and the corresponding command (loading rfc input data only). If the FID and CMD match with one of the IBUFs and the cache access is a read, the loaded data is directly queued into the matched address slot in IBUF with the current address accessed. This mechanism is very similar to the conventional LSQ. Whenever the head slot in IBUF is updated (ready), it is provided to the RFC for the computation. This ensures that data to RFC is delivered in a correct order. The proposed mechanism is shown in Figure 5.6 (a) and (b). As shown, the mechanism requires the FID and CMD lines to be added to the bus.

After a computation is completed in the RFC, the output data is queued into an output buffer (OBUF) as shown in Figure 5.6(b). The OBUF is a simple FIFO register file since the queued data is already in-order. The presence of OBUF may reduce the stall time for the *rfc_sw_out* instruction ready in commit stage due to the latency of computation in RFC. In addition, it also reduces the stall time for the computation in RFC. For example, if OBUF is not present, the ready output to be stored blocks the following processed input data and the whole computing unit until the output is resolved by the *rfc_sw_out* instruction in commit stage, and vice-versa.

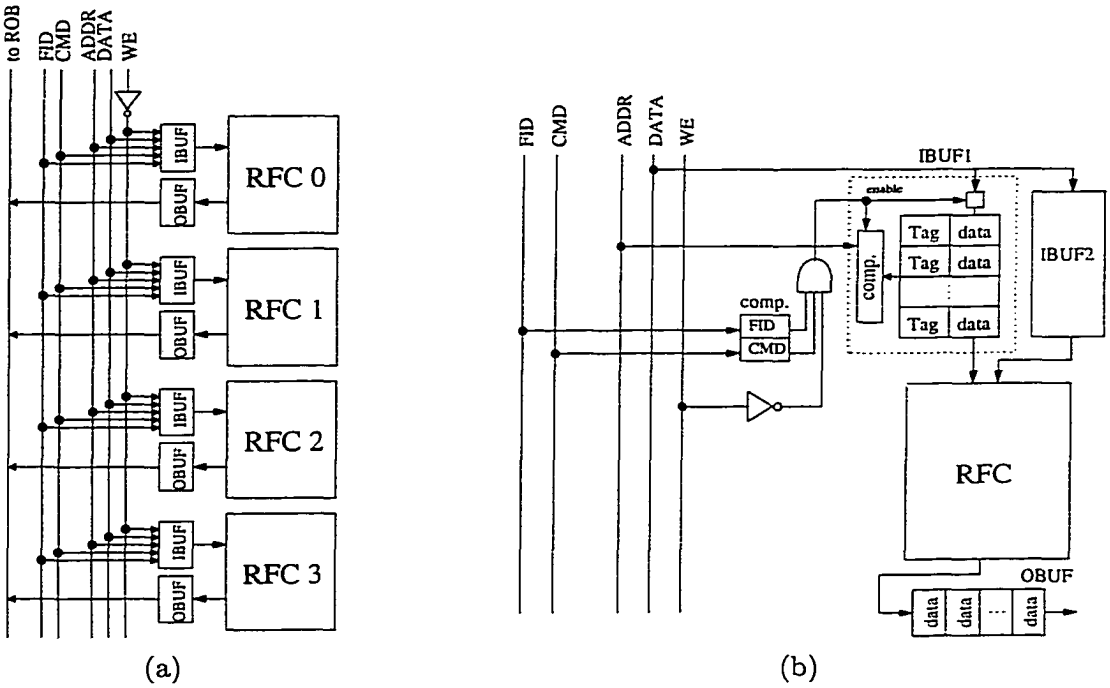


Figure 5.6 (a) Overview of I/O buffers organization; (b) I/O buffers to dedicated to RFC

The *rfc_terminate* instruction sets the RFC state into the non-RFC mode after finishing an entire computation. This setting should be done in the commit stage to avoid mis-execution of pending *rfc* instructions. If the same computation within the current configuration may be performed in the near future, the RFC state (*RFC_EXE*) is not changed.

The *rfc_lw_in*1/2 instructions do not affect any state when a mis-prediction/speculation

or an exception occurs because the $rfc\_lw\_in1/2$ instructions do not modify the precise state in microarchitecture. If an exception or mis-speculation occurs. the only repair necessary is to flush the instructions in IBUF as done in a conventional LSQ. In addition, all the computed data in RFC is resolved by the rfc_sw_out instructions. This implies that wrong data elements, which should not be processed (such as mis-speculation), will not be stored because the rfc_sw_out instructions are committed in-order.

5.2.4.4 Partial reconfiguration process

A whole function in an application may not be mapped to an RFC as a computing unit at one time. For instance, in an FIR filter, if the number of taps for the filtering coefficients is larger than the number of physical taps implemented in an RFC, we configure the first set of taps in the RFC and then reconfigure it partially for the next set of coefficients at run-time. In partial reconfiguration, not all of LUTs in the RFC need to be reconfigured since only the coefficients are changed. This can be achieved using a cache write operation in rfc_lw_conf. To direct the partial configuration data to a specific RFC, we use the rfc_lw_conf instruction described in Section 5.2.3 as we do for initial loading configuration. However, the RFC state must be changed to $CONF$ mode and the modified LRU mechanism is set from RFC_EXE to $CONF$ mode to replace the specific block in a set associativity as well. The $rfc\_partial\_set$ instruction shown in Section 5.2.3 sets the RFC state register and all the required flags as $rfc\_lw\_conf\_start$ does. This mechanism protects the current configuration to be retained data in the RFC for the partial reconfiguration by setting the RFC state from RFC_EXE to $CONF$ mode directly. Note that we assume a correct program will not try to write in the same address space where the configuration is stored. The following rfc_lw_conf instructions reconfigure the designated RFC as done for the initial configuration. Again, the partial configuration data resides on main memory with a reserved address space which maps to the same area in the cache.

5.2.4.5 Forwarding mechanism between LSQ and I/OBUF

Store instructions may need to forward the data to subsequent load instructions for correct operations. Otherwise, the old data is accessed. The forwarding mechanism between load instructions is not necessary because loads do not modify the data. Forwarding data between store instructions is not required because the subsequent stores have the latest data. However, for the faster execution by removing the read access to memory, a forwarding mechanism may be used.

The rfc store instructions need to forward data to both conventional and rfc load instructions if the addresses match. Since the rfc stores are queued and processed in LSQ, the same forwarding mechanism between conventional load and store instructions is used for forwarding data from the rfc stores to load instructions. The addresses are compared before the load instructions are issued to access the memory. The only difference is that the data to be forwarded by rfc stores resides in OBUF. Thus, additional data buses are required between LSQ and OBUF.

Similarly, the rfc load instructions queued into IBUF may receive data from the conventional store instructions in LSQ. Since only preceding stores forward data to the subsequent loads and IBUF and LSQ are separated, the identification of the sequence order between store and rfc load instructions is required. To identify the order, an instruction sequence number is tagged to the instructions in both LSQ and IBUF. Using the tag number, the store instruction compare its address with only the addresses of the subsequent rfc load instructions. It also requires additional buses between LSQ and IBUF. The tag numbers can be used to identify the instructions to be flushed in IBUF as well when a mis-prediction occurs.

5.2.4.6 Forwarding mechanism between IBUF and OBUF

In some RFC computations, i.e. IIR, a calculated value in the RFC currently is reused for the computation. Thus, a mechanism is required to forward data from OBUF to IBUF. The addresses in LSQ for store instructions must be compared with the address of a load instruction to see if there is a match and the data forwarding is required. If so, the rfc load instruction

should be tagged in IBUF. When the rfc stores instructions store the data from OBUF to the memory (or cache), the IBUF may also read the corresponding data from the bus directly by checking the tags, which is similar to the mechanism for loading data from memory to IBUF as described in Section 5.2.4.3. This ensures that the rfc load instructions do not receive old data.

5.2.5 Compiling requirements for the specialized computations

The availability of RFCs allows programmers to use specialized computing units with simple pre-defined and code-optimized function calls (e.g. FIR(), DCT(), etc.). The pre-defined mapping configuration data and parameter information are included into the compiler libraries. It simplifies programming and more function calls can be added to the compiler as and where they are developed. Figure 5.7 shows a code sequence for a generic pre-defined function call to use RFCs. Note that it uses unrolling of $rfc\_lw\_in1/2$ and rfc_sw_out instructions 4 to 8 times to allow faster pipelining of data in the IBUF and storage of results. The number of unrolled rfc instructions is determined by the size of IBUF.

```
START:
        rfc_lw_conf_start   0              0                  # set the RFC status
                                                              # as CONF
                                                              # F_class 0
                                                              # no additional IBUF
        addiu               $r5, $0        0                  # loop index
CONFIG:
        slt                 $r4, $r5       (no of iterations) # number of sets
        bne                 $r4, $0        load_conf
        j                   conf_end
load_conf:
        rfc_lw_conf         FID            offset($r1)        # load configuration data
                                                              # from memory to RFC
        addiu               $r1, $r1       (cache line size)  # address increment
                                                              # (cache line size)
        addiu               $r5, $r5       1                  # loop index increment
        j                   CONFIG
conf_end:
        rfc_lw_conf_end     FID            offset($r1)        # set the RFC status
                                                              # as CONF done

        addiu               $r5, $0        0                  # loop index
EXECUTE:
        slt                 $r4, $r5       (no of iterations) # number of sets
        bne                 $r4, $0        load_store
        j                   END
load_store:
        rfc_lw_in           FID            offset($r2)        # load the input data
        rfc_lw_in           FID            offset($r2)        # from memory
        rfc_lw_in           FID            offset($r2)
        rfc_lw_in           FID            offset($r2)

        rfc_sw_out          FID            offset($r3)        # store the output data
        rfc_sw_out          FID            offset($r3)        # to memory
        rfc_sw_out          FID            offset($r3)
        rfc_sw_out          FID            offset($r3)

        addiu               $r2, $r2       32                 # address increment
        addiu               $r3, $r3       32                 # address increment
                                                              # (one word)
        addiu               $r5, $r5       1                  # loop index increment
        j                   EXECUTE
END:
        rfc_terminate                                         # set the RFC status
                                                              # as EXE done
```

Figure 5.7 A basic frame code using RFC as specialized computing units

CHAPTER 6. EXAMPLES OF RFC

We have experimented with two applications, Convolution and DCT/IDCT. In this chapter, we describe how we map the applications into the reconfigurable functional cache (RFC). First, we map each application into RFC separately, then we merge two applications into a single RFC. We also compare the overall area of separated RFCs and a combined RFC in Section 6.2. Next, we compare the execution time of these applications on RFCs with the execution time on General-Purpose Processor (GPP) in Section 6.4. The main advantage of the RFCs is on-chip processing, which implies faster processing time, no off-chip bottlenecks, and the balance/utilization of on-chip caches between storage and computation.

6.1 Functions to be mapped to RFC

6.1.1 Convolution (FIR filter)

An RFC configured to perform a Convolution function is presented in this section. The number of pipeline stages for the Convolution in an RFC depends upon the size of the cache to be converted. Our simulation is based on an 8KB size cache with 128 bits per block/16-bit wide words implementing 4-LUTs with 16-bit output. A conventional Convolution algorithm (FIR) is shown in Equation 6.1.

$$y(n) = \sum_{k=0}^{M} w(k)x(n-k) \tag{6.1}$$

One stage of Convolution consists of a multiplier and an adder. In our example, each stage is implemented by an 8-bit constant coefficient multiplier and a 24-bit adder to accumulate up to 256 taps in Figure 6.1(a). The input data is double pipelined in one stage for the appropriate computation [30]. An 8×8 constant coefficient multiplier can be implemented using two 4×8

constant coefficient multipliers and a 12-bit adder with appropriate connections [54]. A 4×8 constant coefficient multiplier is implemented using 12 4-LUTs with single output from each LUT on FPGAs. In our implementation, we split the 12-bit wide LUT contents of a 4×8 conventional constant coefficient multiplier into two 16-bit output 4-LUTs (part 1, 2) with 6-bit wide multiple outputs for a lower routing complexity of the interconnections as shown in Figure 6.1 (b). The first six bits of each content are stored in LUT part 1 while the last six bits are stored in LUT part 2 to realize a 4×8 constant multiplier.

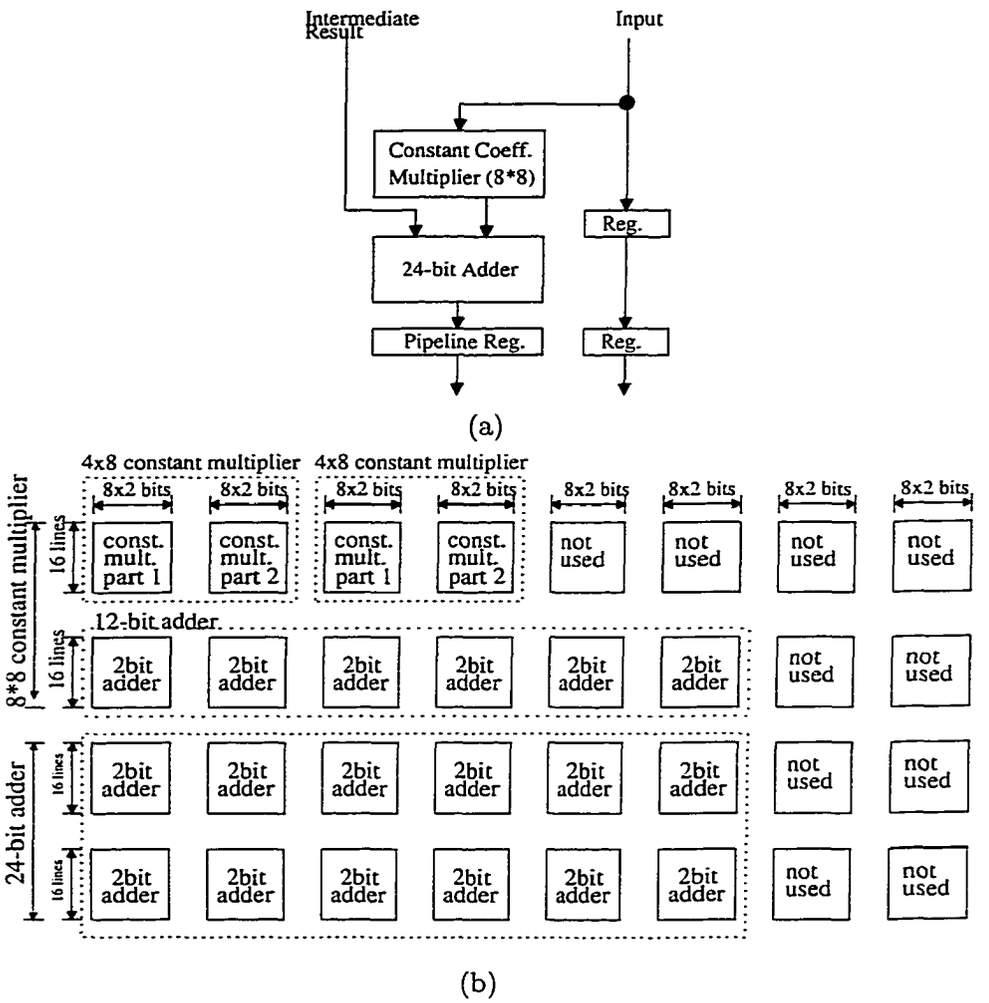


Figure 6.1 (a) One stage of Convolution; (b) Array of LUTs for one stage of Convolution

The concept of a carry select adder is employed for an addition using the LUTs described in Section 4.1. Therefore, we need a 6-bit wide result for a 2-bit addition, three bits when carry-in=0 and three bits when carry-in=1 from a LUT. An n-bit adder can be implemented using $\lceil \frac{n}{2} \rceil$ such LUTs and the carry propagation scheme. The output is selected based on the input carry.

One stage of Convolution can be implemented with 22 LUTs. To keep the number of LUT rows a power of two for cache operation, we put 6 LUTs in each LUT row and have 4 LUT rows to use 22 (out of 32) required LUTs. The final placement of LUTs is shown in Figure 6.1 (b). A few LUTs in the figure are not used for the computation. In Figure 6.1(b), pipeline registers and interconnections for LUTs are not shown. For an 8KB RFC, we have 32 rows of LUT which can be used to implement 8 taps of the Convolution algorithm.

6.1.2 DCT/IDCT (MPEG encoding/decoding)

In this section, we show a reconfigurable cache module performing the DCT/IDCT function, which is the most effective transform technique for image and video processing [60, 61, 65, 68]. To be able to merge the Convolution and DCT/IDCT functions into the same cache, we have implemented DCT/IDCT within the number of LUTs in the Convolution cache module.

Given an input block $x(i,j)$, the $N \times N$ 2-dimensional DCT/IDCT in [68] is defined as

$$X(u,v) = \frac{2}{N} C(u)C(v) \sum_{i=0}^{N-1} \sum_{j=0}^{N-1} x(i,j) \times cos\frac{(2i+1)u\pi}{2N} cos\frac{(2j+1)v\pi}{2N} \tag{6.2}$$

$$x(i,j) = \frac{2}{N} \sum_{u=0}^{N-1} \sum_{v=0}^{N-1} C(u)C(v)X(u,v) \times cos\frac{(2i+1)u\pi}{2N} cos\frac{(2j+1)v\pi}{2N} \tag{6.3}$$

where $x(i,j)$ $(i,\ j = 0, \cdots, N-1)$ is a matrix of the pixel data, $X(u,v)$ $(u,\ v = 0, \cdots, N-1)$ is a matrix of the transformed coefficients, and $C(0) = \frac{1}{\sqrt{2}}$, $C(u) = C(v) = 1$ if $u, v \neq 0$.

