Fast Thread Communication and Synchronization Mechanisms for a Scalable Single Chip Multiprocessor

by

Stephen William Keckler

B.S. Electrical Engineering, Stanford University, 1990
S.M. Computer Science, Massachusetts Institute of Technology, 1992

Submitted to the Department of Electrical Engineering and Computer Science
In Partial Fulfillment of the Requirements for the Degree of

Doctor of Philosophy
in Electrical Engineering and Computer Science

at the

Massachusetts Institute of Technology
June 1998

Signature of Author ______________________________

Department of Electrical Engineering and Computer Science
May 18, 1998

Certified by ______________________________

Dr. William J. Dally
Professor of Electrical Engineering and Computer Science
Thesis Supervisor

Accepted by ______________________________

Dr. Arthur C. Smith
Chairman, Committee on Graduate Students
Department of Electrical Engineering and Computer Science

Fast Thread Communication and Synchronization Mechanisms for a Scalable Single Chip Multiprocessor

by

Stephen William Keckler

Submitted to the
Department of Electrical Engineering and Computer Science
on May 18, 1998, in partial fulfillment of
the requirements for the Degree of Doctor of Philosophy in
Electrical Engineering and Computer Science

Abstract

Much of the improvement in computer performance over the last twenty years has come from faster transistors and architectural advances that increase parallelism. Smaller feature sizes have decreased the transistor switching time but at the same time increased the resistance of interconnect wires, resulting in slower signal transmission in on-chip wiring. Since future chips will have more silicon area and include more execution units, a much larger demand for parallelism is emerging. However, the increased significance of wire delay will require monolithic components, such as processors and caches, to be small and that the communication wires connecting them be short.

Computer systems typically exploit concurrency using either instruction level parallelism (ILP) or coarse-grain parallel threads running on a multiprocessor. This thesis proposes mechanisms for exploiting on-chip parallelism at a fine grain to bridge the gap between ILP and coarse-grain multiprocessing. Fast interprocessor communication and synchronization enables the use of tasks with run lengths as small as 10 cycles. At the same time, these interaction mechanisms are less susceptible than conventional microprocessor designs to longer wire delays imminent in future silicon process technologies. As fine-grain parallelism is orthogonal to ILP and coarse-grain threads, it complements both methods and provides an opportunity for greater speedup.

This thesis presents the architecture and implementation of the MIT Multi-ALU Processor (MAP), a 5 million transistor custom VLSI microprocessor chip. The MAP architecture incorporates 9 function units, split into 3 independent processors. The processors communicate via interprocessor register writes and synchronize using a hardware barrier instruction. These integrated mechanisms allow threads to communicate 10 times faster and synchronize 60 times faster than using a shared on-chip cache. The fast interprocessor interaction enables the MAP to exploit both instruction-level parallelism and fine-grain thread level parallelism. On a suite of applications, speedups of 1.2–2.4 are achieved using fine-grain threads on a 3-processor MAP chip.

Thesis Supervisor: Dr. William J. Dally
Title: Professor of Electrical Engineering and Computer Science

Acknowledgments

> It is good to have an end to journey toward, but it is the journey that matters in the end.
>
> – *Ursula K. LeGuin*

And quite a journey it has been. I am deeply indebted to my guide, mentor, and thesis advisor, Professor William J. Dally. From the day I arrived at MIT, Bill has treated me as a colleague and a partner. He has taught me the value of both breadth and depth in conducting systems research and impressed upon me that in order to make a significant contribution, one needs to span many technological levels. Bill always had the knack to ask the hard questions that inspired me to look at a problem in a new way or to continue to dig deeper. His willingness to give me leadership responsibilities taught me valuable lessons in both research and project management. Thanks also to my thesis readers at MIT, Professors John Guttag and Frans Kaashoek.

The M-Machine project was truly a collaborative effort, and without the cooperation of the design team, the chip would not have come together. Many thanks to the other members of the MAP processor design team: Andrew Chang, Whay Lee, and Nick Carter. In addition, I want to acknowledge the efforts Urban Jangren and his team at Cadence Spectrum Design, who were our industrial parters in this venture. Without their support and collaboration, this MAP chip would never have gotten much past the circuit design. I also had the pleasure of working with some very talented undergraduate and masters students including Jeff Bowers, Dan Hartman, Keith Klayman, and Albert Ma. On the software side, thanks go to Daniel Maskit, Yevgeny Gurevich, and Andy Shultz for their efforts on the M-Machine compiler and runtime system.

Great thanks go to my many friends throughout my graduate career, including three I want to mention by name. To Don Yeung, for being an awesome housemate and co-commiserator for many years. To chief yogurthead Stuart Fiske, for dragging me out of the office and making my move to California easier. And to my officemate Scott Rixner, for being the perfect foil for so many jokes and keeping the office environment amusing at so many levels.

Last, but certainly not least, many thanks go to my parents, Bill and Joyce Keckler. They have provided a tremendous amount of emotional, intellectual, and recreational support. Thanks also go to Mom for brushing up on her computer science skills so that she was willing and able to help edit my thesis. Finally, I thank them both for learning not to ask the question: "When are you going to graduate?".

The research in this thesis has been supported by the Department of Defense through a National Defense Science and Engineering Graduate Fellowship, by an Intel Graduate Fellowship, and by the Defense Advanced Research Projects Agency monitored by the Air Force Electronic Systems Division under contract F19628-92-C-0045.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Over the last 20 years, the computer industry has become accustomed to a doubling of microprocessor performance every 18 months. This exponential growth is due to improvements in silicon process technology which has produced both faster clock rates and enabled architectural innovations that have improved performance. Smaller feature sizes have decreased the transistor switching time but at the same time increased the resistance of interconnect wires, resulting in slower signal transmission in on-chip wiring. Technology has already reached a point in which VLSI designers must take wire delay into account for high-speed chips.

In order to get faster performance from computer systems, architects have turned to parallelism at two extremes of granularity: instruction-level parallelism (ILP) and coarse-thread parallelism. Very long instruction word (VLIW) and superscalar processors exploit ILP with a grain size of a single instruction, while multiprocessors extract parallelism from coarse threads with a granularity of many thousands of instructions. Instruction-level and coarse-thread parallelism both have their limits. ILP in applications is restricted by control flow and data dependencies [Wal91]. For multicomputers, there is limited coarse thread parallelism at small problem sizes and in many applications.

This thesis focuses on mechanisms that enable more parallelism to be exploited on-chip without negatively affecting the clock rates of future technologies. The architectural innovations incorporate multiple processors on a chip that are linked with fast synchronization and communication mechanisms. Threads running on different on-chip processors communicate by writing into each other's register files. They synchronize by blocking on a register that is the target of a remote write or by executing a fast barrier instruction. In addition to exploiting instruction-level parallelism, these mechanisms enable a new type of fine–grain thread level parallelism.

The fast communication and synchronization primitives are incorporated into the Multi-ALU processor (MAP) chip, which is a part of the MIT M-Machine. This thesis describes how these mechanisms are integrated into a processor pipeline and presents their implementation in a 5-million transistor custom microprocessor designed at MIT. Through simulation studies, the register-register communication and barrier synchronization instruction are shown to be significantly faster than their counterparts that use only the local on-chip cache. Using a set of real applications, the integrated interaction mechanisms of the MAP chip enable speedups of up to 2.4 times on 3 on-chip processors using fine-grain threads with run lengths of less than 300 cycles.

1.1 Technology Trends

Advances in semiconductor technology over the last twenty years have resulted in dramatic performance and density improvements. Gate delays have dropped substantially, increasing microprocessor clock rates from less than 1MHz to more than 500MHz. Architectural innovations such as pipelining, caching, and dynamic instruction scheduling have also helped push the performance of microprocessors. Figure 1.1 plots the performance of the family of Intel microprocessors over the last 20 years [Gwe95]. The diagram shows both integer performance and clock rate normalized to an 8MHz 8088 microprocessor released in 1981. Based on its past history, the computer industry and its customers have come to expect exponential performance improvements of 55% per year, with about half of the improvement coming from increased clock rate, and the rest coming from architectural advances that increase parallelism. In order to continue to get additional performance each year, future microprocessors must employ architectures that do not slow the clock.

Technology scaling is now having a profound effect on both gate and wire delay. Figure 1.2 shows the absolute delay for both transistors and on-chip wires over several generations of silicon process technologies [Sem97, Boh96]. Due to reduced transistor channel lengths, transistor switching delays are decreasing at a rate of 25% per generation. However, scaling is less kind to interconnect as the RC wire delay is doubling every generation. As wires get smaller, their cross-sectional area decreases, resulting in higher wire resistance. In addition, the wires are closer together, contributing to higher coupling capacitance between wires in the same layer. Lower resistance

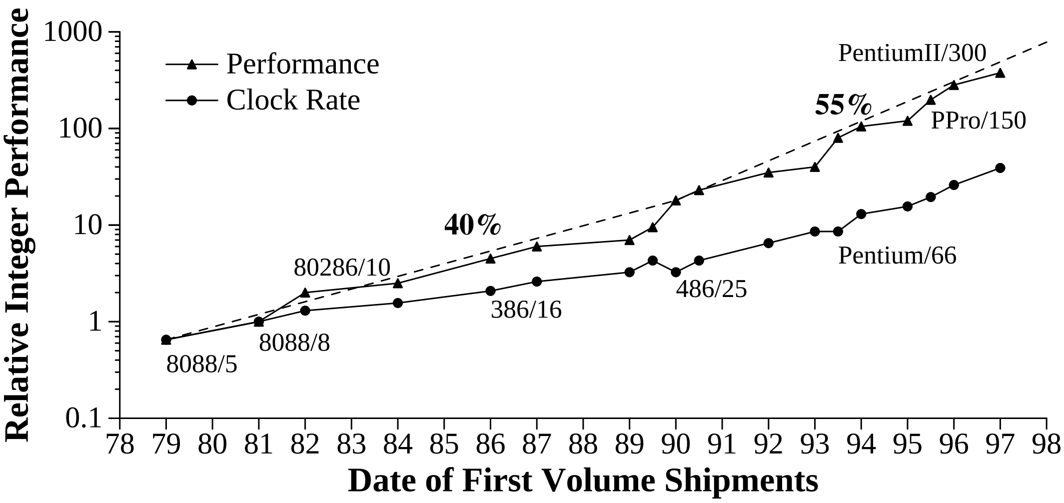


Figure 1.1: Relative performance of the Intel x86 family of microprocessors from 1979–1998, normalized to the 8MHz 8088 of 1981. The total performance improved 55% per during the 1990s, with the clock speed accounting for about half of the speedup.

---

wires such as copper, in conjunction with lower dielectric insulators would reduce the RC product at each generation, but would merely delay the inevitable impact on long wires.

Figure 1.3 shows the effect of technology scaling by plotting the delay in clock cycles to transmit a signal on a wire between two opposite corners of a chip. The computed propagation latency assumes optimal placement of repeaters in order to minimize the effect of RC delay. Nonetheless, the combination of increased wire resistance, repeater delay, faster clock rates, and larger chips results in more than 26 cycles to transmit a bit from corner to corner in a 0.1μm, 2GHz process. On-chip communication latency is no longer uniform and independent of distance. As a result, computer architects are faced with constraints limiting wire lengths in order to achieve high clock rates.

Today's microprocessor architectures, however, are on an evolutionary path that requires global communication, as shown in Figure 1.4. Superscalar microprocessors can be characterized as having both global control and global data. Centralized instruction issue logic examines an instruction window to determine which instructions can be executed simultaneously. The concurrent instructions are then delivered to the execution units which may be distributed throughout the chip. In an attempt to reduce the complexity of the issue logic, a global register file is typically used to

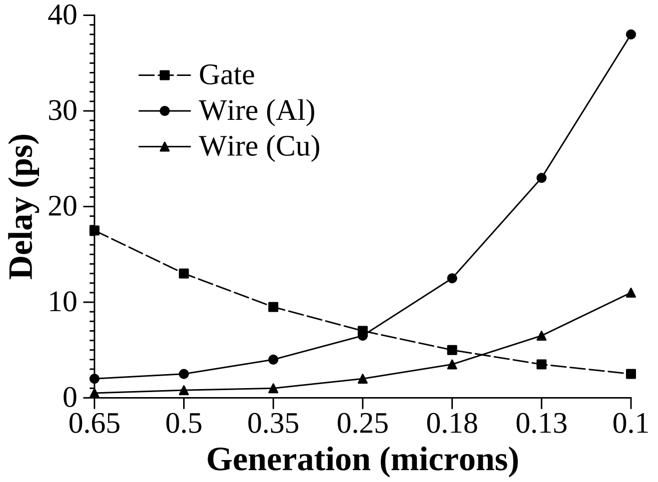


Figure 1.2: Absolute delay of transistors and wires over a several generations of silicon process technologies. The gate delay is decreasing at 25% per generation, while RC wire delay is doubling every generation.

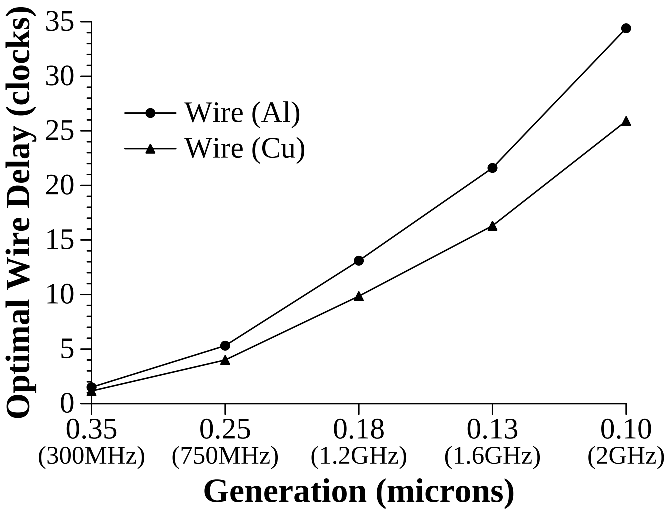


Figure 1.3: Communication latency from chip corner-to-corner using optimal repeater placement. The latency is expressed in terms of the number of clock cycles at the clock rate of the designated process generation. Corner-to-corner delay is increasing dramatically due to slower wires, faster clocks, and larger chips.

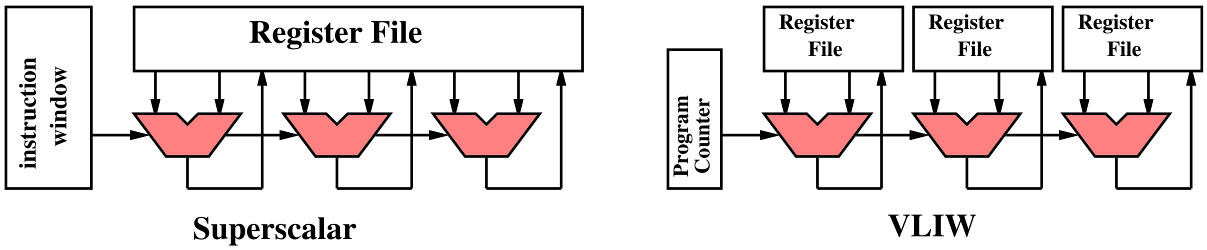


Figure 1.4: Block diagram for two modern microprocessor organizations. Superscalar architectures dynamically schedule instruction level parallelism from a central instruction window and typically employ a global register file. VLIW processors statically schedule instructions in a compiler, and can more readily use local register files.

hold the data for all of the execution units. As the number of execution units increases, the wire lengths between centralized components (the issue logic and the register file) and the distributed execution units will increase. In a 0.1μm process using copper interconnect, a signal can travel approximately 2.5mm in one 2GHz clock cycle. Thus to run at full speed, a superscalar processor in this technology must locate all execution units within one clock cycle of the instruction scheduling logic or pipeline more deeply between the scheduling logic and the execution units. Since a circular area of silicon with 2.5mm radius occupies less than 2% of the area of a 32.5mm square chip, it is clear that a single chip uniprocessor using a centralized dynamic scheduling discipline will not be efficient. Other factors will also limit the scalability of superscalar processors. The complexity of the issue logic is proportional to the product of the instruction window size and the number of execution units. In addition, the size of the register file grows quadratically with the number of register file ports, which can become large with increasing numbers of execution units.

Very long instruction word (VLIW) architectures have been built for supercomputer applications [CNO+88] and are being used in digital signal processors. In a VLIW, a programmer or compiler discovers instruction level parallelism and statically schedules the code across a series of execution units. The register files can be partitioned and placed close to an execution unit with the compiler scheduling the communication among them. The static scheduling and distributed data solves some of the constraints of superscalar architectures. However, a VLIW still has only a single program counter, and branch targets must be broadcast to all execution units.