This $N \times N$ 2-D cosine transform can be partitioned into two N point 1-D transforms. To complete a 2-D DCT, two 1-D DCT/IDCT processes are performed sequentially with an intermediate storage. By exploiting a fast algorithm (the symmetry property) presented in [59, 68], an $N \times N$ matrix multiplication for the $N \times N$ 2-D cosine transform defined in (6.2)

and (6.3) can be partitioned into two $\frac{N}{2} \times \frac{N}{2}$ matrix multiplications of 1-D DCT/IDCT with additions/subtractions before the DCT process and after the IDCT process.

The 8-point DCT with the symmetry property can be written in a matrix form as shown below.

$$\begin{bmatrix} X_0 \\ X_2 \\ X_4 \\ X_6 \end{bmatrix} = \frac{1}{2} \begin{bmatrix} A & A & A & A \\ B & C & -C & -B \\ A & -A & -A & A \\ C & B & B & -C \end{bmatrix} \begin{bmatrix} x_0 + x_7 \\ x_1 + x_6 \\ x_2 + x_5 \\ x_3 + x_4 \end{bmatrix} \tag{6.4}$$

$$\begin{bmatrix} X_1 \\ X_3 \\ X_5 \\ X_7 \end{bmatrix} = \frac{1}{2} \begin{bmatrix} D & E & F & G \\ E & G & -D & -F \\ F & -D & -G & E \\ G & F & E & -D \end{bmatrix} \begin{bmatrix} x_0 - x_7 \\ x_1 - x_6 \\ x_2 - x_5 \\ x_3 - x_4 \end{bmatrix} \tag{6.5}$$

$A = cos\frac{\pi}{4}$, $B = cos\frac{\pi}{8}$, $C = sin\frac{\pi}{8}$, $D = cos\frac{\pi}{16}$, $E = cos\frac{3\pi}{16}$, $F = sin\frac{3\pi}{16}$, $G = cos\frac{\pi}{16}$, where x_i is the pixel data in a row or column and X_u is the transformed coefficient. The 8-point IDCT is written as follows.

$$\begin{bmatrix} x_0 \\ x_1 \\ x_2 \\ x_3 \end{bmatrix} = \frac{1}{2} \begin{bmatrix} A & B & A & C \\ A & C & -A & -B \\ A & -C & -A & B \\ A & -B & A & -C \end{bmatrix} \begin{bmatrix} X_0 \\ X_2 \\ X_4 \\ X_6 \end{bmatrix} + \frac{1}{2} \begin{bmatrix} D & E & F & G \\ E & -G & -D & -F \\ F & -D & -G & E \\ G & -F & E & -D \end{bmatrix} \begin{bmatrix} X_1 \\ X_3 \\ X_5 \\ X_7 \end{bmatrix} \tag{6.6}$$

$$\begin{bmatrix} x_7 \\ x_6 \\ x_5 \\ x_4 \end{bmatrix} = \frac{1}{2} \begin{bmatrix} A & B & A & C \\ A & C & -A & -B \\ A & -C & -A & B \\ A & -B & A & -C \end{bmatrix} \begin{bmatrix} X_0 \\ X_2 \\ X_4 \\ X_6 \end{bmatrix} - \frac{1}{2} \begin{bmatrix} D & E & F & G \\ E & -G & -D & -F \\ F & -D & -G & E \\ G & -F & E & -D \end{bmatrix} \begin{bmatrix} X_1 \\ X_3 \\ X_5 \\ X_7 \end{bmatrix} \tag{6.7}$$

According to the fast algorithm, the number of multiplications in (6.2) and (6.3) can be halved. However, $\frac{N}{2}$ adders and $\frac{N}{2}$ subtracters are needed before the DCT process and after the IDCT process.

The 1-D DCT/IDCT process is a Multiply-and-Accumulate (MAC), which can be represented as $y = \sum_{i=0}^{N-1} a_i x_i$. Although a MAC is already built in the RFC in 6.1.1, the distributed arithmetic [66, 68] is employed in the RFC instead for the DCT/IDCT function. This avoids the run-time reconfiguration of coefficients required for the coefficient multiplier in FIR. Using this scheme, once the coefficients are configured into the RFC, no more run-time reconfiguration is required.

The inner product of each 1-D transform (MAC) can be represented as follows.

$$\begin{aligned} y &= \sum_{i=0}^{N-1} a_i x_i = \sum_{i=0}^{N-1} a_i(-b_{i0} + \sum_{r=1}^{W_d-1} b_{ir}2^{-r}) \\ &= \sum_{r=1}^{W_d-1} \left[\sum_{i=0}^{N-1} a_i b_{ir}\right] 2^{-r} + \sum_{i=0}^{N-1} a_k(-b_{i0}) \end{aligned} \tag{6.8}$$

where $x_i = -b_{i0} + \sum_{r=1}^{W_d-1} b_{ir}2^{-r}$ with two's complement form of an input word length W_d and a_i ($i = 0, 1, 2,..., N\text{-}1$) are the weighted cosine factors. According to (6.8), the multiplication with the coefficients can be performed with a ROM containing 2^N pre-calculated partial products ($\sum_{i=0}^{N-1} a_i b_{in}$) in a bit-serial fashion. The inner product computes the sums of partial products corresponding to the same order bit from all the input elements processed in the current stage using a set of serial shift registers. For the output of the inner product, one more shift register is required. Therefore, one processing element (PE) contains a ROM and a shift accumulator for the partial summations of corresponding data bit order. In this configuration, each inner product is completed in the number of clock cycles that is the same as the word length of input. With N PEs, N-point DCT can be completed in parallel. Using the symmetry property presented in (6.4), (6.5), (6.6), and (6.7), the contents of a ROM can be reduced by $2^{\frac{N}{2}}$. However, it requires two sets of $\frac{N}{2}$ adders and $\frac{N}{2}$ subtracters before the DCT process and after the IDCT process.

Due to the coding efficiency and the implementation complexity, a block size of 8×8 pixels is commonly used in image processing [60]. We, therefore, have implemented an 8×8 2-D DCT/IDCT function unit by two sequential 1-D transform processes. In addition, the width of input elements is eight bits. We also select the word length of the coefficients to be 16 bits for the accuracy of the DCT computation.

One PE with conventional architecture is depicted in Figure 6.2(a). One PE implemented in the RFC is depicted in Figure 6.2(b). In the figure, the ROM is placed in the middle of a LUT row to reduce the number of routing tracks. In the given cache size, 8KB, eight such PEs and the additional adders/subtracters for pre/post-processing can be implemented. To make the DCT/IDCT implementation compatible with the Convolution function unit, we place 4-LUTs with 16-bit output in an 8KB sized cache. Only 20 LUT rows (16 for PEs and 4 for pre/post-processing) out of 32 LUT row in the 8KB cache are used for the implementation. However, the LUTs not used in this function still remain in the RFC module for the compatibility with the other functions. Since each PE requires a LUT as a 16×16 ROM and a 16-bit adder, no significant consideration of the LUT placement is necessary in this design unlike the implementation of Convolution.

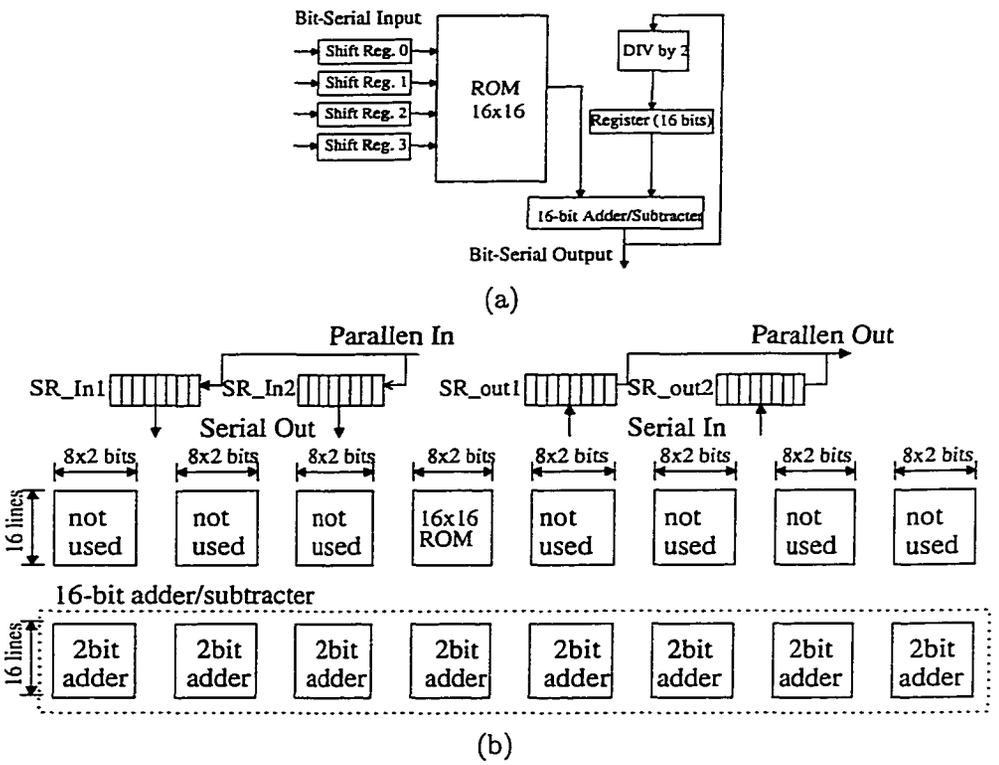


Figure 6.2 (a) DCT/IDCT processing element; (b) Array of LUTs for DCT/IDCT processing element with the input registers

A 16-bit carry select adder is configured as a shift accumulator with the registers not shown in the figure for the self-accumulation in each PE. According to Equation (6.8), only one subtraction is necessary. This is done by the same adder which can keep both addition and subtraction configurations in 12-bit data width (6 bits for adder and 6 bits for subtracter). The adder-subtracter shares the same input and output with the adder without requiring any extra logic. However, an extra control signal is needed to enable the subtraction. The additional adders and subtracters for the pre/post-processing are implemented using the scheme for adder-subtracter described above since each pair of addition/subtraction needs the same input elements. In addition, the 1-bit shift of accumulated data can be easily done by appropriate connections from the registers to the input data lines of the adder. The input/output shift registers are added only to the in/out port of the actual DCT/IDCT function unit after the pre-processing unit and before the post-processing unit. This means only one set of shift registers are necessary since all the PEs compute using 4 bits out of the same set of input data in each transform of a row or a column as shown in (6.4)-(6.7).

In the actual implementation, we add one more set of shift registers to remove any delay due to loading or storing in/out data from other memories. All the loading/writing back from/to the storage can be overlapped with the computation cycle time in PEs by appropriate multiplexing of the dual shift registers. Adding shift registers allows in/out data to be ready to be processed and written immediately after the previous computation, without any idle time. The controller described in Section 4.5.3 handles and controls the computation procedure. Once the host processor passes the required information to the controller, all the control signals are sent by the controller.

The computation process of an 8×8 2-D DCT is as follows. The function unit on the RFC computes the 1-D transform for an entire row by broadcasting a set of input data after the pre-addition/subtraction process to eight PEs in eight time units in a bit serial fashion (i.e. a half set of data to four PEs and another half set of data to other four PEs). A set of bit serial output from eight PEs is carried out to the output shift registers in the same fashion. The eight global bit lines described in Section 4.3 are used as input and output data lines.

To avoid the delay of the global lines for the cache operations due to additional switches, we can place other routing tracks into the space between global bit lines, such as feedthrough. Since we have already added one additional metal layer for the global bit lines, this layer can be used to route additional lines. This implies that we have enough vertical routing tracks in this architecture. This computation is repeated 8 times, once for each row, for 8 rows of an 8×8 image. In the mean time during each computation, the next set of input data is fetched in another set of input registers and the previous output data is written into an additional memory. All the intermediate results from the 1-D transform must be stored in a memory and then loaded for the second 1-D transform which performs the same computations to complete a 2-D transform. Therefore, 2-D DCT/IDCT is computed with two additional memories similar to the Convolution function. A data flow diagram of the computation process for 8×8 2-D DCT is depicted in Figure 6.3.

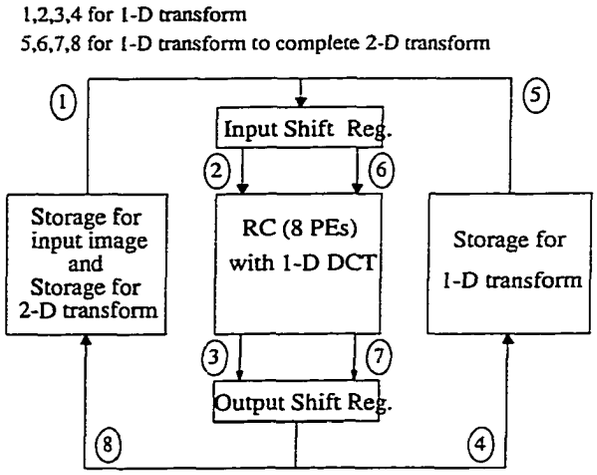


Figure 6.3 Data flow of the computation process for 8×8 2-D DCT transform

Several other opportunities for reconfigurable units exist in this architecture as described below.

- The flexible use of the shift registers working as normal registers for the main on-chip ALU which is an approach similar to [9]. Since the registers are not an integral part of the cache operations, it is easy to convert them.

- Although the width of coefficients in the ROM configuration is fixed at 16 bits in this example, the coefficient width is flexible in this architecture between one and sixteen. Moreover, the width of input elements can be easily extended by adding more shift registers without modifying the current configuration.

- The emulation of a ROM in the RFC does not imply fixed processing coefficients. Hence, different sets of coefficient values can be loaded using the conventional cache operation for the other distributed arithmetic operations.

6.1.3 Reconfigurable cache merged with multi-context configurations

Since we implement Convolution and DCT/IDCT in the same RFC framework, we can merge the two functions into one RFC. With the concept of multi-context configurations mapped into multi-bit output LUTs and individual interconnections, the RFC can be converted to either of two function units. The placement of LUTs is shown in Figure 6.4.

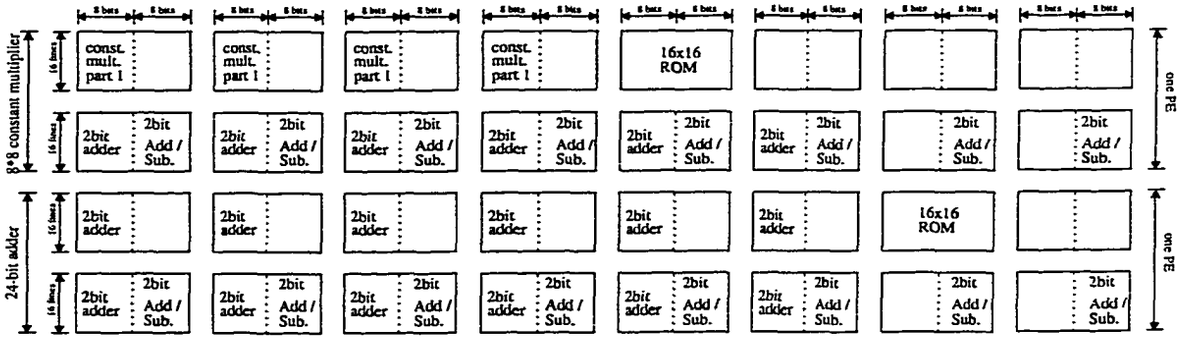


Figure 6.4 An array of LUTs for the combined RFC with Convolution and DCT/IDCT

A combined RFC with two functions takes less area than the sum of the areas of two individual function units because the additional area cost is due to interconnections only. The logic is absorbed in the available cache memory based LUTs. The required interconnection for each function is placed independently together in the combined RFC, which implies that there is no sharing of interconnection between the two functions. As described in Section 5.1, we

use fixed interconnection since it takes less area and propagation time than the programmable interconnection. The actual area of the RFC framework (base array cache) and interconnection is shown in the last part of Section 6.2.

6.2 Area

To measure the actual area overhead of cache array-only for both *memory cell array cache* and *base array cache*, we experimented with layouts of the RFC with only Convolution, only DCT/IDCT, and with both functions. As we compare the access time of RFC for cache operations in two cache models, *the memory cell array cache* and *the base array cache* in Section 4.4, the area overheads are estimated with respect to the two cache models.

Figure 6.5 represents one stage of the Convolution unit in the RFC described above. The pipeline registers are not shown in the figure. According to our layout experiment, the total area of the reconfigurable module including the pipeline registers with an FIR filter, which supports up to 256 taps, is 1.53/1.12 times the area of data array in *the memory cell array cache/the base array cache* without other logic components, respectively (described in Section 4.4). To see the exact area overhead of memory array-only, We consider the area overhead of RFC with respect to the base area of only the data cache array, which does not include the additional cache logic – specifically, row/column decoders, tag/status-bit part, and sense amplifiers. The *percentage* of RFC area overhead would appear to be even lower, had we inflated the base area by including the area for these logic components. However, the actual area overhead remains the same.