Today's architectures that require frequent global communication are not well-matched with the

Machine	Clock Frequency	Interaction	Latency time	Latency cycles
Berkeley NOW [MVCA97]	167MHz	round trip message	21.6μs	3600
Intel Paragon [CLMY96]	50MHz	round trip message	19.9μs	995
Meiko CS-2 [CLMY96]	66MHz	round trip message	20.3μs	1340
SGI Origin [LL97]	195MHz	remote memory read	540ns	105
Sun Ultra [CPWG97]	250MHz	remote memory read	550ns	138
Hal Mercury [WGH$^+$97]	200MHz	remote memory read	1.1μs	220

Table 1.1: Interaction latencies of recent parallel computers.

technology of the near future. Instead, future architectures must exploit physical locality on-chip to keep wires short. Incorporating multiple processors on a chip that can communicate efficiently with one another via processor registers is well suited to these emerging technological constraints. By doing this, the MIT MAP chip keeps all control and most communication local to each processor, enhancing locality and diminishing the effects of slow wires. Global communication is abstracted out of the local processor design and can be optimized or pipelined independently. Compared to alternative methods of controlling large numbers of execution units, register-coupled processors offer both simple implementation and attractive speedups.

1.2 The Parallelism Gap

Conventional computer systems exploit parallelism at two extremes. Instruction-level parallelism comes from instructions in a single stream that can be executed concurrently. During scheduling, each instruction can be considered its own task, and communication between instructions typically takes place through data registers. In order to effectively exploit ILP, the communication must be fast (1–2 cycles) and have essentially no overhead.

At the other end of the spectrum are coarse grained parallel computers, with interactions requiring 100s of cycles ($\geq 1\mu$s) on conventional multiprocessors, and 1000s of cycles ($\geq 10\mu$s) on multicomputers. Table 1.1 details interaction latencies for some recent commercial and research machines. The Berkeley, Intel, and Meiko results include all components of a round trip message. The SGI, Sun, and Hal numbers include only the latency of performing a remote read that can be completely satisfied in hardware by a remote memory module. Communication between two threads running in parallel on separate processors will take much longer. In either case, exploiting parallelism on machines with such long interaction latencies requires very coarse–grain threads with infrequent communication and synchronization. Consequently, coarse-thread parallelism typically comes from outer loops of applications that are identified by hand or using a loop-parallelizing compiler.

The difference between ultra-fine instruction-level and coarse multiprocessor parallelism exposes a *parallelism gap*. This gap will continue to widen as microprocessor clock rates are increasing faster than multiprocessor interconnection network latency. The fast communication and synchronization mechanisms of the MAP chip can be used to bridge the parallelism gap between ILP and coarse–threads. Fine–grain threads which communicate and synchronize frequently can run on-chip and exploit parallelism that is not available with multiprocessor communication latencies. Because of the long interaction latencies between processors, typical multiprocessors require applications with large data sets in order to achieve significant parallel speedup. In fact, most reports of parallel speedup includes increasing the size of the applications data set as more processors are used.

However, the programs that people run every day are not large scientific applications. Many applications, such as electronic circuit simulation, have small data sets, large computation requirements, and ample concurrency. Unfortunately, simulation of a relatively small circuit that may require four hours on a uniprocessor is not likely to get faster on a conventional multiprocessor. Parallelizing this application will require partitioning of the data set across the processors. The overhead from slow multiprocessor communication mechanisms will overwhelm any benefit from fine-grain concurrency. With sufficiently fast communication and synchronization between processors, the models of the different transistors can be evaluated concurrently which will enable significant speedup even on small problem sizes. Inner–loops of applications are also prime targets for fine–grain threads. This inner–loop parallelism is orthogonal to coarse-thread parallelism and ILP, and can be used in concert with conventional methods to achieve faster performance. Since future chips will have more silicon area and include more execution units, a much larger demand for parallelism is emerging. At the same time, the increased significance of wire delay will require monolithic components, such as processors and caches, to be small and that the communication wires connecting them to be short. By using efficient mechanisms for communication and synchronization, programs can exploit fine-grain concurrency in the parallelism gap and allow parallel processing techniques to be applied to many important small applications.

1.3 The MIT M-Machine

The MIT M-Machine is designed to exploit parallelism at all granularity levels. The M-Machine consists of a collection Multi-ALU processor (MAP) chips connected via an integrated network interface and a 2-dimensional network. Each MAP chip has 3 clusters of execution units that communicate with one another through registers using an on-chip communication switch. Each cluster has three execution units, two register files, and a local instruction cache. Instruction level parallelism can be exploited within a cluster and across the clusters using register–register communication. Fine-grain threads can exploit inner loop parallelism on-chip, with the independent clusters communicating and synchronizing using the integrated interaction mechanisms of the MAP chip. Coarse grain threads are spread across multiple M-Machine nodes to exploit outer-loop parallelism.

The MAP chip, which is the focus of this thesis, is itself a prototype for future microprocessor designs. While the baseline MAP chip design contains three processors, this particular implementation, unlike current microprocessors, is scalable. The architecture of the MAP chip mitigates the effect of the long wire delays present in current and future silicon process technologies by partitioning the execution units into independent processor clusters. Three execution units are included in a cluster so that all local wires can easily be driven in a small fraction of the clock cycle. Long global wires are necessary to connect the clusters and the cache banks in the memory system. Each of these components operates independently from one another and can tolerate multiple cycles of communication latency. The computation power in the MAP chip can be scaled in two dimensions. Limited scalability is achieved by incorporating additional execution units into each cluster, subject to the local wiring latency constraints. A greater degree of scalability comes from increasing the number of processor clusters. The number of clusters is restricted by the silicon area on a chip and the scalability of the communication mechanisms between clusters. With three processors on the MAP chip, a simple crossbar communication system is sufficient. A more scalable communication substrate that exploits physical locality can enable more clusters built on a single chip without requiring long wires between remote clusters. Multiprocessor network topologies, such as meshes and

	MAP Chip	J-Machine [NWD93]	Alewife [ABC+95, LA94]		Monsoon [PC90]
Communication:					
Form	register-register	messages	shared memory	messages	dataflow tokens
Packet size	1 word	>2 words	4 words	>2 words	1 token
Latency	1 cycle	11 cycles	38 cycles	42 cycles	8 cycles
Synchronization:					
Data Form	scoreboard	message dispatch	spin lock	message dispatch	data arrival
Barrier Form	`cbar` instruction	messages	spin locks	messages	N/A
Overhead	1 cycle	55 cycles	64 cycles	128 cycles	N/A

Table 1.2: Comparison of communication and synchronization mechanisms.

trees, could certainly be used to extend the scalability of a single-chip implementation. Regardless of the configuration of the processors, the key to the scaling the execution resources on a single chip is to exploit locality.

The contributions of the MAP chip architecture come not only from the partitioning of execution units into clusters, but also from the mechanisms employed to interact among them. The MAP chip integrates communication and synchronization mechanisms into the core of each processing element which results in drastically lower latencies than available even in other experimental fine-grain parallel computers. Table 1.2 compares the on-chip communication and synchronization mechanisms of the MAP chip to those of other experimental parallel computers. The communication between different MAP processors takes place through registers, as an instruction on one processor can write into the register file of another processor with a latency of one cycle. The J-Machine integrates a network interface very closely with the processor and allows a user program to send messages directly from the contents of a register file. Two adjacent processors can communicate in 11 cycles including instruction and network interface overhead on both ends. The Alewife shared memory multiprocessor integrates hardware support for both shared memory and message passing communication between processors. Fetching data that resides in a remote cache line requires 38 cycles, while message communication between two processors requires 42 cycles.

For synchronization, the MAP chip introduces two novel instructions: `empty` and `cbar`. The `empty` instruction can modify a MAP cluster's register scoreboard which is used for data synchronization. Instructions that need data from remote processors wait in a *synchronization* pipeline stage. When a remote register write completes, the scoreboard bit for the designated register is marked full, enabling execution of any instruction that is waiting for the value. A scoreboard entry is marked full automatically when registers are written and is emptied manually using a user-level `empty` instruction. Data synchronization on message passing systems such as the J-Machine or Alewife typically takes place when the message arrives at a remote processor. The dispatch of code that handles the message can signal to the thread that is waiting for the value. In a shared memory machine, data synchronization is enforced using spin locks in memory. For control synchronization, the MAP chip implements a barrier instruction (`cbar`) which enables threads on all three processors to synchronize with only one cycle of overhead. Barrier synchronization on message passing machines, even only two processors, requires many more cycles to complete (55 cycles for the J-Machine and 128 cycles on Alewife). Shared memory barriers which are implemented using locks in memory can also be expensive, requiring 64 cycles to synchronize two Alewife processors. With its integrated mechanisms and on-chip interactions, the MAP chip can communicate and synchronize more than an order of magnitude faster than previous fine-grain parallel machines. This provides the opportunity to exploit forms of parallelism that have been previously unavailable.