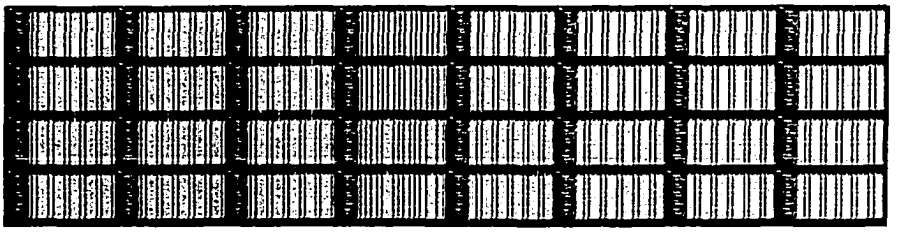


Figure 6.5 A possible layout of RFC for one stage of the Convolution

For the DCT/IDCT function unit on an RFC, the required interconnection is fixed just as in the Convolution cache module. In the DCT/IDCT function, no complicated routing is required and the number of LUT rows in the RFC is less than that for the FIR filter while the number of registers is higher. Thus, according to our experimental layout for DCT/IDCT, the total area of the DCT/IDCT module is 1.48/1.09 times the area of data array in *the memory cell array cache/the base array cache*, respectively, including the accumulating registers and the shift register at the in/out port. Again, those basic units, such as row/column decoders, tag/status-bit part, and sense amplifiers, are not included in this comparison as mentioned above.

In Table 6.1 and Table 6.2, the area overhead of FIR and DCT/IDCT in the RFC is compared with designs for these functions previously reported in the literature. The designs we compare within this thesis are from the literature of the last 10 years. We compare our result to the best-area implementations in the literature. We have estimated the area for an RFC in λ^2 by scaling λ from $0.25\mu m$ to $12\mu m$ (some of them are not shown in the tables). As we explained in Section 4.4, the area overhead for RFC is also estimated with respect to the two base cache architectures. For the memory cell array cache, the area overhead includes the dedicated decoders, switches, RFC-interconnect, and the required registers while the area overhead in the base array cache with the parallel decoding and segmented bit/word lines consists of only the interconnect and the registers. Some of these designs include pads area. For a fair comparison, only the core sizes are listed in both tables by estimating the area of the core part of the entire chips. Also, the area overhead of RFC in the tables is the area of only additional units to support the functions implemented. In other words, the original cache area is not included in the area overhead. The core area of design in [64] shown here is estimated in [62].

In Table 6.2, the core area of 1-D IDCT in design [66] excludes I/O pads and buffer area. We scale the reported total area by the proportion of the reported core area to the reported total area. The area of FIR and DCT/IDCT in the RFC includes all the required registers such as pipeline registers for FIR and accumulating/shift registers for DCT/IDCT.

Table 6.1 Area comparison of FIR Filters and RFC overhead for FIR

	Yoshino et al. [62]	Hatamian-Rao [63]	Ishikawa et al. [64]
Number of Taps	64	40	15,19
Coefficient Word-length	14 bits (fixed)	12 bits (programmable)	8 bits (fixed)
Technology	0.8μm BiCMOS	0.9μm	1.2μm
Core Area	**49** mm^2	**22** mm^2	**80** mm^2

	FIR Filter in the RFC (area overhead of the cache)		
Number of Taps	256 taps (with 8 physical taps)		
Coefficient Word-length	8 bits		
Technology	0.8μm	0.9μm	1.2μm
Area Overhead[1]	**11.28** mm^2	**14.28** mm^2	**25.39** mm^2
Area Overhead[2]	**3.45** mm^2	**4.37** mm^2	**7.77** mm^2
% for area overhead[1]	53% of area for array-only of the memory cell array cache		
% for area overhead[2]	12% of area for array-only of the base array cache		

[1] Area overhead of the "only" memory cell array

[2] Area overhead of interconnections and registers regarding the "only" base cache array described in Section 4.4

Table 6.2 Area comparison of DCT/IDCT chips and RC overhead for DCT/IDCT

	Masaki et al. [66]	Madisettii-Willson [67]	Uramoto et al. [68]
Function	1-D IDCT	8×8 DCT/IDCT	8×8 DCT/IDCT
Technology	0.6 μm	0.8μm	0.8μm
Core Area	**9.4** mm^2	**10** mm^2	**21.21** mm^2

	8×8 DCT/IDCT in RC (area overhead of cache)	
Technology	0.6μm	0.8μm
Area Overhead[1]	**5.9** mm^2	**10.5** mm^2
Area Overhead[2]	**1.51** mm^2	**2.68** mm^2
% for area overhead[1]	48% of area for array-only of the memory cell array cache	
% for area overhead[2]	9% of area for array-only of the base array cache	

[1] Area overhead of the "only" memory cell array

[2] Area overhead of interconnections and registers regarding the "only" base cache array described in Section 4.4

Most of the reported FIR filter designs have fixed coefficients with as many physical MACs as the number of taps. Although coefficients are programmable in [63], only 40 taps can be supported for various types of filter. Besides, the time taken for run-time reconfiguration in a serial fashion is high due to the limited number of pins. The time of run-time reconfiguration of coefficients in the RFC is much smaller because multiple LUT writes are achieved per cache write operation. For a fair comparison, the area per tap can be calculated roughly in each filter by dividing the core area by the number of taps. According to the area per tap, the area of a tap in the RFC is larger than others with respect to the memory cell array cache while the area per tap in the RFC is smaller than others with respect to the base array cache area overhead. Although only 8 taps are implemented physically in the RFC, the FIR cache module can support up to 256 taps with fast configuration not visible to the application.

Since some of the filters have a different word length, we compare the area of 16×16 constant coefficient multiplier and 32-bit accumulator (MAC) implemented in the RFC with the same word length of MAC as presented in [69, 70]. Since constant coefficient multipliers are used in most DSP and multimedia applications, we implemented a 16×16 constant coefficient multiplier, one MAC stage for FIR. The MAC area is estimated based on the number of LUT rows used and interconnection in RFC. In our experimental layout, the MAC (16×16) area in the RFC is less than or equal to two times the area of one MAC stage of Convolution (8×8) in the RFC. This area is smaller than that of the existing MACs as shown in Table 6.3 in both cases. This implies that an FIR filter with 16-bit word-length can be easily implemented in the RFC with a similar area overhead for four physical taps. However, it can still support up to 256 taps.

Note that the designs reported in [62] and [63] implement FIR with 14-bit and 12-bit coefficients, respectively, while we report RFC area overhead for an 8-bit coefficient design. It is hard to develop a precise analytical model for the area parameterized by the number of bits in a coefficient. Some parts (such as multiplier) may scale non-linearly with the number of coefficient bits depending on the algorithm. Some parts would scale sub-linearly such as control and global routing. For an approximate comparison, we assume that the area scales linearly

Table 6.3 Area comparison of Multiplier-Accumulator's and RC overhead for one MAC stage

	Izumikawai et al. [69]	Lu-Samueli [70]
Size of In/Out	16b×16b/32bits	12b×12b/27bits
Technology	$0.25\mu m$	$1.0\mu m$
Area	**0.55** mm^2 (core)	**9.30** mm^2 (chip)

	MAC in the RC (area overhead)	
Size of In/Out	16b×16b/32bits	
Technology	$0.25\mu m$	$1.0\mu m$
Area Overhead[1]	**0.28** mm^2	**4.41** mm^2
Area Overhead[2]	**0.08** mm^2	**1.35** mm^2
% for area overhead[1]	51% of area for array-only of the memory cell array cache	
% for area overhead[2]	11% of area for array-only of the base array cache	

[1] Area overhead of the "only" memory cell array

[2] Area overhead of interconnections and registers regarding the "only" base cache array described in Section 4.4

in coefficient width. Hence, the 8-bit version of [62] would take area $28mm^2$ ($\frac{8}{14} \times 49mm^2$) and for [63] the area would be $14.7mm^2$ ($\frac{8}{12} \times 22mm^2$), which are comparable to the RFC area overhead for FIR. The main advantage of RFC for FIR is the reconfigurability which allows the RFC-FIR to have a virtually infinite number of taps unlike other customized FIR chips. The number of taps is also configured with a faster reconfiguration time. We, therefore, conclude that the area per tap for RFC is comparable to that of the customized FIR chips.

The area of the previous designs for DCT/IDCT in Table 6.2 is larger than the proposed DCT/IDCT cache module except [67] with respect to area overhead of the memory cell array cache. The 2-D DCT/IDCT functions are implemented with a similar procedure as in the DCT/IDCT cache module - two 1-D DCT steps. Since the DCT function is implemented using a hardwired multiplier in [67], the area is smaller than the cache module with respect to the area overhead of the memory cell array cache. However, the area overhead with respect to the base array cache is smaller than all the previous designs shown in the table. The DCT function in [68] has two 1-D DCT units, so the area of one 1-D DCT unit is roughly half of

the overall area which is still larger than the RFC overhead.

In the combined multi-function RFC, each function needs a fixed interconnection topology. Therefore, the total area of interconnection occupied by the two functions in the combined RFC is the sum of the individual interconnection areas for Convolution and DCT/IDCT. According to our experimental layout of the combined cache, the total area of the RFC with two functions is 1.63/1.21 times the area of data array in *the memory cell array cache/the base array cache*, respectively, with all the required registers and without other components described above. The actual area of the combined cache module is shown in Table 6.4.

Table 6.4 Area overhead of the combined reconfigurable cache

Function	FIR, DCT/IDCT			
Technology	$0.6\mu m$	$0.8\mu m$	$1.0\mu m$	$1.2\mu m$
Interconnect & registers for FIR	**1.94** mm^2	**3.45** mm^2	**5.39** mm^2	**7.77** mm^2
Interconnect &registers for DCT/IDCT	**1.51** mm^2	**2.68** mm^2	**4.19** mm^2	**6.04** mm^2
RFC framework (base array cache)	**4.41** mm^2	**7.83** mm^2	**12.24** mm^2	**17.62** mm^2
Area Overhead[1]	**3.45** mm^2	**6.13** mm^2	**9.58** mm^2	**13.81** mm^2
Area Overhead[2]	**7.86** mm^2	**13.96** mm^2	**21.82** mm^2	**31.43** mm^2
% for area overhead[1]	63% of area for array-only of the memory cell array cache			
% for area overhead[2]	21% of area for array-only of the base array cache			

[1] Area overhead of the "only" memory cell array

[2] Area overhead of interconnections and registers regarding the "only" base cache array described in Section 4.4

Since the decoders for LUTs account for most of the area overhead in the RFCs, adding more interconnection does not add much area in the combined RFC. The base array cache described in Section 4.4 consists of dedicated 4-to-16 decoders, four address lines, and a number of switches to connect the local bit lines to the global bit lines. The area of combined RFC is smaller than the sum of smallest areas in the existing FIR and DCT/IDCT function units in both cache models. This implies that we can add additional multiple functions in the

existing RFC with a relatively small area overhead. The interconnection area for individual functions is also listed in Table 6.4. Moreover, since some part of the area for routing tracks between the two functions is overlapped, for example, adders, constant multiplier, and ROMs, the area of interconnection in the combined RFC may be less than the sum of two individual interconnection areas. The fixed interconnection for the functions can be efficiently routed and does not take much area. The placement and routing of the RFC has been done manually as a first cut. We can expect the area overhead to reduce further if we place and route carefully.

6.3 RFC with different cache organizations

We described the RFC for computations based on an 8KB size of cache memory with 512 sets and 16-byte cache line in Section 6.1. In this section, we show how different cache organizations for an 8KB sized cache memory work for the RFC. The RFC for the filtering operations, shown as an FIR filter in Section 6.1.1, can be implemented in various cache organizations in 8KB. Figure 6.6 shows the implementations of filters using the RFC in different cache organizations with respect to the number of sets and size of cache line. In the figure, one block of 2KB represents a MAC (Multiply-and-Accumulate) stage for a 16-bit constant coefficient multiplier and a 32-bit accumulator implemented in the RFC while an 8-bit constant coefficient multiplier and a 24-bit accumulator is implemented in 1KB as presented in Section 6.1.1. This mapping organization can be applied to any other computations (such as DCT/IDCT) as long as the required number of LUTs to implement a function is satisfied in a cache memory size. The area overheads of RFCs in the different organizations are similar to that of the RFC (8KB with 512 sets and 16-byte line) shown in Section 6.2 because the number of LUTs is the same and the interconnection is not much varied. The arrows in the figure represent a flow of pipeline stages to perform a filtering operation. Note that eight MAC stages are mapped into a 16KB cache memory with 256 sets and 64-byte line using a mapping scheme similar to 8KB sized cache memory. This indicates that a higher number of stages or processing elements can be added into a larger sized cache memory without any significant modification. In addition, if the number of sets in a cache reduces, the number of cache lines to be configured reduces

since more LUTs can be written simultaneously using a cache write operation for a line. This is one advantage of using cache memory for LUT-based reconfigurable logic. The shaded parts in each MAC in the figure show the portion to be reconfigured partially for the constant coefficient multiplier, especially in the FIR filter. Thus, the amount of time for the configuration is determined by the number of sets and cache line size, not the number of LUTs for a function.

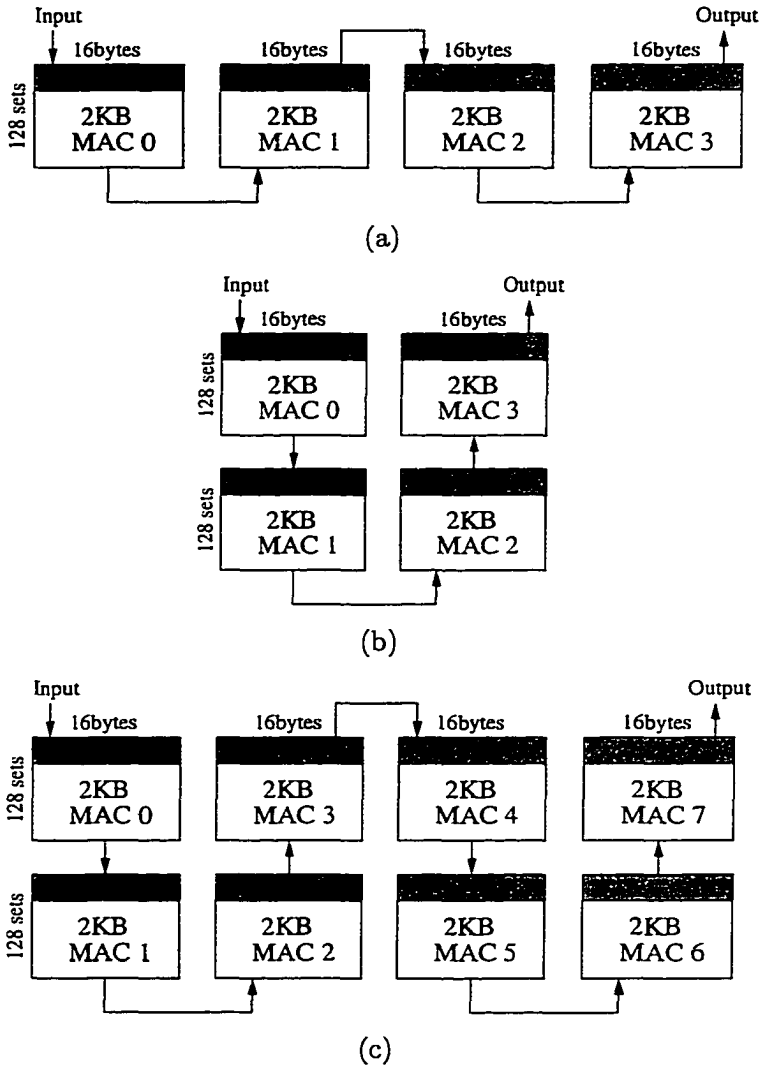


Figure 6.6 A filter (16mult-32acc) with (a) 128 sets and 64 bytes/line (8KB); (b) 256 sets and 32 bytes/line (8KB); (c) 256 sets and 64 bytes/line (16KB)

6.4 Execution time

In this section, we show the execution time comparison for Convolution and DCT/IDCT as presented in [28, 29]. In this experiment, we compare the performance of these functions on a Sun UltraSPARC workstation with their execution time derived from a model using an RFC. These models are based on various computing parameters, such as the number of taps and the size of images. The RFC computing time model assumes that all input/output data is available in cache memory with no stalls. Hence, the modes based performance is a best case scenario. The simulation results of a processor integrating RFCs are shown in Chapter 7.

6.4.1 Convolution

We compare the execution time of the FIR filter using a reconfigurable functional cache (RFC) to a conventional general purpose processor (GPP) running the algorithm in Equation 6.1. Since the RFC may have to be flushed, we show the results for the following two cases. In the first case, no data in the cache needs to be written back to main memory before it is reconfigured as the function unit, for example, caches with write-through policy. In the second case, the processor has to flush all the data in the cache before configuring it (i.e. written back to the main memory). The extra time is denoted by the '*flush time*' and is required for write-back caches.

The total execution time of the Convolution in the RFC consists of configuration and computation times. The configuration time includes the times for adder and constant coefficient multiplier configuration. In the second case, the cache flush time is also added to the configuration time. The actual parameter values to compute the times are given in Table 6.5.

We chose the values to be as conservative as possible with respect to SPARC II*i* processor cycle time at 270 MHz [51] (where the GPP simulation was performed). The access time for the data cache in SPARC II*i* processor is 1 cycle in a pipelined fashion (it is a 16KB direct mapped cache with two 16B sub-blocks per line). In a typical processor, this access time can be anywhere from 1-2 cycles. Hence, we chose 3 cycles for the cache access time in RFC for a conservative model. Had we chosen a lower cache access time (1 or 2 cycles), the RFC

execution time would appear to be even more favorable since other parameters, such as LUT read time in RFC, were based on the cache access time - 3 cycles (12ns). The main memory access time is 20 cycles. The parameters for the cache structure are based on an 8KB size cache with 8 words per block and 16 bits per word (L_{cache}, L_{LUT}, *and* W_n). Since 8 words in a cache block are stored in an interleaved fashion, each bit of one word is stored every 8 bits. The 1st and 9th bits of a LUT content can be written in the LUT simultaneously by writing one word (parameter m=2). The computation time of one stage/PE in the RFCs is chosen by the following factors. Each stage in the Convolution function unit requires three LUT reads (with additional time for propagation through a number of multiplexers) while each PE in DCT/IDCT unit does two LUT reads with additional time for multiplexers. We use read time for a LUT of 8ns with the multiplexer propagation time - less than the cache access time because the LUT is much smaller and faster than the 8KB cache memory. The expressions for the times are presented below.

- Config. Time for adder
  $= [\ (R_{mem/cpu})(\frac{a}{m})(L_{LUT}) + (R_{cache/cpu})(\frac{a}{m})(L_{cache} - L_{LUT} \times S)\ ] \times T_{cpu}$
- Config. Time for constant multiplier
  $= (R_{mem/cpu})(\frac{a}{m})(L_{LUT})(TAP) \times T_{cpu}$
- Cache Flush Time
  $= (R_{mem/cpu})(W_n)(L_{cache}) \times T_{cpu}$
- Computation Time
  $= [\ (\frac{TAP}{S}) \times (X + 2S - 1)\ ] \times T_{1\_stage}$

In the computation time, we add $2S$ instead of S for the initial pipeline steps because we exploit the double pipelined input data in each stage of the Convolution as shown in Figure 6.1(a). In addition, we separate the configuration time for adders and multipliers. The reason for this is that only one set of data for a LUT is necessary when reconfiguring the LUTs

for adders because the contents of all the LUTs are the same, while a different configuration data is necessary for multipliers. The time for storing and loading input and intermediate data can be overlapped with the computation time. Therefore, data access time for the computation is not added.

Table 6.5 Parameters for the RFCs

Parameter	Description	Values
T_{cpu}	One cpu cycle time	4ns
T_{1_stage}	The time to complete the computation in one stage/PE	24ns/16ns
$R_{mem/cpu}$	Main memory access latency	80ns
$R_{cache/cpu}$	Cache memory access latency	12ns
L_{cache}	Number of cache lines in the cache	512
L_{LUT}	Number of contents in a LUT	16
W_n	Number of words per cache block	8
a	Number of bits required to configure a content of one LUT for a 2-bit adder with 3bits for carry=0 and 3bits for carry=1 & for the half of a 4x8 constant coefficient multiplier	6
r	Number of bits required to configure a content of LUT for a ROM	16
m	Number of bits to be written by one word when configuring	2
S	Number of taps/PEs implemented in the RC	8
Parameters for Convolution		
TAP	Number of taps	8 - 256
X	Number of data	64 - 8192
Parameters for DCT/IDCT		
W_d	The width of input elements	8 bits
N	The size of a basic block image	8
IMG	The size of an entire image	8x8 - 1920x1152

The execution times for RFC and the GPP are shown in Table 6.6. We assume that all the input data fit into a data cache for both the RFC and the GPP computations according to the following observation. We traced the number of cache misses in the GPP for all the cases in Figure 6.6. From the trace we found that regardless of the number of taps and data

elements in the computation, the number of cache misses does not vary with the execution time. Therefore, we neglected the effect of the cache miss penalty in the comparison. We simulated Convolution with floating point variables instead of integers which leads to faster processing in GPPs. The choice between memory cell array and base array determines cache access time in GPPs. As we explained in Section 6.6. the RFC based on memory cell array will give smaller access time in a GPP even for other applications while the RFC based on base array will increase the cache access time by 1-2%. We have assumed the cache access time in the GPP and in the processor with RFC to be the same for both cache types (memory cell array or base array).

Table 6.6 Comparison of execution time of Convolution between SPARC and RFC (μsec)

No. of Taps	No. Data elmt	SPARC (269.8 MHZ)	RFC w/o memory flush		Ratio	RFC w/ memory flush		Ratio
			config	compute		config	compute	
8	256	384.58	48.38	6.50	7.01	376.06	6.50	1.01
	1024	1553.36		24.94	21.19		24.94	3.87
	4096	6307.85		98.66	42.90		98.66	13.29
	8192	12605.09		196.97	51.38		196.97	22.00
16	256	735.18	79.10	13.01	7.98	406.78	13.01	1.75
	1024	2963.83		49.87	22.98		49.87	6.49
	4096	11928.19		197.33	43.15		197.33	19.75
	8192	23893.06		393.94	50.51		393.94	29.84
32	256	1435.98	140.54	26.02	8.62	468.22	26.02	2.91
	1024	5792.27		99.74	24.11		99.74	10.20
	4096	23270.04		394.66	43.48		394.66	26.97
	8192	46540.36		787.87	50.13		787.87	37.05
64	256	2840.07	263.42	52.03	9.00	591.10	52.03	4.42
	1024	11465.30		199.49	24.77		199.49	14.50
	4096	45898.64		789.31	43.60		789.31	33.25
	8192	91831.01		1575.74	49.93		1575.74	42.38
128	256	5651.69	509.18	104.06	9.22	836.86	104.06	6.01
	1024	22737.80		398.98	25.04		398.98	18.40
	4096	91360.32		1578.62	43.76		1578.62	37.82
	8192	182750.06		3151.49	49.92		3151.49	45.82
256	256	11265.51	1000.70	208.13	9.32	1328.38	208.13	7.33
	1024	45287.32		797.95	25.18		797.95	21.30
	4096	183016.17		3157.25	44.02		3157.25	40.80
	8192	368557.75		6302.98	50.46		6302.98	48.30

The speedup of RFC over the GPP for Convolution is shown in Figure 6.7. Our results show that the RFC provides a better performance improvement than the GPP as the number of data elements increases. Figure 6.7 shows that the performance improvement is almost independent of the number of taps without memory flush in (a). The ratio of the computation time with less taps decreases with memory flush in (b) because the flush time affects the ratio of the total execution time more with the decrease in the number of taps.

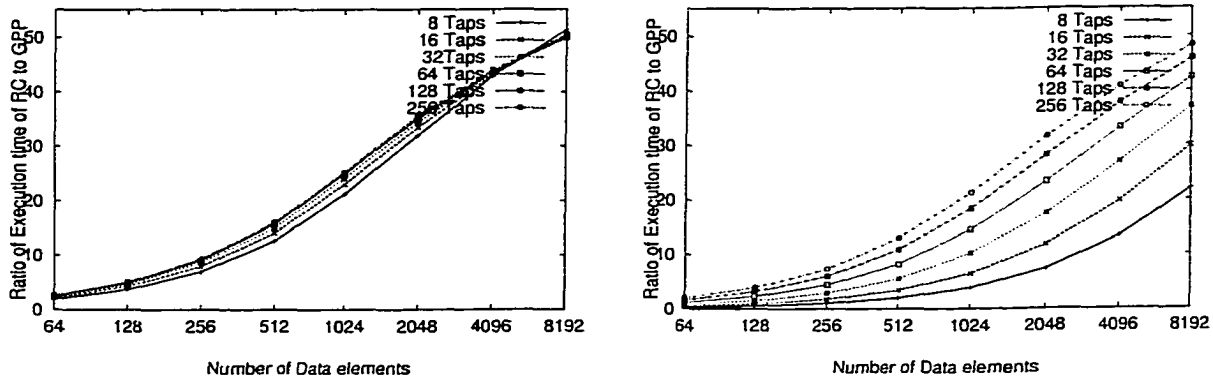


Figure 6.7 Ratio of execution time of RFC and GPP for Convolution: (a) without memory flush; (b) with memory flush before converting into the computing unit

6.4.2 DCT/IDCT

As described in Section 6.1.2, the 2-D DCT/IDCT can be completed by two 1-D transforms. This procedure is similar to the data caching scheme which is adapted for the FIR filter module (i.e. two additional memories for processing with intermediate data). We compare the execution time of the 2-D transforms in RFC and the GPP executing the fast DCT algorithm described in Section 6.1.2. As in the previous example, the two cases of cache '*flush time*', no cache flush and cache flush, are considered in this section.

The total execution time of the DCT(IDCT) in the RFC consists of configuration and computation times. The configuration time includes the writing times for the contents of ROMs and adders. In addition, in the case of cache flush, the cache flush time is also to

be added in the configuration time. The actual parameter values to compute the times for this function used are the same as for the Convolution in Table 6.5. The expressions for the execution times are presented below.

- Config. Time for accumulators and pre(or post)-adders/subtracters
  $= [\ (R_{mem/cpu})(\frac{2a}{m})(L_{LUT}) + (R_{cache/cpu})(\frac{2a}{m})((S+2) \times L_{Lut})\ ] \times T_{cpu}$
- Config. Time for ROM
  $= [(R_{mem/cpu})(\frac{r}{m})(S \times L_{LUT})] \times T_{cpu}$
- Cache Flush Time
  $= (R_{mem/cpu})(W_n)(L_{cache}) \times T_{cpu}$
- Computation Time
  $= [2\times(\text{1-D transform})]\times(\frac{\text{Image size}}{\text{Basic block size}})\times T_{1\_stage} = [2\times(N+W_d\times N)]\times\frac{IMG}{N\times N}\times T_{1\_stage}$

The cache '*flush time*' is the same as earlier. Configuration data needs to be written to all the PEs once only because all the data elements in an image are processed with the same coefficients using the distributed arithmetic. The configuration procedure of the Convolution in the previous section is applied to DCT/IDCT. As described earlier, the time of loading and writing all the in/out data from/to memories can be overlapped with the computation. Thus, only the initial loading and the final writing time, which is overlapped in the transition of data set, is added to the computation time of each 8 × 8 1-D transform for data access time. In this configuration, the adder is used as both a 16-bit adder and a 16-bit subtracter with 2 sets of configuration data. Since only one of the pre/post-adders (subtracters) is necessary for DCT and IDCT, respectively, the configuration time of pre-(or post)adders/subtracter with the same configuration scheme is added in the execution time.

The execution times of the GPP and RFC are shown in Table 6.7. The assumption regarding the cache misses of data mentioned in Section 6.4.1 has been applied to this simulation. Therefore, the main memory access time is not considered for in/out data of the computation. For a larger size of image than the basic block, 8 × 8, we partitioned the entire image into

a number of basic block images. We assume that the cosine weighted factors are pre-stored as coefficients in an array when the GPP processes the DCT/IDCT, which means the actual cosine coefficient computation is not performed in the GPP. It is much faster than the computation with the actual cosine factors. Again, floating point variables are employed in our simulation of DCT/IDCT for faster processing in the GPP.

Table 6.7 Comparison of execution time of the DCT/IDCT between SPARC and RFC (μsec)

Size of Image	No. of 8 × 8 2-D (I)DCT	SPARC (269.8MHZ)	RFC w/o memory flush		Ratio	RFC w/ memory flush		Ratio
			config	compute		config	compute	
8 × 8	1	168.86	101.12	2.30	1.63	428.80	2.30	0.39
16 × 16	4	641.85	101.12	9.22	5.82	428.80	9.22	1.47
32 × 32	16	2346.78	101.12	36.86	17.01	428.80	36.86	5.04
64 × 64	64	9276.67	101.12	147.46	37.32	428.80	147.46	16.10
128 × 128	256	37498.88	101.12	589.82	54.27	428.80	589.82	36.81
256 × 256	1024	148428.84	101.12	2359.30	60.33	428.80	2359.30	53.24
512 × 768	6144	859776.81	101.12	14155.78	60.31	428.80	14155.78	58.95
1920 × 1152	34560	4850821.01	101.12	79626.24	60.84	428.80	79626.24	60.59

According to the speedup in Figure 6.8, the RFC for DCT/IDCT has a better performance improvement over the execution time of the GPP as the size of input image increases. The performance improvement is roughly independent of the memory flush in the larger size of images. Since the computation is ROM based, only the initial configuration is necessary. Thus, the larger sizes in the results, 512 × 768 (TV-image) and 1920 × 1152 (HDTV), do not rely on the flush time. For MP@HL (Main Profile at High Level) decoding, the maximum allowable time to process a macroblock is 4.08μs [66]. The result shows that it is possible to process a block in 2.30μs.

6.4.3 Multi-context reconfigurable functional cache

There is no difference between individual and combined caches in terms of the execution time. However, the combined cache may have a slightly higher propagation delay due to longer wires caused by the inclusion of interconnections, in our instance, this causes 1.6% increase in

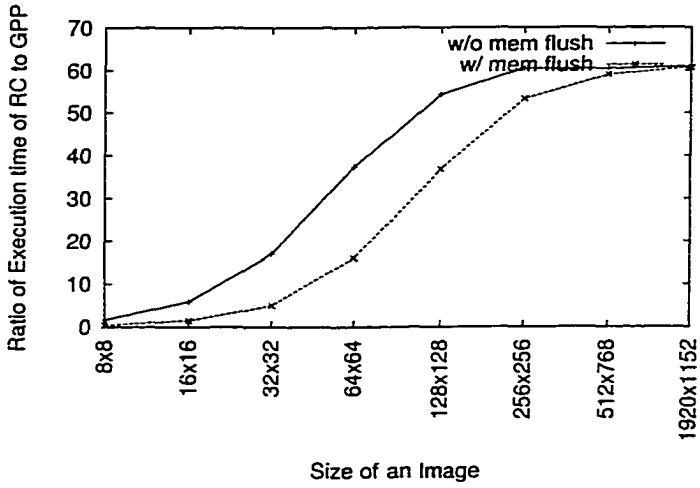


Figure 6.8 Ratio of execution time of RFC and GPP for DCT/IDCT with and without '*flush time*'

cache access time. Therefore, we can assume that both individual and combined RFCs have almost the same execution performance.

CHAPTER 7. SIMULATION

In this chapter, we provide the results of our experiments. We describe the functions implemented in RFC in Section 7.1. The simulation environment and parameters (methodology) are described in Section 7.2. Finally, we measure and compare the performance of a base processor without and with RFCs in Section 7.3.

7.1 Computing units configured using RFC

In the RFC based on the organization (LUT-based reconfigurable logic) described in Chapter 4, we implemented two primitive functions in DSP and multimedia applications, Multiply-and-Accumulate (MAC) and Distributed Arithmetic (DA). The MAC consists of constant coefficient multiply and accumulate. The DA unit performs a sum of products with a look-up table based multiplication in a bit-serial fashion and consists of a ROM and a shifting accumulator. This kind of function unit can be used in many of the important DSP and media applications [72] with a small amount of area.

The MAC implemented in RFC performs a filtering operation with a sliding window of coefficient with 8 or 16-bit integer and fixed-point data. A floating-point computation in software can be performed in the DA unit with a binary format of floating-point numbers. In RFCs, four MACs performed in a pipelined fashion and eight processing elements (PEs) processed in parallel are implemented. One MAC contains a 16-bit multiplier and a 32-bit accumulator while one PE includes a 16×16 ROM and a 16-bit shift accumulator. More details for the actual implementation can be found in Chapter 6. The comparison with the previous ASIC chips for FIR and DCT in Section 6.2 shows that the area overhead of RFC is smaller than the core area of those units.

7.2 Experimental methodology

7.2.1 Benchmarks

Discrete Cosine Transform (DCT) is the most efficient technique in image encoding and compression schemes. Two 1-D DCT/IDCT processes can represent a 2-D DCT/IDCT process. Convolution is a DSP algorithm that multiplies two (integer) arrays of dimension-two and is a common requirement in signal processing and image processing for pattern recognition, edge detection, etc.

Using these two algorithms mapped to the computing units in RFCs, we simulated the most common media processing applications - Motion Picture Experts Group 2 (MPEG2) - *mpeg2encoder* [73] and *mpeg2decoder* [73, 74], Joint Photography Experts Group (JPEG) - *cjpeg* encoder [73, 74] using DCT/IDCT and Finite Impulse Response (FIR) [75], Infinite Impulse Response (IIR) [75] using the concept of Convolution algorithm. The inputs to the benchmarks are shown in Table 7.1.

Table 7.1 Benchmarks used in this thesis

Benchmark	Description with input
mpeg2enc	MPEG2 encoding for four 352×240 frames from YUV components
mpeg2dec	MPEG2 decoding for DVD (surfer.m2v) into YUV components
cjpeg	JPEG encoding for 1024×768 image (8-bit vigo.ppm) into vigo.jpg
FIR	16 / 256 taps with 16K data elements (need partial configurations - total 4 / 64 times)
IIR	two sets of coefficients (total 9) with 16K data elements

The data type for mpeg2enc/dec and cjpeg is an 8-bit integer and a 16-bit integer for FIR and IIR. The benchmarks are compiled using Simplescalar gcc [71] from the source codes in UCLA mediabench while some of large application data (for example, DVD) are from Berkeley Multimedia Workload [74]. The number of filtering coefficients is variable in FIR while the IIR performs an auto-regressive moving-average (ARMA) filter with four auto-regressive filter coefficients and five moving-average filter coefficients [75]. Both are processed in 16-bit

multiplication and 32-bit accumulation. The IIR is used as a high pass filter in the VSELP vocoder [75]. In IIR, two sets of coefficient window sliding and multiplying arrays are subtracted. One time configuration for 8 PEs is required in DCT/IDCT. Since the physical number of taps (16/32-bit MAC) implemented in RFC is four, we need a partial reconfiguration as described in Section 5.2.4.4 to perform more than four physical taps in FIR and IIR with intermediate results reused.