In many ways, the MAP chip's mechanisms are similar to those found in a dataflow machine. The Monsoon dataflow computer was built to process dataflow tokens, which typically consist of a single instruction. When a token fires, it computes a result and transmits it to other tokens that are waiting to use that result as an operand. When the operand arrives, it can immediately enable a token for execution. Data is delivered directly from one instruction to another on different processing elements, and individual instructions stall until all of their operands arrive. The MAP chip's register communication and synchronization are analogous to the communication between computation tokens in Monsoon. Using these mechanisms, the MAP chip can provide extremely fast interprocessor communication to conventional programs and allow them to exploit parallelism at a finer granularity than ever before.

The experimental methodology used to evaluate the MAP chip includes both simulation and implementation. A highly accurate simulator is used to investigate the performance of the architecture and in particular the communication and synchronization mechanisms. The MAP chip project is also somewhat unique among academic architecture research in that it verifies the feasibility of the architecture through silicon implementation. The MAP prototype is a 5 million transistor custom VLSI chip implemented in a 0.5μm CMOS technology. All of the logic and circuit design of the MAP chip is complete, and tapeout is anticipated for May 1998. Further investigations on the hardware prototype will follow when the chips are tested and incorporated into a 16 processor M-Machine system in Autumn 1998. Building the prototype is extremely valuable because it allows us to "close the loop" by examining not only the performance of the proposed mechanisms, but also the design complexity and area cost in a real piece of silicon. Furthermore, it validates the assumptions made during high level simulation and provides the designers with valuable insight for future projects.

1.4 Contributions

The primary contributions of this research are:

- A scalable microprocessor architecture and implementation that partitions execution units into independent clusters to mitigate the effects of the emerging wire delay constraints of present and future silicon process technologies.
- Direct register-register transfers between on-chip processors for single-cycle communication between threads.
- A register scoreboard to enforce local and remote data synchronization that is updated by register writes but is also under software control. An `empty` instruction allows a user to manually invalidate registers.
- A fast barrier instruction (`cbar`) that enables threads on different on-chip processors to synchronize their instruction streams with only one cycle of overhead.
- An experimental evaluation of fine and coarse-grain parallelism, quantifying the benefit of using integrated register communication and synchronization mechanisms instead of a locally shared cache.
- A custom VLSI chip that demonstrates the physical feasibility of the mechanisms. The logic design of the chip shows how a register scoreboard and a synchronization pipeline stage unify data synchronization for arithmetic, memory, and communication instructions, and how the hardware structures can be incorporated into a microprocessor pipeline.

- A design, implementation, and physical evaluation of zero-cycle multithreading, which combines both instruction and thread level parallelism to interleave instructions from multiple threads over a common set of execution units.

1.5 Background

Computer systems designers have exploited parallelism to help improve application performance. However, the nature of the parallelism that can be exploited is dictated largely by the overhead to communicate and synchronize among the parallel components. The study of synchronization cost performed by Chen, Su, and Yew [CSY90] explored a spectrum of granularities including instruction, statement, and loop level parallelism. They found that statement oriented parallelism was far more sensitive to synchronization overhead than loop level parallelism. However, even with substantial synchronization overhead the statement level parallelism still yielded 4 to 20 times speedup over sequential. This study suggests that the amount of fine-thread parallelism available in applications is considerably greater than what we have exploited so far using simple approaches to parallelization, and that it scales well beyond three processors. It also shows that to extract this parallelism requires very low-overhead synchronization.

Previous architectures and machines have exploited parallelism at a large grain size, typically using loops, because they did not have sufficiently low latency communication mechanisms. However, some architectures have integrated fine–grain mechanisms into their computation engines to enable faster interactions among parallel threads.

Register Communication: The Cray X-MP implemented two central vector processing units with a bank of shared address, scalar data, and semaphore registers that could be accessed by either processor [RR87]. These registers were typically used for self scheduling of loops. The registers were not general purpose and values were copied to a processor's local register set prior to using the data. The CMU iWarp employed a form of register–register communication between processors to enable systolic communication and computation [BCC+90, PSW91]. Communication channels were established between processors, and words could be sent to a remote communication unit through the channel. At the receiver, a communication unit copied the data into a register visible by a computation unit.

Multithreaded Systems: Architectures that support fine–grain threads in a multiprocessor typically implement fast thread creation and dispatch mechanisms. The *T architecture, whose threads are in the range of 15 instructions, implements `fork`, `join`, and `next` instructions that interact with a memory task queue and a synchronization coprocessor to allow threads on different processors to communicate with one another [NPA92].

Like the MAP chip, the Tera Computer System [ACC+90] also exploits fine-grain threads using a multithreaded multiprocessor architecture. In a Tera machine, interaction between threads takes place only through memory, and full-empty bits are provided on each memory location to enable fast synchronization. In addition, Tera's architecture penalizes single threaded code by providing no support for data locality, and by a hardware scheduling policy which prohibits a single thread from using the execution resources on every cycle.

On-chip Thread Parallelism: The Hydra and Simultaneous Multithreading (SMT) architectures aim to scale on-chip parallelism beyond the limits of ILP. The Hydra architecture explores the design tradeoffs of building a single-chip multiprocessor, focusing on the memory system [NHO96]. Coarse grained tasks execute independently and communicate via a level-1 or level-2 cache. SMT adds multithreading to a traditional superscalar to exploit both instruction and thread-level parallelism [TEL95]. Execution resources are dynamically assigned to different threads, and instructions from them may execute simultaneously. Both Hydra and SMT provide only memory-based mechanisms for communication and synchronization between threads and are thus limited to threads that can tolerate longer communication latencies. The work in this thesis is complementary to these projects in that register-based mechanisms could easily be incorporated into these architectures, extending the granularity of parallelism they are able to exploit.

The Multiscalar architecture attempts to deduce fine–grain parallelism at runtime [SBV95]. Basic blocks of the program are assigned dynamically to different execution units, while hardware is responsible for enforcing the data dependencies among the blocks. Communication takes place via a unidirectional ring to which each thread can read or write. This promising approach to extracting speculative fine-thread parallelism is well matched to implementation using register-based mechanisms in lieu of the special hardware suggested in [SBV95].

1.6 Thesis Roadmap

This thesis focuses on communication and synchronization mechanisms that are used to exploit both instruction level and fine-thread parallelism across multiple processors within the MAP chip. Chapter 2 describes the architecture of the MAP processor, including the organization of execution units, the memory system, and the network interface. This chapter also details the physical implementation of MAP architecture in a 5 million transistor custom VLSI chip using a 0.7μm drawn process. An area analysis of the components of the MAP chip show that an implementation of a MAP processor with all of the architecture's floating-point units and more on-chip memory capacity and bandwidth could easily be built using a 0.35μm process.

Chapter 3 details the cluster pipeline of the MAP chip, including the bypassing and instruction issue mechanisms that enable fast cluster interaction. This chapter presents a new synchronization pipeline stage that determines when a thread can execute its next instruction, based on the instruction's data requirements and the availability of execution resources. In addition, the synchronization stage dynamically interleaves instructions from multiple threads on a cycle-by-cycle basis to tolerate instruction, memory and communication latencies.

In order to determine the benefit of the MAP's on-chip interprocessor communication and synchronization mechanisms, a simulation study using the Verilog logic design of the MAP chip was performed. This study, detailed in Chapter 4, shows that a remote thread can be created in 11 cycles, and that two threads can communicate or synchronize in a single cycle. Communication is 10 times faster and synchronization is 60 times faster than using only an on-chip cache.

Chapter 5 describes how the register communication and synchronization mechanisms can be used to exploit instruction-level parallelism across the independent MAP processors. Hand scheduling of procedures that contain ILP splits the code into independent streams that execute simultaneously on separate clusters. Synchronizing explicitly when communication is necessary is shown to be competitive with the implicit synchronization of a VLIW.