7.2.2 Simulator and parameters

The microarchitecture in Simplescalar simulator [71] is chosen as a base processor in our simulation. We modified the C source code in the simulator to support the proposed microarchitecture with the RFCs. The new rfc instructions described in Section 5.2.3 are compiled using existing load/store instructions with the annotated field in Simplescalar instead of modifying the compiler. However, the operation and effect of those instructions are the same as we described earlier. We replaced the original code for the targeted computations in the benchmarks with the code for RFCs shown in Section 5.2.5 to configure and exploit the RFC as a specialized computing unit with the dynamic set associativity. Each benchmark with the RFC is optimized differently with the rfc instructions to perform each core computation, such as DCT, IDCT, FIR, and IIR in the C source code.

The new instructions are embedded into the source code using *inline assembly* code with the substantial address for data in benchmarks. Some of the operations in the corresponding computation are not mapped to RFC due to the lack of available resources (LUTs) in RFC or difficulty of the replacement with the rfc instructions, for instance, moving data from one array to another array. Especially, we could not map one of the coefficient multipliers in IIR due to the lack of available LUTs. In FIR, an array is copied into another array using a conventional C code. In the modified simulator, we traced the exact operations of RFC in each cycle. When input data is loaded into RFC, the simulator holds the data until the RFC is ready and then, processes the data with the exact number of cycles to be taken in RFC. Finally, it stores the output to the memory hierarchy. The rfc instructions replace the floating

point DCT/IDCT function call in mpeg2 encoder/decoder since the hardware implementation is comparable to the floating-point computation in software [68]. Similarly, the integer DCT in cjpeg is replaced with the rfc instructions.

The parameters for the simulation in the base processor are shown in Table 7.2. These are the same for both the processor without and with RFCs. However, we varied the cache parameters, such as size and associativity. Since the cache access time in RFC is not significantly increased or even decreased as estimated in Section 4.4, we assumed the L1 cache latency of the base processor is the same in both the models without and with RFC. The operation latency in FIR/IIR is 3 cycles per pipeline stage for four 4×16 constant multipliers and 20-bit/24-bit/32-bit adders (four LUT reads and propagation of multiplexers). In DCT/IDCT, the latency is 1 cycle per processing element for a 16×16 ROM and a 16-bit adder (two LUT reads and propagation of multiplexers). The DCT/IDCT on the RFC computes the 1-D transform for a row of an 8×8 image with eight PEs in eight cycles (8-bit data) in a bit serial fashion. Additions and subtractions of input data elements are performed before the computation for DCT/IDCT, while after for IDCT. Thus, the number of execution cycles for one row in an 8×8 image in RFC is 9 cycles after loading all the eight input elements. The size of I/O buffers for RFCs is eight.

7.3 Performance measures

We compare the number of cycles taken to execute each application with various cache parameters. First, we show the cache organization with RFCs built in each module (way) as depicted in Figure 5.3 (a) - full dynamic associativity (FDA). Next, we compare the result of the above organization with that of the alternate cache organization shown in Figure 5.3 (b). The further partitioned cache uses one smaller module for the reconfiguration to reduce the performance impact using the partial dynamic associativity (PDA) as described in Section 5.2.2.

FDA: Figure 7.1 shows the total number of execution cycles, which is normalized to the execution cycles with 32K 2-way set associative cache, in mpeg2dec, mpeg2enc, cjpeg, FIR-16taps, FIR-256taps, and IIR. The graph also shows the fraction of cycles taken by the core

Table 7.2 The base processor parameters with RFC and without RFC in Simplescalar simulator

Issue width	8
Instruction window size	RUU: 64 LSQ: 64
Functional units - integer arithmetic - integer multiplier - floating point arithmetic - floating point multiplier	 4 1 4 1
L1 data cache - size - associativity - line size - hit latency	 32KB, 64KB, 128KB 2 / 4-way 64 bytes 2 cycles (same for both)
L1 inst. cache - size - associativity - hit latency	 64KB 2-way with 64B line 2 cycles
L2 cache - size - associativity - hit latency	 1MB 4-way with 64B line 10 cycles
Memory - access latency (pipelined) - memory bus width - memory ports	 60 cycles for 64B 8 bytes 4
TLB - D-TLB - I-TLB - miss latency	 512KB 256KB 30
Branch prediction - bimodal predictor size - branch mis-prediction latency - return address stack size	 2KB 3 8

computation mapped into RFC (DCT, IDCT, FIR, and IIR) in the benchmarks of the base processor without RFC and with RFC (including configuration time).

The performance improvement is shown as an overall speed-up in Figure 7.2. The speed-up of each core computation performed in the RFCs is also shown in the figure. The specialized computing units configured from RFCs improve the performance of each core function significantly. However, the overall speed-up relies on the frequency of those function calls in the entire application. The speed-up for FIR with 256 taps is higher than with 16 taps although the number of partial configurations for 256 taps is 16 times more than that for 16 taps. This implies that the configuration time is not a dominating factor for the performance if a large set of data is processed. The configuration time is compensated by accelerating the performance of core computation.

Various cache organizations: Our simulation shows that the number of execution cycles in the benchmarks using RFC as a specialized computing unit is smaller than that of the base processor without RFC in all the cases except the 32/64KB 2-way set associative cache in mpeg2enc. To see the effect of cache organizations, we simulated the benchmarks with various cache organizations in the base processor as shown in Figure 7.3 (mpeg2dec, mpeg2enc, cjpeg) and 7.4 (fir16, fir256, iir). In the figure, the execution cycles are normalized to the 16KB direct-mapped cache memory. The level-1 data cache miss rate for each benchmark is also shown. Note that the other parameters remain the same with the above simulation.

Unlike the relatively low effect in other cache organizations, the direct-mapped cache for mpeg2enc increases the number of execution cycles by about 16.1% for 16KB and 7.2% for 32KB due to the significant increase in the cache misses. Thus, the performance degradation in using RFC is caused by the reduction in cache capacity to half and in associativity to direct-mapped, when RFC is configured for mpeg2enc. However, in all other cases of the benchmarks, a larger cache memory hardly increases the performance as we stated in Section 1. Note that FIR (16 taps) and IIR with the direct-mapped cache degrades the performance slightly compared to other organizations. However, the use of RFC in 2-way set associative caches does not scale down the performance significantly. The low effect of the dynamic associativity

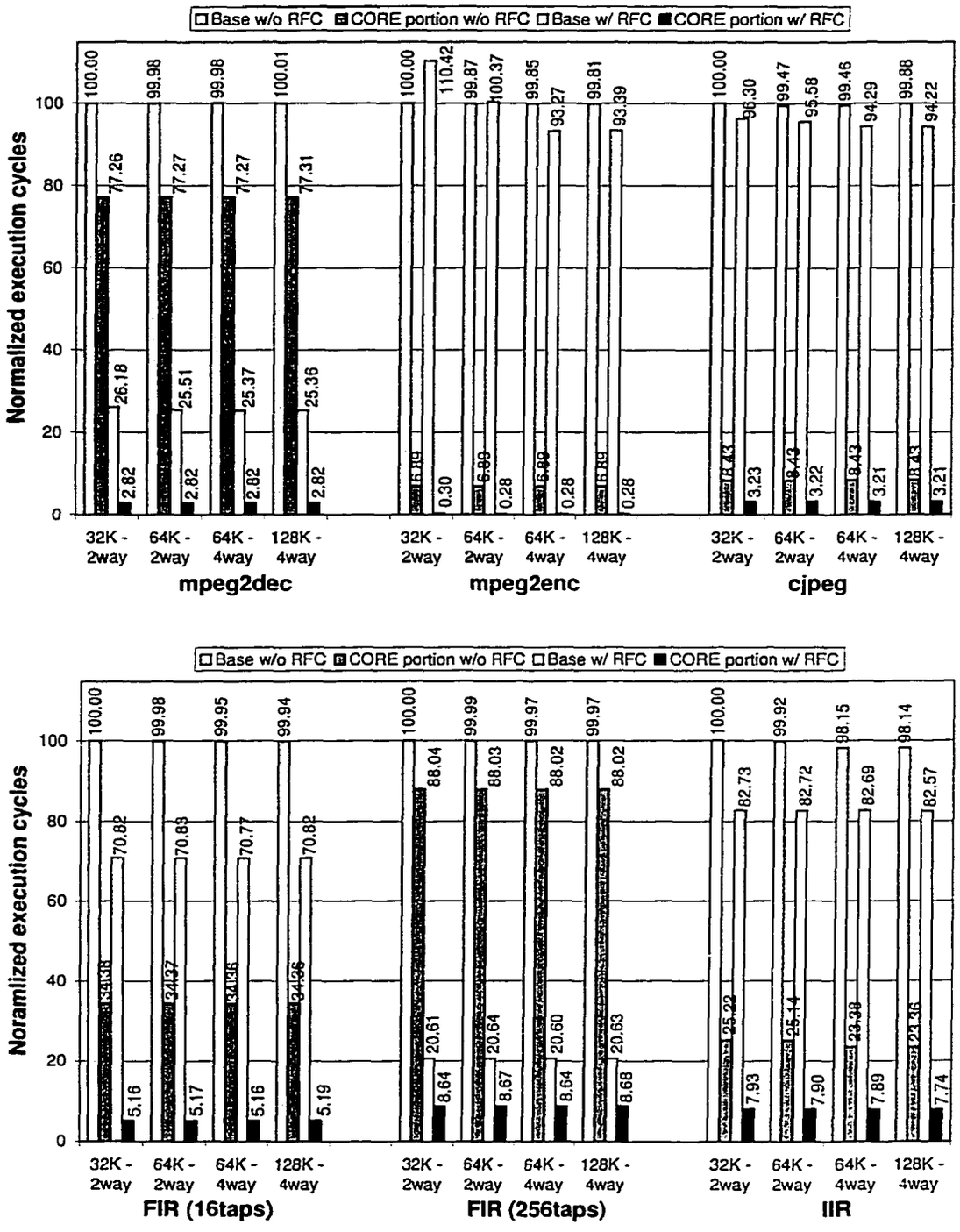


Figure 7.1 Normalized execution cycles in the base processor w/o and w/ RFC

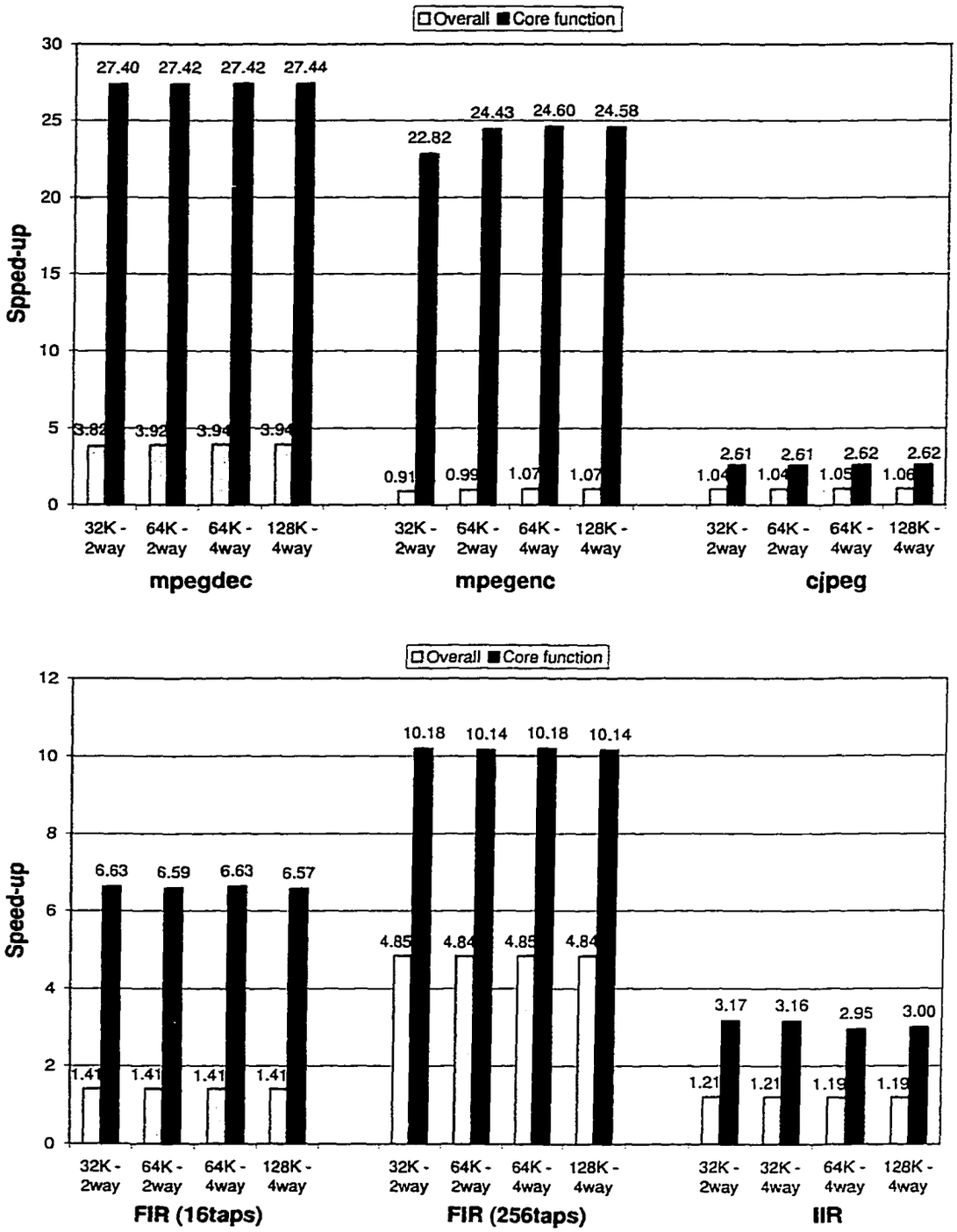


Figure 7.2 Speed-up of benchmarks using RFCs (overall and core computation)

is because many instructions in those applications are replaced with rfc instructions. In most of benchmarks, the direct-mapped cache with a smaller size slightly increases the number of execution cycles. The reason for that is the large number of sets in direct-mapped caches could distribute the blocks to be mapped to the same location (set) into other sets.

PDA: The effect of the dynamic associativity degrades the overall performance in mpeg2enc. We partition a cache memory further with the minimal size of RFC as shown in Figure 5.3(b). This partial dynamic associativity can reduce the impact of the reconfiguration compared to the full dynamic associativity when reconfiguring. The further partition in 64KB 2-way cache for mpeg2enc increases the overall performance by $1.05X$ as shown in Figure 7.5, unlike in 64KB 2-way with the full dynamic associativity ($0.99X$). However, the 32KB 2-way cache with the smaller partition still degrades the performance to $0.98X$. The number of execution cycles in other cases is slightly decreased as compared to those of the full dynamic associativity for RFC. We show the miss rates in the simulations using the partial dynamic associativity for RFC in Figure 7.6. The number of cache misses decreases with the partial dynamic associativity in most cases because only a small portion of address space is mapped to the low associative sets in a cache.