The boundaries of the parallelism gap are explored in Chapter 6. A suite of applications is characterized by parallelizing their inner and outer loops to quantify the benefits of the MAP's mechanisms over using just a shared on-chip cache. On the fine–grain parallelism that is found in the inner loops, the MAP's mechanisms provide speedups of up to 2.4 times using 3 on-chip processors, while communicating through memory yields less speedup and sometimes slowdown. On coarser grain outer-loop parallelism, processor interactions are infrequent enough that the MAP's barrier instruction provides little performance improvement.

Chapter 7 discusses some of the lessons learned from the MAP design and implementation, including the value of controlling complexity. Finally, Chapter 8 contains the conclusions of this work and outlines future research.

Chapter 2

M-Machine Overview

The M-Machine is intended to exploit parallelism at all levels and to extract more parallelism from problems of fixed size, rather than requiring enormous problems to achieve peak performance. The M-Machine consists of a collection of computing nodes interconnected by a bidirectional 2-D mesh network [FKD+95]. Each six-chip node contains a Multi–ALU (MAP) chip and 1 MWord (8 MBytes) of synchronous DRAM (SDRAM) with error correction (ECC). Three clusters of execution units are implemented on each MAP chip, and mechanisms are employed to enable fast communication and synchronization between them. This enables the on-chip execution units to interact frequently and exploit both instruction and thread-level parallelism. The MAP chip also includes a network interface and router that are integrated into the execution unit pipeline. The bandwidth from the processor core to the local synchronous DRAM (SDRAM) and to each network channel is balanced at 8 bytes per processor cycle. The low latency and high bandwidth network interface and router allows threads on different MAP chips to communicate efficiently. To connect to peripheral devices, the MAP chip includes a dedicated I/O bus. In the M-Machine, I/O devices may be connected to either every node or a subset of nodes, such as those on an edge of the mesh.

This chapter focuses on the architecture and implementation of the MAP chip. Section 2.1 describes the components of the MAP chip, including the processing, memory, and communication subsystems. Sections 2.2 through 2.4 discuss the physical implementation of the MAP chip and how the design was modified to meet the on-chip area constraints. Section 2.5 analyzes the scalability of the MAP chip for future process generations. Finally, Section 2.6 concludes with some interesting lessons about conducting a project such as this in an academic environment.

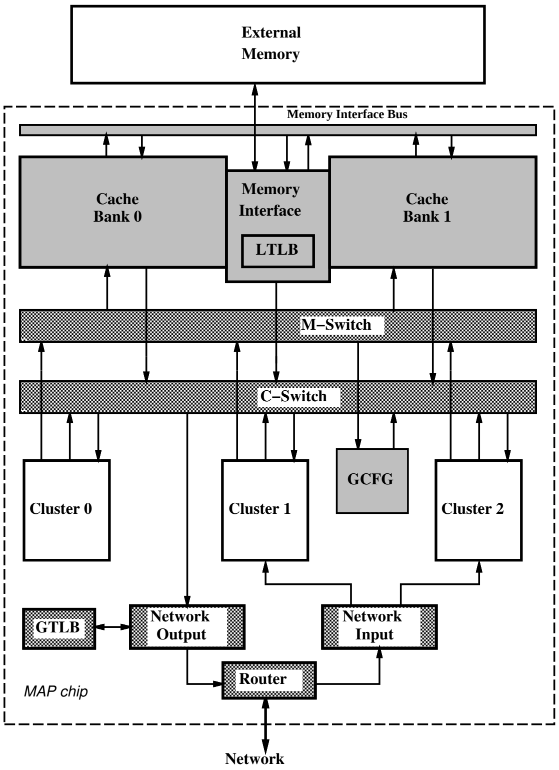


Figure 2.1: Block diagram of the MAP architecture.

2.1 The MAP Chip Architecture

As shown in the block diagram of Figure 2.1, a MAP chip consists of four major subsystems: processing clusters, communication switches, memory banks units, and a network unit. The processing subsystem contains three execution clusters, each of which is an independent processor. A cluster includes an instruction cache, two register files, and three execution units. The memory subsystem is composed of a shared unified cache organized into two banks so that it can process two memory requests simultaneously. The Global Configuration Space (GCFG) controller enables the control registers of the MAP to be accessed via memory operations. Two crossbar switches interconnect these components. Clusters make memory requests to the appropriate bank of the interleaved cache over the 142-bit wide (51 address bits, 66 data bits, 25 control bits) 3×2 M–Switch. The 88-bit wide (66 data bits, 22 control bits) 7×3 C–Switch is used for inter-cluster communication and to return data from the memory system. Both switches support up to three transfers per

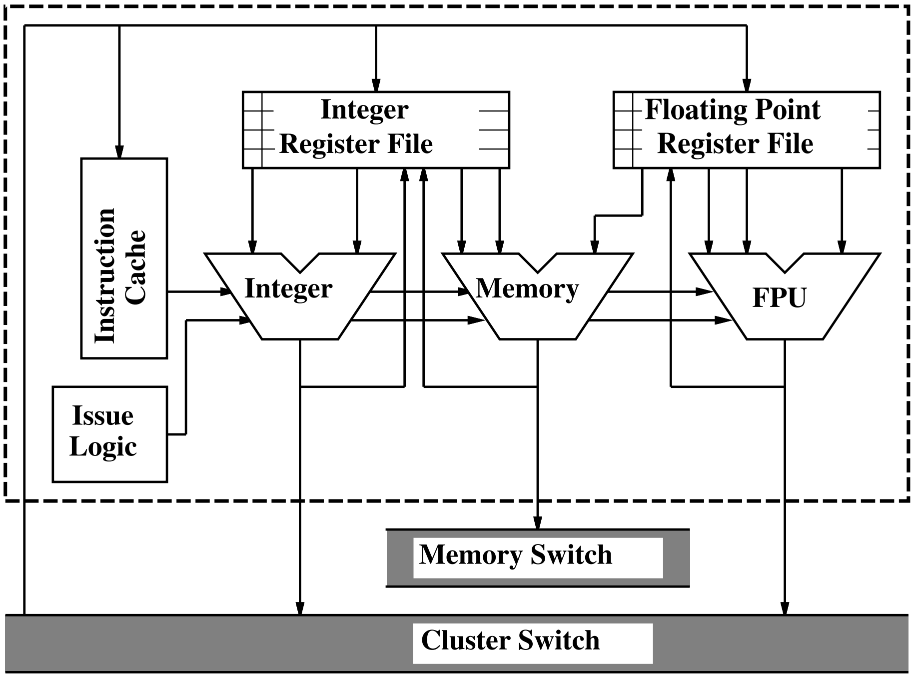


Figure 2.2: A MAP cluster consists of 3 execution units, 2 register files, an instruction cache and ports onto the memory and cluster switches.

cycle; each cluster may send and receive one transfer per cycle. The network subsystem contains the network interface units and a router. A cluster can communicate with another MAP chip by injecting a message directly into the router, which automatically delivers the message to the destination using the routers on intermediate MAP chips.

2.1.1 MAP Execution Clusters

Each of the three MAP clusters is an independent 64-bit pipelined processor with a local 4KB instruction cache. Figure 2.2 shows a block diagram of a cluster, including three execution units, two register files, and interfaces to the global switches. The integer unit executes arithmetic, logical, shift, and comparison operations. The memory unit executes load and store operations to memory, as well as most of the integer unit operations, excepts shifts and compares. The floating-point unit includes a 4 stage multiply-add pipeline and an iterative divide/square-root unit. It executes all floating-point operations, as well as integer multiply and integer divide.

The integer and floating-point register files each contains six banks of 16 registers. The integer register banks share four read ports and three write ports that are read and written independently by both the integer and memory units. The floating-point register banks are accessed through four read ports and two write ports. The floating-point unit reads three operands (for multiply-accumulate) and writes one result to the floating-point register file. The memory unit also reads one operand from the floating-point register file for floating-point store operations. The additional write port on each register file allows the Cluster Switch to update registers without interfering with local writes. Integer register 0 (*i0*) and floating-point register 0 (*f0*), are both hard wired to the value zero, while integer register 1 (*i1*) is mapped to the program counter. Both the integer and floating-point units can write results to remote register files via the Cluster Switch. The Cluster Switch is also used to move data between the integer and floating-point register files on the same cluster. The memory unit sends its load and store instructions to the shared on-chip cache using its port to the Memory Switch. Misses in the instruction cache generate load operations that go to the unified cache as well. In addition to the data registers, a cluster also has six banks of 16 one-bit condition code (CC) registers. These registers hold results of comparison operations and are used for conditional branches and predicated execution of instructions. Of the 16 condition code registers, 8 are reserved for local access while the other 8 are global and can be written by remote clusters.