2-way vs. 4-way: The RFC built in a 2-way set associative cache increases the number of cache misses significantly in mpeg2enc. Even with the partial dynamic associativity, the 2-way set associative cache does not produce the performance of the other organizations in mpeg2enc due to the low associativity. To reduce the impact of low associativity, which causes the conflict misses, we simulated mpeg2enc with the same size of cache memory (32KB), but with 4-way associativity (8KB-RFC in each module). In Figure 7.7, the execution cycles with various cache organizations of 32KB are normalized to that with the 32KB 2-way associative cache. The results show that the 4-way set associativity improves the overall performance similar to that of other cache organizations in Figure 7.1. This implies that the conflict misses of data in the same address space are reduced significantly by increasing the associativity. Therefore, we conclude that at least 4-way associativity is preferred to embed the RFC into a

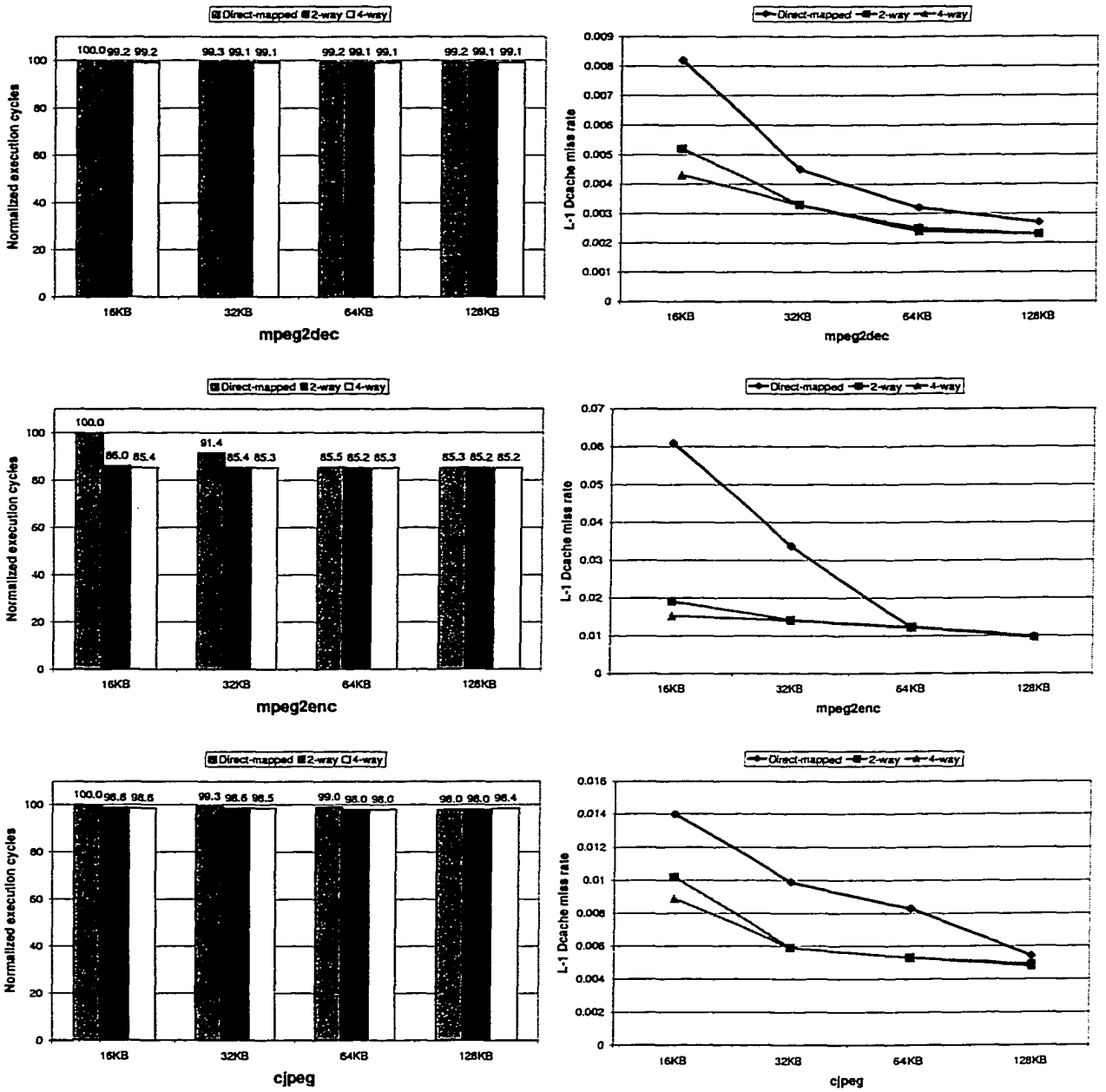


Figure 7.3 Normalized execution cycles with various cache organizations

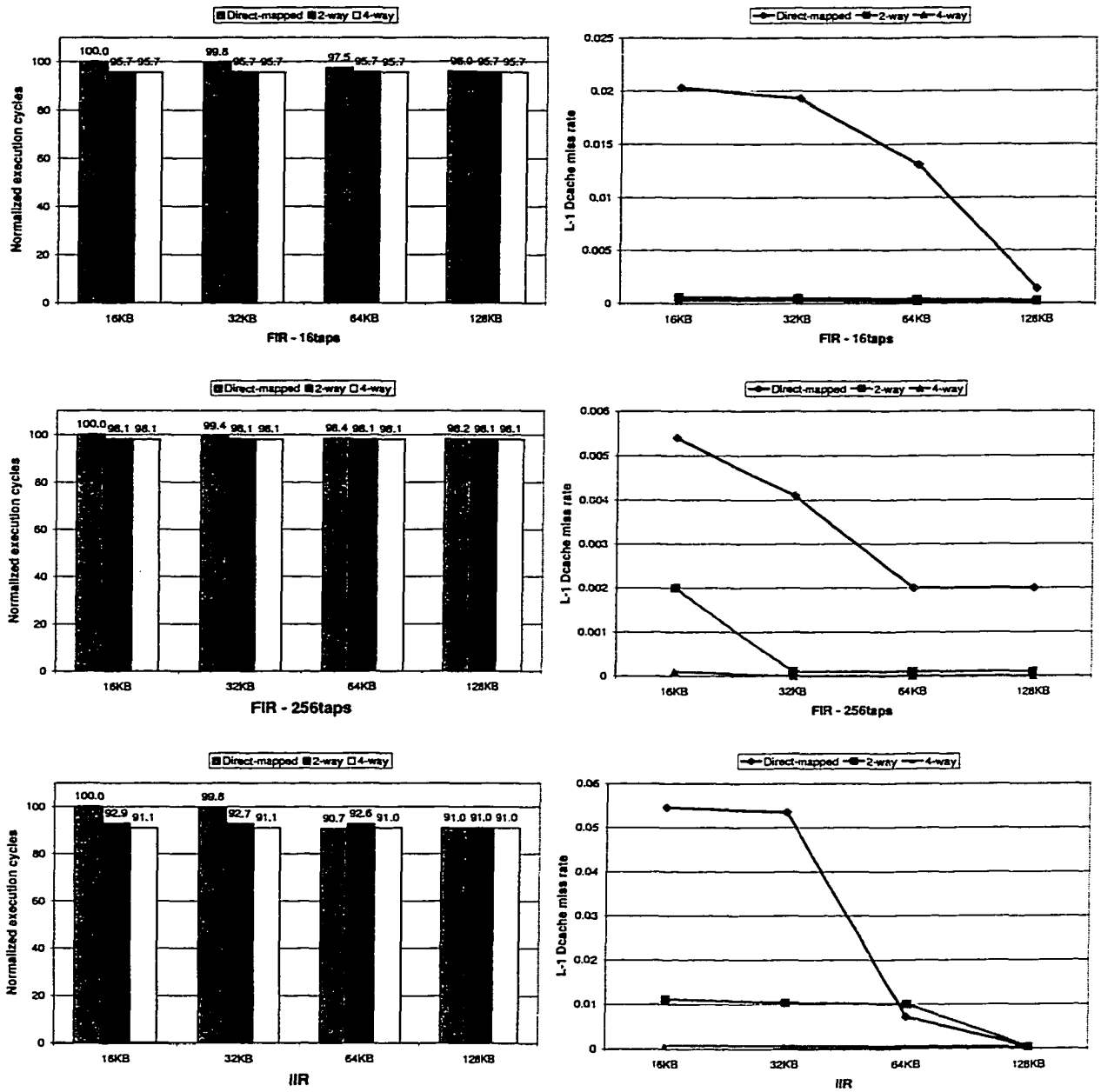


Figure 7.4 Normalized execution cycles with various cache organizations

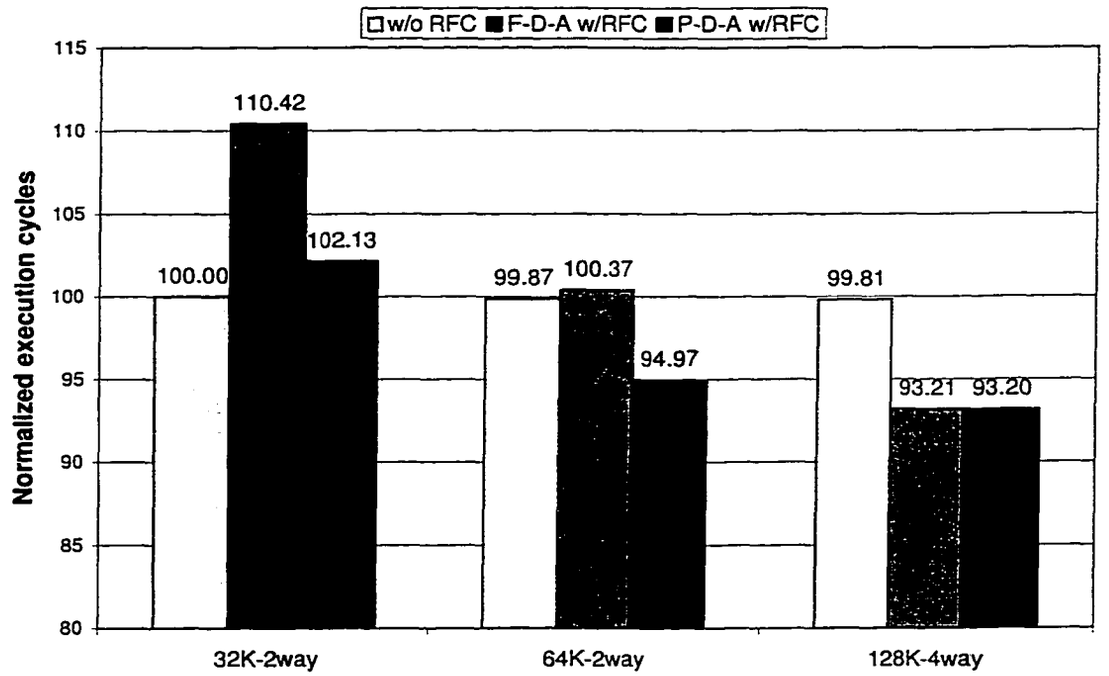


Figure 7.5 Normalized execution cycles for mpeg2enc with Full Dynamic Associativity (FDA) and Partial Dynamic Associativity (PDA)

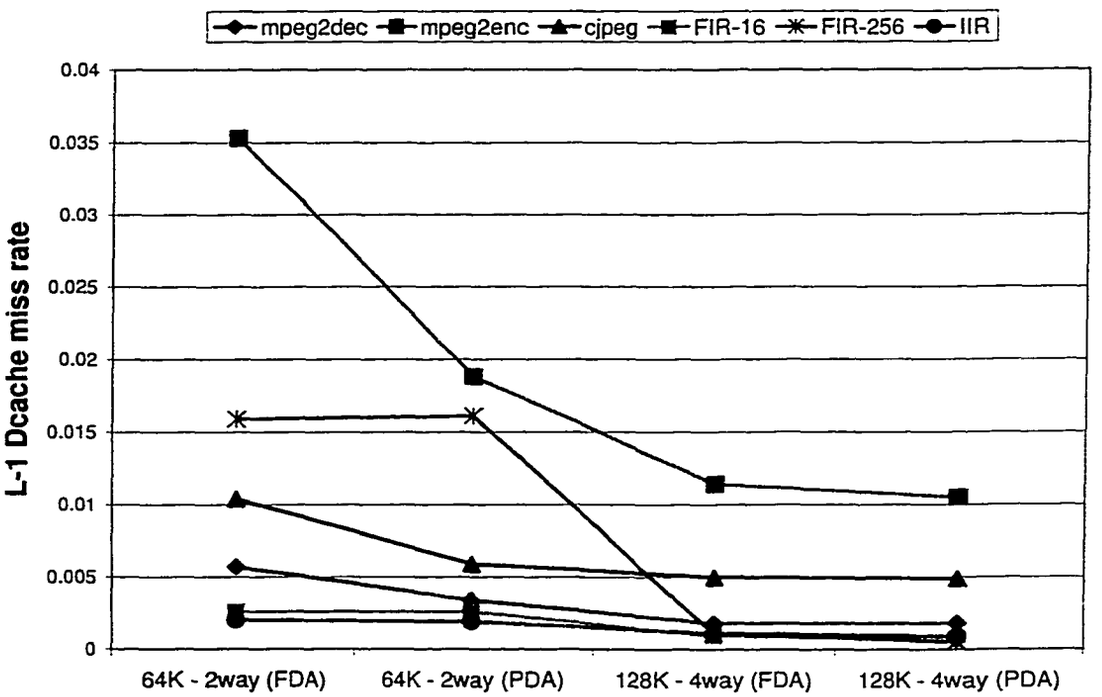


Figure 7.6 Miss rate with Full Dynamic Associativity (FDA) and Partial Dynamic Associativity (PDA)

cache memory. This minimizes the impact of the reconfiguration with respect to the caching capacity. This should not be a problem as most microprocessors use a 4-way set associative cache.

The most important factors for speed-up are the reduced number of instructions and the acceleration of computation with a specialized unit. Using the rfc instructions, we can reduce the number of instructions significantly because most of the arithmetic instructions are replaced with the operations in the RFC. Additionally, there is a reduction in the number of stack memory accesses from the original source code. Since the computing unit in RFC is specialized and customized for the desired computations, it can potentially feed the RFC with new data every cycle. Moreover, the specialized unit replaces many operations from the original source code. The data is loaded once in the pipeline and then processed through all the required stages using a specialized computing unit configured from the RFC.

Configuration time: The time for configuration is determined with the number of sets and cache line size in a cache module regardless of the associativity. If we configure partially at run-time, the number of partial configuration steps is an additional factor. The number of cycles for RFC configuration in the benchmarks is shown in Table 7.3. The configuration cycles in the various organizations are shown in Table 7.4. The results indicate that the number of sets is the dominant factor to determine the configuration time. This is because the number of sets corresponds to the number of memory accesses (misses) and a larger cache line writes contents of many LUTs simultaneously.

Table 7.3 Number of cycles for the configuration in benchmarks

	32K-2way/64K-4way (256 sets)	64K-2way/128K-4way (512 sets)
mpeg2dec	1654	3112
mpeg2enc	1534	2998
cjpeg	1596	3060
FIR - 16 taps - 256 taps	 2236 12256	 4264 25684
IIR	1657	6125

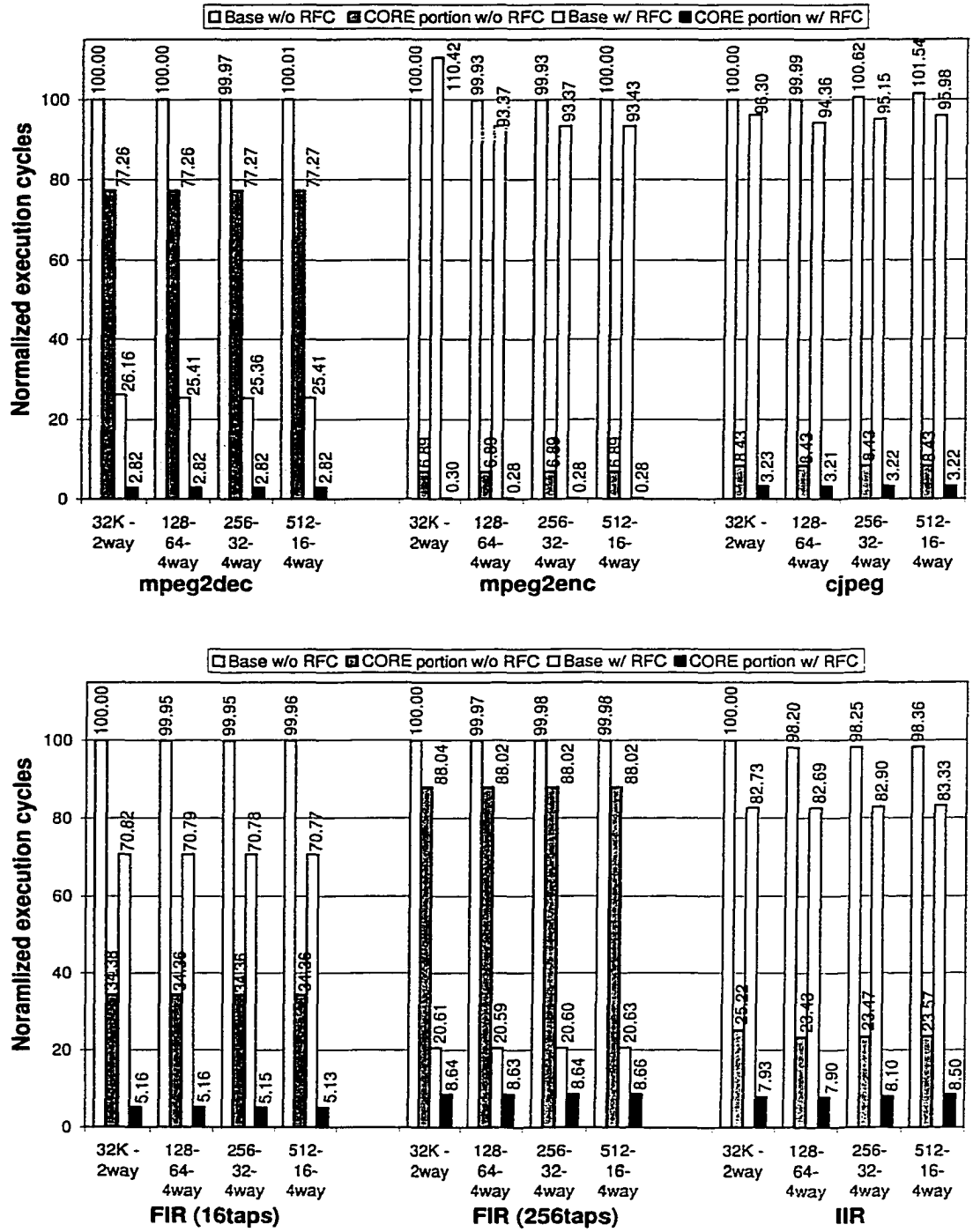


Figure 7.7 Normalized execution cycles w/o RFC and w/ RFC built in 4-way associative cache (FDA)

Table 7.4 Configuration cycles with different cache organizations (32K - 4way)

	128 sets and 64B	256 sets with 32B	512sets with 16B
mpeg2dec	910	1592	2986
mpeg2enc	798	1478	2872
cjpeg	860	1540	2934
FIR			
- 16 taps	1264	2177	4140
- 256 taps	6844	12137	25620
IIR	1725	3085	5885

Cache misses: When one of the data cache modules is configured as a computing unit, the capacity of cache memory is reduced to 50% and 75% for 2-way and 4-way set associativity in the full dynamic associativity, respectively. To see the effect of reduced memory capacity, we profile the number of data accesses and misses in the level-1 data cache memory without and with RFCs in Table 7.5 for the selected benchmarks. The miss rates for the benchmarks are also shown in Figure 7.8. In a low associativity (2-way), the number of misses after configuring an RFC as a computing unit is higher than the base processor without RFCs. However, the number of misses in 128K 4-way set associative cache does not vary significantly with the dynamic associativity. In FIR using the RFC, the total number of data accesses is cut down due to the reduced number of stack accesses. Since the number of accesses for intermediate data in the RFC process is added as many times as the number of partial reconfigurations, the total number of misses increases. Note that the number of misses with RFC in mpeg2dec and IIR is smaller than without RFC. Most of the instructions executed in mpeg2dec are replaced with the rfc instructions (about 77% for IDCT). However, the miss rate is higher than without RFC. The data forwarding between OBUF and IBUF for IIR described in Section 5.2.4.6 reduces the number of misses significantly. This forwarding may not be done in the original benchmark with the base processor (without RFC) due to the nature of the program. As shown in Figure 7.6, the number of misses reduces further with the partial dynamic associativity.

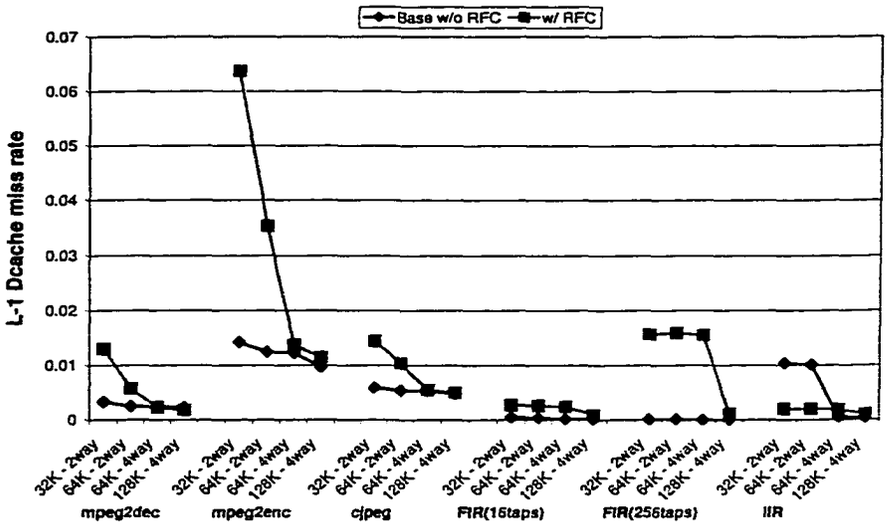


Figure 7.8 Miss rate for level-1 data cache memory

Table 7.5 Number of data accesses and misses to level-1 data cache

		mpeg2dec		mpeg2enc		cjpeg	
		access	miss	access	miss	access	miss
32K 2-way	w/o RFC	557217369	1841048	395764840	5621892	88192493	522514
	w/ RFC	243479064	3166490	388017519	24724727	88187936	1272728
64K 2-way	w/o RFC	557244797	1407812	396131144	4898529	88181257	473107
	w/ RFC	244079977	1399239	377216829	13327688	88201102	917253
64K 4-way	w/o RFC	557242614	1316662	396774657	4856816	88185264	466513
	w/ RFC	244214721	569710	376845411	5157770	88192895	486697
128K 4way	w/o RFC	557343109	1289796	396638308	3842490	88178858	423220
	w/ RFC	244215876	444390	376690409	4305989	88175818	438739
		fir16		fir256		iir	
		access	miss	access	miss	access	miss
32K 2-way	w/o RFC	5633761	2752	37092533	4247	4537186	47106
	w/ RFC	3666798	9843	6618498	104043	3685969	7416
64K 2-way	w/o RFC	5634039	2109	37092077	2851	4536698	45774
	w/ RFC	3667633	9420	6621253	105480	3686000	7286
64K 4-way	w/o RFC	5634260	1372	37091554	1378	4538886	2061
	w/ RFC	3667575	8724	6619275	102924	3685741	6828
128K 4way	w/o RFC	5634260	1353	37091554	1360	4538887	1866
	w/ RFC	3668947	3129	6622567	7449	3686433	3939

RFC organization: According to the simulation results with various parameters, we determine the overall cache organization with RFC, such as associativity, number of sets, and line size. A high-associativity minimizes the number of conflict misses caused by the reduction in the associativity. A smaller number of sets and larger line size (with the minimal size of RFC) reduces the configuration time because of a smaller number of memory accesses writing many multi-bit LUT configurations in parallel. This is preferred if the reconfiguration of RFCs occurs frequently, for instance, in an FIR filter with a large number of taps. However, a large sized cache memory with a large number of sets reduces number of misses in general. The large number of sets could reduce the conflict misses by distributing the blocks to be mapped to a set into other sets, especially when an RFC is used as a computing unit. In the case of a large number of sets in cache memories, the partial dynamic associativity, which changes the associativity only in a small portion of cache, can reduce the impact of the reconfiguration further. The computing time using RFCs does not vary with the different cache organizations except with a small cache line size (16 bytes).

Memory bandwidth: A number of arithmetic instructions for the core computations are replaced with the memory operations. In other words, the whole core computation is transferred from software (instructions) to the specialized computing hardware with only the interface instructions (rfc load/store). This may require a high off-chip memory bandwidth (to level-2 cache) for a fast execution with the RFCs. We profile the memory bandwidth between the level-2 unified cache memory and the off-chip memory as the required bandwidth (for instructions and data) in bytes per cycle. The configuration data carries additional traffic in using the RFCs. Figure 7.9 shows the memory bandwidth normalized to mpeg2dec for the benchmarks. The memory bandwidth with the RFCs does not increase significantly. This is because of the reduced number of instructions using the rfc instructions. A number of memory accesses to fetch the instructions are removed. In FIR/IIR, the required memory bandwidth with RFC is significantly higher than without RFC compared to DCT/IDCT. The partial configuration data causes the higher memory bandwidth, especially, in FIR with 256 taps.

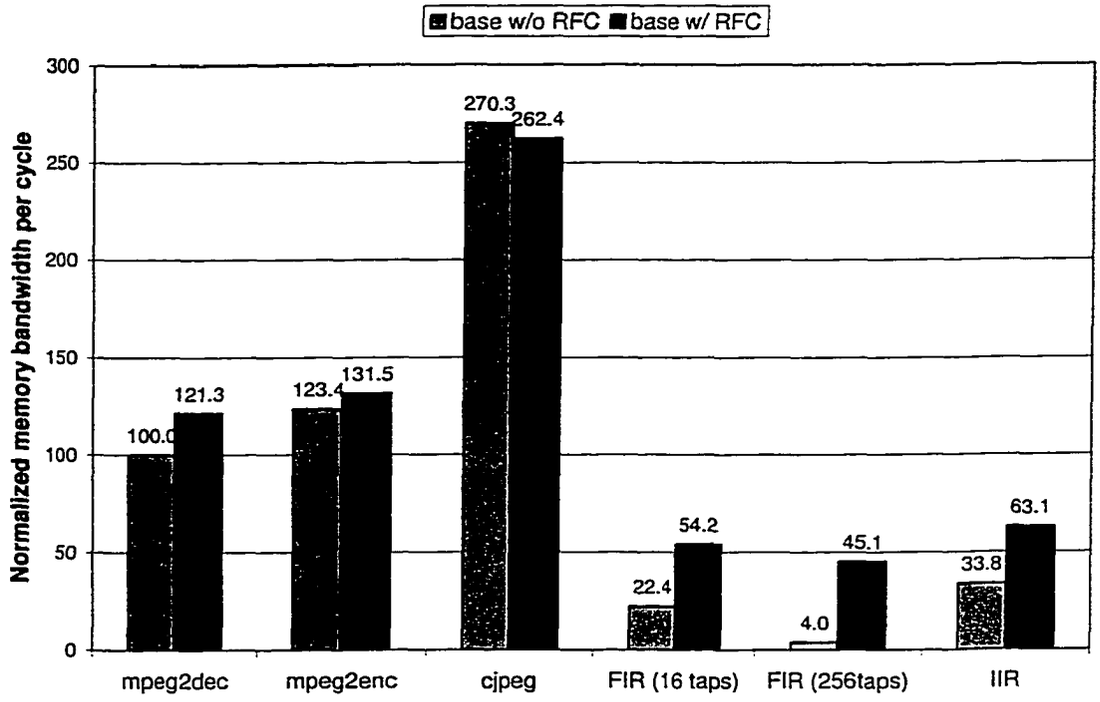


Figure 7.9 Memory bandwidth required per CPU cycle with a 32K - 4way set associative cache

Adaptability: The simulation also shows that the base processor with a smaller size cache memory and RFC performs better compared to a larger size cache without RFC. This may suggest a microarchitecture with a smaller cache size matching the performance of a traditional microarchitecture with a larger cache size. This fact is true in media applications due to the streaming nature of data and the lack of temporal locality as mentioned in Section 1. However, the base processor considered in this thesis is a general-purpose computing microprocessor. General applications with more random data accesses and higher temporal locality other than media applications may need larger on-chip cache memory for a better performance. (e.g. HP PA-8500 [18] with 1MB D-cache for fewer cache misses). This implies that a general purpose computing processor would need a larger cache memory for a higher performance of applications. These observations motivate the need for an adaptive amount of resources - between memory and computing unit on demand. Thus, in ABC with RFCs, the resources are more fully utilized and some of the applications are accelerated significantly.

To see the effect of the reconfigurability (adaptability) in this thesis, we also simulated

SPEC-95 with 3-way and 4-way set associative caches with the parameters shown in Table 7.2. We used the *sim-cache* simulator in Simplescalar [71]. *Sim-cache* profiles only the cache functionalities without any other processor parameters. The simulation parameters for memory hierarchy are 32KB/128KB level-1 data cache and 1MB L-2 cache. The miss rates of level-1 data cache with both organizations in SPEC-95 are shown in Figure 7.10. The 4-way set associativity reduces the number of cache misses in most cases, which can improve the overall performance for the applications. We also simulated the SPEC-95 with 3-way and 4-way associative caches and the base parameters shown in Section 7.2.2. Some of the applications were improved ranging from 0.3% to 1.6% with the 4-way set associative cache. If we had a separate specialized computing unit (like dedicated reconfigurable logic) and 3-way set associative cache, we would not get the improvements achievable by a 4-way set associative cache with RFC for SPEC-95 benchmarks. This provides us with a strong motivation to implement RFC in general purpose processors.

7.4 Microprocessor with RFC vs. without RFC

A conventional general-purpose processor without the reconfigurable functional cache can potentially store more data in its cache memory. However, a large cache does not speed up FIR/IIR and DCT/IDCT as much as specialized computing resources can. A small area overhead from a reconfigurable functional cache in a processor to provide these specialized computing resources accelerates these functions. Moreover, the reconfigurability does not affect other functions' execution time negatively since it does not penalize the cache access (may be a bit slower or even faster as mentioned in Section 4.4). For example, a program with half of its execution time due to FIR or DCT will speed up the FIR/DCT-half with RFC while rest of the program retains the same execution time.

Since the reconfigurability of cache is an orthogonal design axis/issue with respect to a conventional cache structure, the cache strategy does not affect the normal cache operation. RFC does not require the existing cache structure to be modified. It does add additional units such as partitioned decoder and interconnects without destroying conventional cache

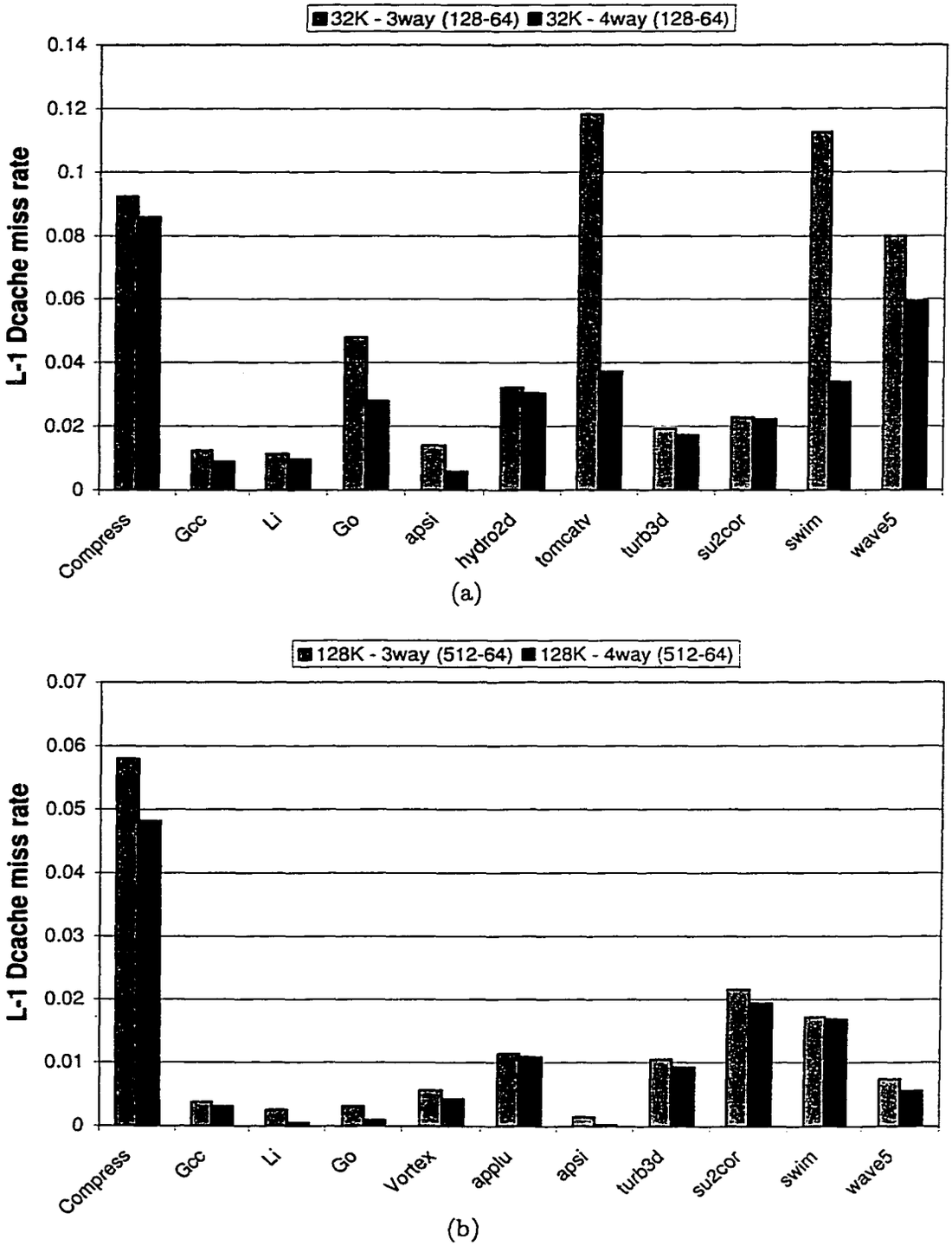


Figure 7.10 Miss rate with 3-way and 4-way set associative cache for SPEC-95 (a)32KB ; (b)128KB

architecture and strategies. Simply, a cache converts into a special function unit as a custom-computing machine. The only negative effect is a potentially higher miss rate when part of the cache is reconfigured for computing. However, less memory capacity does not impact the performance of FIR and DCT (commonly used in multimedia applications) because a larger size of cache memory does not scale linearly in a higher performance for media applications [76, 19, 20, 21]. Only the processor area will increase due to additional RFC logic (which is smaller than other customized chips) with a higher performance for FIR and DCT/IDCT.

In summary, we are targeting a visible high performance (10's of speed up) of computations in a microprocessor with a certain family of applications, such as multimedia and DSP applications instead of improving general-purpose computing. The RFC could be one solution with low complexity in microarchitecture and design. Other general-purpose computing applications do not require high performance and need not be accelerated as much as media applications since they do not contain high computing bandwidth tasks. The result of comparison depends on the frequency of application use. However, media applications are used frequently requiring high performance. This trend will continue until the advent of new application commodities. Therefore, we are developing one possible solution that can accelerate the most common media applications visibly on general-purpose microprocessor with low area/time overhead instead of special purpose processor or dedicated hardware.

CHAPTER 8. CONCLUSION

We have shown an Adaptive Balanced Computing (ABC) microarchitecture using RFCs as a dynamic allocated resource. We have also shown a prototype of Reconfigurable Functional Cache (RFC), which can perform both as a function unit and a cache, providing dynamic memory/computing resources. The evaluation of ABC with minimal modification in microarchitecture and cache memory is presented. The RFC can serve as a special computing unit without any significant modification and overhead in area/time domains in cache architecture and a microarchitecture. The RFC can be integrated into a microprocessor by adding a small number of well-matched instructions to the existing Instruction Set Architecture (ISA) and making minor changes to a conventional compiler. The proposed microarchitecture utilizing RFC could also work in parallel with any ASIC/FPGA-like coprocessor in on-chip microprocessor. The resource reconfiguration produces a higher performance by providing resources specialized to the computing requirements as compared to the fixed configuration of level-1 cache memories. The reconfiguration impacts the overall performance minimally. The result of simulations for mpeg2 encoder/decoder in Distributed Arithmetic and FIR/IIR in MAC using the RFC indicates that a certain class of applications (such as multimedia and DSP that are compute-intensive and well-structured) can be accelerated highly. The area penalty for this reconfiguration is about 50-60% of the memory cell cache array area with faster cache access time, and 10-20% of the base cache array area with 1-2% increase in the cache access time. In this thesis, a function-level optimization using the RFC is introduced as well. Since the reconfigurable unit is based on LUTs, other applications, which have the same behavior (but different characteristics - such as different coefficients and sequence of computation), may be mapped/executed into the RFC without any significant addition. We have also shown a possi-

bility of dynamic distributed microarchitecture on demand of compute-bound applications by unloading heavy computations from the processor core to RFC with a small architectural and design modification. We are currently working towards an exclusive microarchitecture state for RFCs to produce more parallelism using a DMA type of cache management unit, which loads and stores data to/from the RFC independent on the core processor. The future work includes programmable interconnection in RFC to support various computations. This would further promote the efficient use of reconfigurable functional caches on a general purpose processor. If independent multiple tasks are executed and multiple RFCs are used simultaneously, more function-level parallelism can be achieved.