Each MAP instruction may contain 1, 2, or 3 operations, with one operation for each execution unit. Operations are scheduled statically by a compiler and packed into instruction words. The MAP's instruction scheduling hardware does not reorder instructions. Operations in a single instruction must issue together but may complete out of order. Synchronization between instructions must take place through hardware enforced data dependencies in order to prevent write-after-write hazards to the register file. Load and store operations to the memory system that issue from a given cluster will access the memory system and complete in order. Every operation may be conditionally executed depending on the one-bit value of one of the condition code registers.

Concurrency is exploited within a cluster using multithreading. Up to six instruction streams may be simultaneously loaded in the processor pipeline, each residing in its own *thread slot* consisting of a set of pipeline registers and private portions of the register files. Each thread can access one bank of the integer, floating-point, and condition code register files. The instruction streams are called V-Threads, as instructions from them drop vertically into a common set of execution resources on a cluster. Instructions from different V-Threads are interleaved over the execution units on a cycle-by-cycle basis. If one thread is waiting for a result from a previous instruction, another thread may use the execution units instead. Consecutive instructions entering the execution pipeline may be from distinct V-Threads, or they may be from the same V-Thread. This flexible interleaving allows the MAP to exploit multithreaded parallelism and to mask variable pipeline, memory, and communication delays. The pipeline design required to implement this multithreading is discussed further in Chapter 3.

The MAP also supports concurrency by executing threads in parallel across its arithmetic clusters. Threads executing on different clusters are known as H-Threads since they can enter their own execution pipelines horizontally and concurrently. H-Threads that occupy the same thread slot number on different clusters are members of the same V-Thread. To facilitate closer interaction between clusters, H-Threads within the same V-Thread may communicate and synchronize with one another by writing into each other's register files. Two threads that reside in different V-Threads can communicate with one another through memory. Because memory operations issued by threads on separate clusters may complete out-of-order, synchronization through registers, a cluster barrier, or on a memory location is necessary to coordinate memory communication between clusters.

The combination of multiple clusters and fast interactions among them can be used to support multiple execution models, including instruction, thread, and loop level parallelism. Because of the integrated register communication, the MAP chip can exploit fine-grain thread level parallelism that would be infeasible on a traditional multiprocessor.

To exploit instruction-level parallelism, the compiler can schedule an instruction across all three clusters using H-Threads from the same V-Thread. The compiler must insert explicit register-based synchronization operations or use the cluster barrier instruction to enforce instruction ordering between H-Threads. Unlike the lock-step execution of traditional VLIW machines, H-Thread synchronization occurs only as required by data or resource dependencies. While explicit synchronization incurs some overhead, it allows H-Threads to slip relative to one other in order to accommodate variable–latency operations such as memory accesses. These intercluster communication synchronization and communication mechanisms are examined in more detail in Chapter 4.

The partitioning of V-Threads and H-Threads across the clusters is shown in Figure 2.3. The

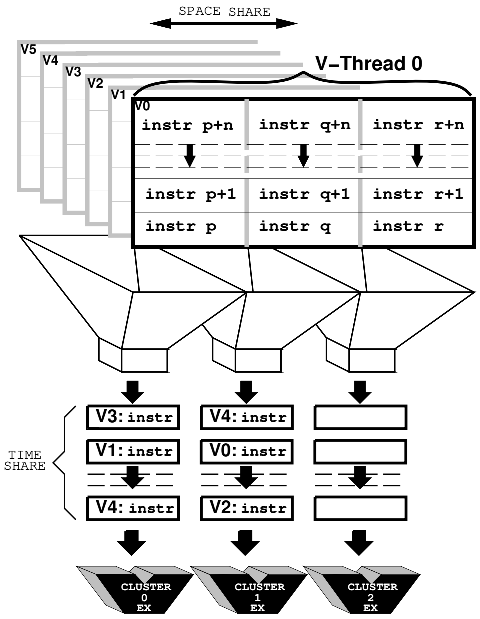


Figure 2.3: Multiple V-Threads are interleaved dynamically over the cluster resources. Each V-Thread consists of 3 H-Threads which execute on different clusters.

state for each of the six V-Threads (V0–V5) is stored in the pipeline registers and register files in each cluster. One H-Thread from each V-Thread resides on each cluster and consists of a sequence of 3-wide instructions containing integer, memory, and floating-point operations. As shown for cluster 0, the instructions from each V-Thread are dynamically interleaved over the execution units, so that an instruction from V3 can follow one from V1 without any pipeline stalls. A cluster's execution resources are time-shared by those V-Threads that are executing. Each cluster is independently controlled, and instructions from the same V-Thread need not be executed simultaneously on all clusters. The parallel clusters and multithreading at each cluster allow the execution resources to exploit both instruction and thread-level parallelism and achieve high utilization of the function units.

2.1.2 Memory System

The MAP's memory system is designed to provide high bandwidth and low latency access to the on-chip cache and off-chip memory. As illustrated in Figure 2.1, the 32KB unified cache is organized as two word-interleaved 16KB banks which permits references to consecutive addresses to proceed in parallel. On a memory reference, a virtual address is used to access the cache directly and no virtual to physical translation is required. However, since all processes reside in the same global virtual address space, the cache will never contain any aliases. Each cache line consists of 8 words (64 bytes). The external memory interface includes virtual memory support and a synchronous DRAM (SDRAM) controller to manage the off–chip memory. The SDRAM controller exploits the pipeline and page modes of the external memory and performs single error correction and double error detection on the memory transfers.

Each cache bank receives memory operations from the clusters via the Memory Switch. For a `load` operation that hits in the cache, the data is immediately returned to the requesting cluster on the Cluster Switch. Operations that miss in the cache are delivered to the external memory interface (EMI), which accesses the local translation lookaside buffer (LTLB) to find a virtual to physical translation for the requesting address. If a translation is found, the EMI accesses the SDRAM, returns the data to the requesting cluster through the Cluster Switch, and inserts the line containing the data into the cache. If a translation is not found, an LTLB miss event is triggered and handled in software on cluster 0. Physical page frames and virtual pages are 4 Kbytes (64 8-word cache blocks) in length. Cache hits have a 3 cycle latency including both Memory and Cluster Switch traversal. Cache misses require 8–15 cycles to resolve, depending on whether the SDRAM can be accessed in page or pipeline mode. The memory system implements uncached load instructions which allow data to be accessed without polluting the cache. In addition, it implements two block buffers which serve as both a victim cache and a write buffer when a cache block is evicted [Jou90].

In memory, each MAP word is composed of a 64-bit data value, one synchronization bit, and one pointer bit. The synchronization bit is used to implement fine-grain memory synchronization on a word-by-word basis. A pair of special load and store operations specify a precondition and a postcondition. If the precondition matches the synchronization bit, the bit is set to the postcondition and the load or store completes normally. Two possible outcomes exists if the precondition test fails. If the memory operation is *unfaulting*, then the programmer is notified in a condition code register that the operation failed. The program can then retry the operation and spin if necessary. If the memory operation is *faulting*, the request is placed in a hardware queue and nothing is returned to the program. A software handler can later retry the memory operation or cause the user program to block.

The pointer bit is used to provide data and code protection in a single global virtual address space through guarded pointers [CKD94]. Guarded pointers implement a light-weight capability system that organizes the global address space into segments which must be accessed with unforgeable pointers. Paging manages the relocation of data in physical memory within the virtual address space. The protection and paging mechanisms are independent so that data integrity may be preserved on variable-size segments of memory. Segmentation checks are performed in the cluster during address calculation. When a memory operation executes, the permission of the pointer is examined to determine if the operation is legal. If the check determines that a memory operation is illegal, a cluster's memory unit triggers an exception. Data is controlled on a segment-by-segment basis which can prevent protected data from being read, and read-only segments from being written. These mechanisms of guarded pointers enable multiple protection domains to be resident in the processor simultaneously, and allow an individual thread to change its addressing environment very inexpensively

The memory system also includes support for sharing data across multiple MAP chips. Each LTLB and page table entry includes two *block-status* bits for each cache line in the page (128 bits total). These bits encode four possible cache line states, including `Read-only`, `Read-write`, `Dirty`, and `Invalid`. Copies of a cache line may reside simultaneously on different M-Machine nodes, and the block-status bits are used to help keep the data coherent. For example, a write to a `Read-only` cache line will trap to a software routine which can then retrieve an exclusive copy of the line by sending a message to the home node of the data. Remote data can be cached locally in both the on-chip cache and local memory. A more complete description and evaluation of the hardware and software support for shared memory on the M-Machine can be found in [Car98].

2.1.3 Global Configuration Space

The Global Configuration Space (GCFG) controller enables a program to access the MAP chip's internal registers and control state. These locations are mapped into the configuration address space, which is separate from the virtual and physical memory address spaces. GCFG requests are accepted from the Memory Switch and results are returned via the Cluster Switch. Centralized state, such as the global cycle counter, the performance monitoring controllers and counters, the thread status bits, and the I/O interface are located within the GCFG controller.

The GCFG controller communicates with local configuration space controllers to access other state that is distributed throughout the MAP chip. In each cluster, all of the registers and scoreboards can be read and written via configuration space. To read a remote cluster's register, a load is performed to the appropriate configuration space location. Upon receiving it, the GCFG controller forwards it over the Cluster Switch to the local configuration space (LCFG) controller in the target cluster. The LCFG controller accesses the register by injecting a synthetic operation into the pipeline that delivers the data to the requesting cluster using the Cluster Switch. In this fashion, threads can be started, stopped, and swapped in and out by using a sequence of load and store operations. Since remote access using the GCFG is somewhat slow, it is not intended to be used intensively by application programs. However, some thread control instructions are made available to the user. Both the `hfork` and `hexit` instructions, which allow a thread to be created and destroyed, are interpreted as GCFG store operations. In response to one of these requests, the GCFG performs a series of transactions that update the appropriate registers in both the GCFG state and the cluster. This serves as a shortcut for creating and terminating threads, allowing faster thread interaction with less overhead.

2.1.4 On-Chip Switches

The Memory and Cluster Switches connect the different asynchronously executing components of the MAP chip. The Memory Switch allows clusters to make memory requests to both of the on-chip cache banks as well as the Global Configuration Space controller. The Memory Switch has 3 input ports, one for each cluster, and 2 output ports, one for each cache bank; the GCFG controller shares an output port with Bank 1. In each cluster, both the memory unit and the instruction fetch unit compete for the Memory Switch port, using a round-robin arbitration scheme.

The Cluster Switch is used to return requested data from the memory system to the clusters, to allow clusters to communicate with one another using register-register transfers, and to transfer the contents of outgoing messages from the registers of the sending cluster to the network output unit. Most Cluster Switch transactions include only a single word, but a burst mode is used to transfer atomically a stream of words from the cluster to the network interface for `send` instructions.

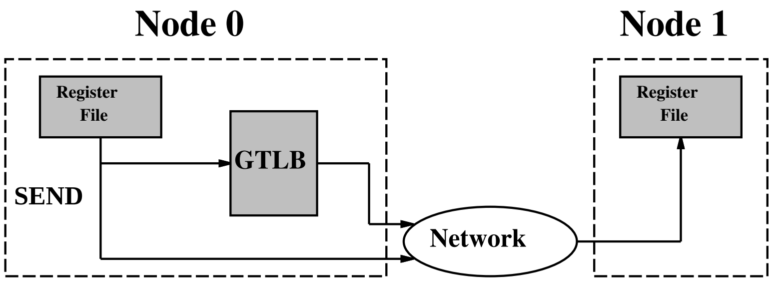


Figure 2.4: The MAP `send` instruction transfers data from the register file on one M-Machine node to the register file of another.

Arbitration is performed among the seven requesters to determine which requests will be satisfied. External Memory Interface (EMI) requests have the highest priority, followed by the Global Configuration Space controller. At the next tier, round robin arbitration selects between the two cache banks. At the lowest level, a round robin strategy selects among the three clusters. Since all data returned from the memory system is ultimately in response to cluster based memory requests, placing the clusters at the lowest priority for the Cluster Switch provides natural backpressure to the execution units. Thus a cluster's execution units will stall if previous switch transactions are clogging the Cluster Switch.

2.1.5 Communication Subsystem

In order to reduce the latency to communicate between different MAP chips, the M-Machine provides a fast, protected, user-level message passing substrate [LDK+98]. Each MAP chip includes a highly integrated network interface and a 2-dimensional mesh router. As shown in Figure 2.4, messages are composed in a thread's general registers and launched atomically with a user-level `send` instruction. A state machine sequences the message contents over the Cluster Switch to the network output unit (NETOUT). The NETOUT injects the message into the network using a port into the router located on-chip. The message is routed to its destination using the routers on other MAP chips that lie along the message's path. When the message arrives at its destination, it is queued in a hardware FIFO that is mapped to a register in an H-Thread belonging to the system V-Thread (V-Thread 5). A system-level message handler removes the message contents from queue, and performs the required action. Two network priorities are provided, one each for requests and replies. Messages are routed in dimension order using up to four virtual channels. The router connects MAP chips together using 36 pins for each of the 4 physical channels. The network operates at twice the clock frequency as the MAP core so that one 64-bit word can be sent every cycle. Adjacent MAP chips communicate using a bidirectional signalling discipline, in which they both can simultaneously transmit and receive data on the same wire [DLD93].

Most message passing computers prevent users from monopolizing the network and from communicating with unauthorized remote processors by only allowing protected system code to access the network interface. The MAP chip eliminates the system call overhead of messaging by employing a `send` instruction, which specifies a destination pointer, an instruction pointer (*handler_ip*), and a message length of up to 8 words. To prevent a user from sending a message to an unauthorized node, the destination must be a pointer to a virtual address. The `send` instruction determines whether the pointer is valid, and the NETOUT unit automatically translates the pointer into a physical node identifier via a global translation lookaside buffer (GTLB). The GTLB caches entries of a software global destination table (GDT), much like a TLB caches page table entries. The *handler_ip* specifies what procedure is to run when the message arrives at the destination. In order to restrict the code that can be invoked remotely, the `send` instruction checks that the *handler_ip* is a valid pointer of type *execute_message*.

2.1.6 Exceptions

The MAP chip minimizes the downtime of threads due to exceptions by reducing the overhead for each exception and providing mechanisms to eliminate as many exceptions as possible. On each cluster, V-Thread 3 is reserved for handling local exceptions that can be detected during the first half of the execution unit clock cycle. These exceptions include executing a privileged instruction while in user mode, storing to an illegal pointer, and sending to an illegal address. When the execution unit detects an exception, it stalls the pipeline and writes the information associated with the exception into the registers of the dedicated *exception* V-Thread. Since the exception is executed in its own thread slot, the thread that caused the exception merely waits until the exception is complete. No user registers need to be saved and restored and the pipeline does not need to be restarted. As a result, a null exception handler can start and complete in less than 5 cycles.

The exception record includes the address of the instruction that faulted and the reason for the exception. At that time all user threads are prohibited from executing instructions, so that a second exception cannot occur while the exception handler is busy. When the exception handler is complete, it re-enables the user threads. An exception caused by the exception handler or by a system level event handler is a system software error and results in an unrecoverable catastrophic exception.

In order to enable speculative execution of instructions, the MAP provides a mechanism for deferring exceptions. Deferred exceptions can result from creating of an illegal pointer or loading from an illegal address. When a deferrable exception is detected, a special pointer called an ERRVAL (error value) is written into the result register. The ERRVAL is a tagged guarded pointer that encodes the address of the instruction that created it, and the reason for its creation. ERRVALs can propagate through subsequent arithmetic instructions, allowing a stream of speculative instructions to occur without the risk of an unwanted exception. Instructions that have no result (such as a store) and comparison operations that write into a single-bit condition code register cannot propagate an ERRVAL and must take an exception.

2.1.7 Events

Exceptions that occur outside the MAP cluster are termed *events* and are handled asynchronously by generating an *event record* and placing it in a hardware event queue. Local TLB misses, block status faults, memory synchronizing faults, and message arrivals are events that are handled without blocking execution of any user level program. These events are precise in the sense that the faulting operation and its operands are specifically identified in the event record, but they are handled asynchronously, without stopping the thread. Each H-Thread in V-Thread 5 handles one class of events. Local TLB misses are handled on cluster 0, and arriving messages are handled on clusters 1 and 2, depending on the priority of the message. Memory synchronization and status faults are handled in V-Thread 4 and can use all three H-Threads in its thread slot to execute the event handler.