BIBLIOGRAPHY

[1] John Vincent Atanasoff and Clifford Berry, "Atanasoff-Berry Computer (ABC)", 1937-1942, available on http://www.scl.ameslab.gov/ABC/ABC.html.

[2] Andre' DeHon, "Reconfigurable Architectures for General-Purpose Computing", Ph D Thesis, Lab. for Computer Science, MIT, Cambridge, MA, 1996.

[3] H.T. Kung, "Memory Requirements for Balanced Computer Architectures", Proceedings of the 13th Annual Symposium on Computer Architecture, pp. 49-54, ACM, 1986.

[4] Chung-Ho Chen and Arun K. Somani, "Architecture Technique Trade-Offs Using Mean Memory Delay Time", in IEEE transactions on Computers, Vol. 45, No. 10, pp. 1089-1100, October 1996.

[5] J. Hennessy, "The Future of System Research", IEEE Computer, 32(8), pp. 27-33, Aug. 1999.

[6] John R. Hauser and John Wawrzynek, "Garp: A MIPS Processor with a Reconfigurable Coprocessor",in Proceedings of the IEEE Symposium on Field-Programmable Custom Computing Machines, Apr. 1997.

[7] André DeHon, "DPGA-coupled microprocessor: Commodity ICs for the early 21st century", In D. a. Buell and K. L. Pocek, editors, Proceedings of IEEE workshop on FPGAs for Custom Computing Machines, pp. 31-39, Napa, CA, Apr. 1994.

[8] R. Razdan and M. D. Smith, "A high-performance microarchitecture with hardware-programmable functional units", in Proc. of the 27th Annual Intl. Symp. on microarchitecture, pp. 172-80. IEEE/ACM, Nov. 1994.

[9] A. Tyagi, " Reconfigurable memory queues/computing units architecture", in Proc. of the Reconfigurable Architecture workshop at 11th International Parallel Processing Symposium, Apr. 1997.

[10] S. Hauck, T. W. Fry, M. M. Hosler, J. P. Kao, "The Chimaera Reconfigurable Functional Unit", in Proc. of IEEE Symposium on FPGAs for Custom Computing Machines(FCCM'97), pp. 87-96, 1997.

[11] Ralph D. Wittig and Paul Chow, "OneChip: An FPGA Processor With Reconfigurable Logic", in IEEE Symposium on FPGAs For Custom Computing Machines (FCCM), 1996

[12] BRASS Research Group, Berkeley Reconfigurable Architectures, Systems, & Software, "Integrating Processors and Reconfigurable Logic", 2001
available on http://brass.cs.berkeley.edu/reproc.html

[13] "The Hokie Instant RISC Microprocessor", Department of Electrical Engineering, Virginia Tech, 1995-1996
available on http://www.ee.vt.edu/ harper/ee6504/rapid.html.

[14] Michael J. Flynn, Patrick Hung, and Kevin W. Rudd, "Deep-Submicron Microprocessor Design Issues", in IEEE Micro Vol. 19, No. 4, July/August 1999.

[15] Y.Patt, S. Patel, M. Evers, D. Friendly, and J. Stark, "One billion Transistors, One Uniprocessor, One Chip", IEEE Computer, pp. 51-57, Sep. 1997.

[16] The National Technology Roadmap for Semiconductors, technical report, Semiconductor Industry Association., San Jose, CA., 1994 and 1997 (updates).

[17] John Hennessy, "The Future of Systems Research", IEEE Computer, pp. 27-33, Aug. 1999.

[18] Gregg Lesartre and Doug Hunt, "PA-8500: The Continuing Evolution of the PA-8000 Family", Hewlett-Packard Company, 2000,
available on http://www.cpus.hp.com/.

[19] P. Ranganathan, S. Adve, and N. P. Jouppi, "Performance of Image and Video Processing with General-Purpose Processors and Media ISA Extensions", Proceedings of the 26th International Symposium on Computer Architecture, pp. 124-135, May 1999.

[20] P. Soderquist and M. Leeser, "Memory Traffic and Data Cache Behavior of an MPEG-2 Software Decoder", 1997 International Conference on Computer Design, Oct. 1997.

[21] J. Eijndhoven et al., "TriMedia CPU64 Architecture" 1999 International Conference on Computer Design, Oct. 1999.

[22] Mark Oskin, Fredric T. Chong, and Timothy Sherwood, "Active Pages: A Computation Model for Intelligent Memory", in Proc. of the 25th International Symposium on Computer Architecture (ISCA'98).

[23] Yi Kang, Wei Huang, Seung-Moon Yoo, Diana Keen, Zhenzhou Ge, Vinh Lam, Pratap Pattanaik and Josep Torrelas. "FlexRAM : Toward an Advanced Intelligent Memory System", Proc. of 1999 International Conference on Computer Design, pp. 47-55, Oct. 1999.

[24] Kimberley Keeton, Remzi Arpaci-Dusseau, and David A. Patterson, "IRAM and Smart-SIMM: Overcoming the I/O Bus Bottleneck", in Proc. of International Symposium on Computer Architecture, June 1997.

[25] David Patterson, Thomas Anderson, Neal Cardwell, Richard Fromm, Kimberley Keeton, Chritoforos Kozyrakis, Randi Thomas, and Katherine Yelick, "A Case for Intelligent RAM", in IEEE Micro. pp 33-44, March/April 1997.

[26] Doug Matzke, "Will Physical Scalability Sabotage Performance Gains?", IEEE Computer, Vol. 30, No. 9, Sep. 1997

[27] Huesung Kim, Arun K. Somani, and Akhilesh Tyagi, "On Reconfiguring Cache for Computing", in the Proceedings of FCCM '99, pp. 296-297, April 1999.

[28] Huesung Kim, Arun K. Somani, and Akhilesh Tyagi, "A Reconfigurable Multi-function Computing Cache Architecture", Intl. Symp. on FPGAs, pp. 85-94, February. 2000.

[29] Huesung Kim, Arun K. Somani, and Akhilesh Tyagi, Extended version of "A Reconfigurable Multi-function Computing Cache Architecture", in IEEE transactions on Very Large Scale Integration (VLSI), Vol. 9, No. 4, pp. 509-523, August 2001.

[30] Srihari Cadambi, Jeffrey Weener, Seth Copen Goldstein, Herman Schmit, Donald E. Thomas, "Managing Pipeline-Reconfigurable FPGAs", in Proceedings ACM/SIGDA Sixth International Symposium on Field Programmable Gate Arrays, February 1998.

[31] Xilinx Inc., "Virtex 2.5V Field Programmable Gate Arrays Datasheet", 2001, available on http://www.xilinx.com/apps/virtexapp.htm.

[32] Xilinx Inc., San Jose, CA 95124-3400, available on http://www.xilinx.com, 2001.

[33] J. Rose, R. Francis, D. Lewis, and P. Chow, "Architecture of programmable gate arrays: The effect of logic block functionality on area efficiency", IEEE Journal of Solid-State Circuits, v. 25, pp. 1217-1225, Oct. 1990.

[34] Andre' DeHon, "Balancing Interconnect and Computation in a Reconfigurable Computing Array (or, why you don't really want 100% LUT utilization)", in Proc. of FPGA'99, Feb. 1999.

[35] Xilinx Inc., "Xilinx Application Notes", 2001, available on http://www.xilinx.com/apps/appsweb.htm.

[36] Altera Inc., "On-Line Literature: Application Notes", 1995 - 2001, available on http://www.altera.com/literature/lit-an.html.

[37] Xilinx Inc., "XC4000 Series Field Programmable Gate Arrays", 2001 available on http://www.xilinx.com/apps/4000.htm.

[38] André DeHon, "A First Generation DPGA Implementation", in Third Canadian Workshop of Field-Programmable Devices, Montreal, Canada, May 29-June 1, 1995.

[39] Michael Bolotski, André DeHon, and Thomas F. Knight, Jr, "Unifying FPGAs and SIMD Arrays", in 2nd International ACM/SIGDA Workshop on FPGAs, Berkeley, CA, Feb. 13-15, 1994.

[40] Bernardo Kastrup, Arjan Bink, and Jan Hoogerbrugge, "ConCISe: A Compiler-Driven CPLD-Based Instruction Set Accelerator", in Proc. of IEEE Symposium on FPGAs for Custom Computing Machines (FCCM'99), pp. 92-101, 1999.

[41] P. Kogge, "The EXECUBE Approach to Massively Parallel Processing", in Proc. of the 1994 International Conference on Parallel Processing, Aug. 1994.

[42] P. Ranganathan, S. Adve, and N. P. Jouppi, "Reconfigurable Caches and their Applications to Media Processing", Proceedings of the 27th International Symposium on Computer Architecture, June 2000.

[43] Craig Wittenbrink and Aurn K. Somani, "Cache Tiling for High-Performance Morphological Image Processing", Machine Vision and Applications, pp. 12-22, Nov. 1993.

[44] Alex Peleg and Uri Weiser, "MMX technology extension to the Intel architecture", in IEEE Micro Vol.16, No. 4, Aug. 1996.

[45] Srinivas K. Raman, Vladimir Pentkovski, and Jagannath Keshava, "Implementing Streaming SIMD Extensions on the Pentium III processor", in IEEE Micro Vol.20, No. 4, July/August 2000.

[46] Stuart Oberman, Greg Favor, and Fred Weber, "AMD 3DNow! Technology: Architecture and Implementations", in IEEE Micro Vol.19, No. 2, March/April 1999.

[47] R. Lee, "Subword parallelism with MAX-2", in IEEE Micro , Vol.16, No. 4, Aug. 1996.

[48] M. Tremblay, J. M. O'Connor, V. Narayanan, and Liang He, "VIS speeds new media processing", in IEEE Micro , Vol.16, No. 4 , Aug. 1996

[49] Keith Diefendorff, Pradeep K. Dubey, Ron Hochsprung, and Hunter Scales, "AltiVec Extension to PowerPC Accelerates Media Processing", in IEEE Micro Vol. 20, No. 2, March/April 2000.

[50] Steven J.E Wilton and Norman P. Jouppi, "CACTI: an enhanced cache access and cycle time model", IEEE Journal of Solid State Circuits. v. 31 May '96 pp. 677-688.

[51] "The UltraSPARC - IIi Microprocessor", Sun microsystems, 1997
available on http://www.sun.com/microelectronics/UltraSPARC-IIi/

[52] James E. Smith and Gurindar S. Sohi, "The Microarchitecture of Superscalar Processors", in Proceedings of the IEEE, December 1995

[53] Mike Johnson, "Superscalar Microprocessor Design", Prentice Hall, 1991.

[54] Mathew Wojko and Hossam ElGindy, "Self Configuring Biary multiplier for LUT addressable FPGAs", in the 5th Australasian Conference on Parallel and Real-Time Systems 1998.

[55] J. Lachman and J. M. Hill, "A 500 MHz 1.5 MB cache with on-chip CPU", Digest of Technical Papers, IEEE International Solid-State Circuits Conference (ISSCC), 1999

[56] Bharadwaj S. Amrutur, "Design and Analysis of Fast Low Power SRAMs", Ph. D. Dissertation, Dept. of Electrical Engineering, Stanford University, Aug. 1999.

[57] Wafer Electrical Test Data and SPICE Model Parameters, 2000
available on http://www.mosis.org/Technical/Testdata

[58] Deepali Deshpande, Arun K. Somani, and Akhilesh Tyagi, "Configuration Scheduling Schemes for Striped FPGAs", in Proc. of FPGA'99, Feb. 1999, pp. 206-214.

[59] W. H. Chen, C. H. Smith, and S. V. Fralick. "A fast computational algorithm for the discrete cosine transform", IEEE Transaction on Communications, Vol.COM-25, no.9, pp. 1004-1009, Sep. 1977.

[60] M. Maruyama, H. Uwabu, I. Iwasaki, H. Fujuwara, T. Sakaguchi, M.T. Sun, and M.L. Liou, "VLSI Architecture and Implementation of a Multi-Function, Forward/Inverse Discrete Cosine Transform Processor", SPIE Vol. 1360, Visual Communications and Image Processing '90, pp. 410-417.

[61] P. Duhamel, C. GUILLEMOT, and J.C. Carlach, "A DCT Chip based on a new structured and computationally efficient DCT algorithm", Proc. of IEEE ISCAS '90, pp. 77-80, New Orleans, May 1990.

[62] T. Yoshino, R. Jain, P. Yang, H. Davis, W. Gass, and A. H. Shah, "A 100-MHz 64-tap FIR Digital Filter in 0.8-μm BiCMOS Gate Array", IEEE Journal of Solid-State Circuits, Vol. 25, No. 6, Dec. 1990, pp. 1494-1501.

[63] M. Hatamian and S. Rao, "A 100 MHz 40-tap programmable FIR filter chip" in Proc. IEEE Int. Symp. Circuits Syst.,, 1990, pp. 3053-3056.

[64] Ishikawa et al., "Automatic layout synthesis for FIR filters using a silicon compiler", in Proc. IEEE Int. Symp. Circuits Syst., 1990, pp. 2588-2591

[65] S. Hsia, B. Liu, J. Yang, and B. Bai,"VLSI Implementation of Parallel Coefficient-by-Coefficient Two-Dimensional IDCT Processor", IEEE Transaction on Circuits and Systems for Video Technology, Vol. 5, No. 5, Oct. 1995.

[66] T. Masaki, Y. Morimoto, T. Onoye, and I. Shirakawa, "VLSI Implementation of Inverse Discrete Transformer and Motion Compensator for MPEG2 HDTV Video Decoding", IEEE Transaction on Circuits and Systems for Video Technology, Vol. 5, No. 5, Oct. 1995.

[67] A. Madisetti, and A. N. Willson, "A 100 MHz 2-D 8×8 DCT/IDCT Processor for HDTV Applications", IEEE Transaction on Circuits and Systems for Video Technology, Vol. 5, No. 2, Apr. 1995.

[68] S. Uramoto et al., "A 100 MHz 2-D discrete cosine transform core processor", IEEE Journal of Solid-State Circuits, Vol. 27, pp. 492-499, Apr. 1992.

[69] M. Izumikawai et al., "A 0.25-μm CMOS 0.9-V 100-MHz DSP Core", IEEE Journal of Solid-State Circuits, Vol. 32, No. 1, Jan. 1997, pp. 52-61.

[70] Fang Lu, and Henry Samueli, "A 200-MHz CMOS Pipelined Multiplier-Accumulator Using a Quasi-Domino Dynamic Full-Adder Cell Design", IEEE Journal of Solid-State Circuits, Vol. 28, No. 2, pp. 123-132, Deb. 1993.

[71] Doug Burger and Todd M. Austin, "The SimpleScalar Toot Set, Version 2.0', Computer Sciences Department Technical Report #1342, University of Wisconsin-Madison, June, 1997.

[72] Xilinx Inc., "The Role of Distributed Arithmetic in FPGA-based Signal Processing", 2000 available on http://www.xilinx.com/apps/arith.htm

[73] Chunho Lee, Miodrag Potkonjak, and William H. Mangione-Smith, "MediaBench: A Tool for Evaluating and Synthesizing Multimedia and Communications Systems" in Proc. of the 30th Annual Intl. Symp. on Microarchitecture, pp.330-335, Dec. 1997.

[74] Nathan T. Slingerland and Alan Jay Smith, "Design and Characterization of the Berkeley Multimedia Workload", University of California at Berkeley Tech Report CSD-00-1122, Dec. 2000.

[75] Texas Instruments, "TMS320C6000 benchmarks", 2000
available on http://www.ti.com/sc/docs/products/dsp/c6000/62bench.htm

[76] Abhishek Singhal, "Reconfigurable Cache Module Architecture", M.S. Thesis, Department of Computer Science, Iowa State University, Ames, IA, May 2000.

ACKNOWLEDGEMENTS

First and foremost I would like to express my gratitude to my supervisor, Dr. Arun K. Somani, whose expertise, understanding, and patience added considerably to this research. I appreciate his gentle encouragement, constructive criticism, and for training me to be a successful researcher. I really appreciate his attention and concern regarding the research and my personal problem, especially, his sincere guidance and advice when I had a hard time because of my physical difficulties. His confidence in my ability directed me to succeed in this work. I also appreciate the long hours that he spent with me working on papers. His efforts and patience with my writing improved my technical writing skills significantly. It was a pleasure to work with him from both the intellectual and personal points of view.

I also wish to express my sincere gratitude to my co-major professor, Dr. Akhilesh Tyagi, for his assistance and advice on the research. I have deep regards for the advice and direction that he gave me. His encouragement and respect to my ideas motivated me to work hard and further. His valuable comments on my work and papers helped me to be a good researcher. I also appreciate his sincere discussion on the research and papers. His insights and encouragement have inspired me to complete this project.

I really appreciate the opportunities Dr. Somani and Dr. Tyagi have given me, particularly the opportunity to research Adaptive Balanced Computing, computer architecture, and VLSI design. I would also like to thank those who have helped me edit this thesis for English usage errors, especially, Jon Froehlich, Rama Sangireddy, and Mona Dalal.