The dedicated handlers process event records to complete the faulting operations. When an LTLB miss occurs, the external memory interface hardware formats an event record containing the

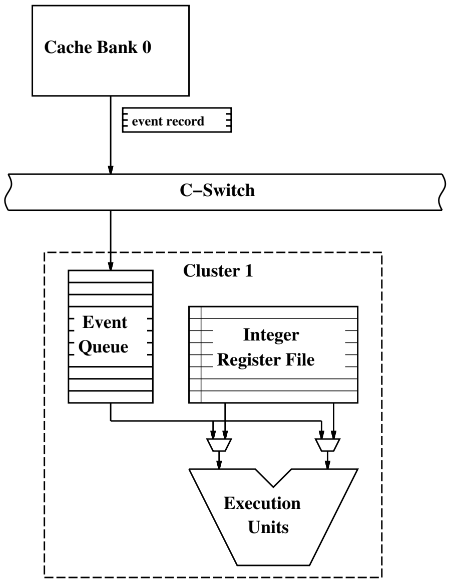


Figure 2.5: An event, such as a synchronization failure, writes an event record from the cache bank into the event queue in Cluster 1.

---

faulting address as well as the data to write or the address to read. The event record is written directly into the register file of V-Thread 5 on cluster 0. The software TLB miss handler reads the record, places the requested page table entry in the TLB, and restarts the memory reference. The thread that issued the reference does not block until it needs the data from the reference that caused the miss.

For message arrival and general memory events, the event record is written into a hardware queue. Figure 2.5 shows the path from a cache bank into the event queue for a memory event such as synchronization failure. For these events, a handler reads event records from the queue and processes them sequentially. Integer registers are mapped to the queue so that a read to *i14* will dequeue the word at the head of the queue, and a read to *i15* will pop the current event record from the queue and return the first word from the next record. Reading from an empty queue causes the event handler to stall until the next word is available. An arriving priority 0 message is placed in the queue of cluster 1 while priority 1 messages are handled in cluster 2.

Handling events asynchronously obviates the need to cancel all of the issued operations following the faulting operation, a significant penalty in a 9-wide machine with deep pipelines. Dedicat-

	Cluster 0	Cluster 1	Cluster 2
V-Thread 0	user	user	user
V-Thread 1	user	user	user
V-Thread 2	user	user	user
V-Thread 3	exception	exception	exception
V-Thread 4	general events	system	system
V-Thread 5	LTLB events	priority 0 message	priority 1 message

Table 2.1: The six threads are partitioned into three user slots, 1 exception slot, and two event slots. Synchronization and coherency events can use all of the clusters in V-Thread 4 if necessary.

ing H-Threads to this purpose accelerates event handling by eliminating saving and restoring of thread state and allowing concurrent (interleaved) execution of user threads and event handlers. Asynchronous event handling does require sufficient queue space to handle the case where every outstanding instruction generates an event. If insufficient space exists in the event queue, user threads must be stalled to prevent additional events from overflowing the queue. In the MAP chip, as many as 13 memory instructions may be outstanding in various memory system pipeline stages. Since each event record is four words, the event queue must be at least 52 words long. The MAP implements a 128 word event queue so that user threads need not be prematurely stalled.

2.1.8 Summary

The MAP chip enables a high degree of on-chip parallelism with all of the components designed for concurrency. Each of the three clusters is able to extract instruction-level parallelism using its three execution units. Both instruction and thread-level parallelism can be executed across all three clusters using the fast intercluster communication and synchronization mechanisms. Two memory operations can access the on-chip cache simultaneously, and the paths to and from the memory system are deeply pipelined. Multithreading allows the execution resources of a cluster to be used when one thread stalls for a short or long period of time. Table 2.1 shows how the different thread slots are allocated to user programs and system services. User threads can be placed in thread slots 0, 1, and 2, while the remaining slots are reserved for specialized system code. To increase concurrency, even the exception and event systems allow user programs to continue running in situations where other processor designs require user code to suspend. For example a message

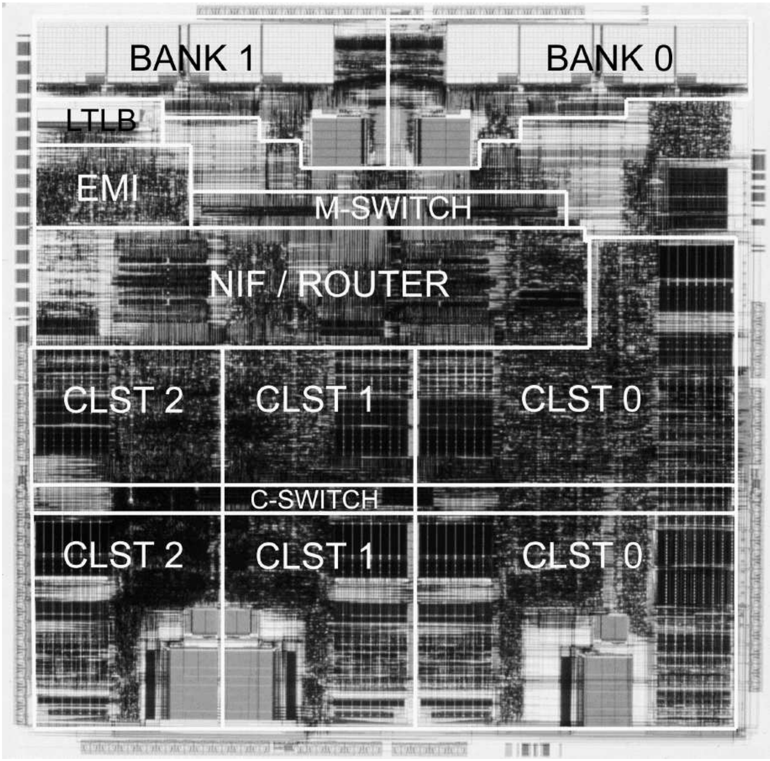


Figure 2.6: Preliminary plot of the MAP chip, measuring 18.3mm × 18.25mm and containing approximately 5 million transistors.

arrival or a TLB miss does not stall any processor or require a context switch. The hardware resources are available to immediately service asynchronous events simultaneously with execution of user code. Coupling these on-chip mechanisms with the integrated network interface and router enables parallelism at all levels to be exploited across an M-Machine multiprocessor composed of multiple MAP chips.

2.2 MAP Implementation

A preliminary layout plot of the entire MAP chip is shown in Figure 2.6. The chip is 18.3mm × 18.25mm and consists of approximately 5 million transistors in a 5 metal layer, 0.7μm drawn (0.5μm effective) process. Approximately 2.7 million transistors are in SRAM memory arrays, 1.2 million are in the datapaths, and 1.1 million are in random logic. In terms of the number of logic

Pin Use	Count
Network	160
Memory interface	98
Special power/electrical control	96
I/O Channel	23
Diagnostic Interface	11
Clock	5
Power/Ground	631
Total:	1024

Table 2.2: Breakdown of pin usage on the MAP chip.

transistors, the MAP is similar to the Sun Ultra II microprocessor [GAB+97]. IBM Corporation is manufacturing the MAP chip for MIT. The package for the MAP is a 1024 pin ball grid array, which uses 393 pins for signals and 631 for power and ground. Table 2.2 summarizes the pin usage on the MAP chip.

The physical organization of the MAP chip is quite similar to the block diagram in Figure 2.1. The cluster datapath and control modules occupy the bottom 60% of the chip, the network interface (NIF) and router are in the middle 15%, and the memory system is in the top 25%. Cluster 0 is significantly larger than the others since it includes a floating-point unit. Within cluster 0, the floating-point datapath runs along the right-hand side, the integer and memory unit datapaths are the left, and the instruction cache is the regular structure at the bottom. The remainder of the area in a cluster is occupied by control logic. The translation lookaside buffer (LTLB) and the external memory interface (EMI) are both located near bank 1. The event queue and the global configuration space controller sit between the floating point unit and cache bank 0.

The Cluster Switch runs horizontally in metal-4 at the midpoint of the clusters and consumes only 8% of the metal-3 and metal-4 routing in the cluster region. However, the wiring congestion near the Cluster Switch is significant since the switch runs over the cluster pipeline control modules. The Memory Switch is at the bottom of the memory system region and occupies about 6% of the metal-3 and metal-4 routing resources there. Table 2.3 summarizes the area costs for the different components of the chip. The area is expressed both in mm^2 and λ^2, where λ is one half the feature size. For our process, $\lambda = 0.25\mu m$, which is one half of the minimum effective transistor length.

Component	Area (mm^2)	($M\lambda^2$)	% of total area
Integer Units (3)	55.9	894	16.7
Memory Units (3)	42.4	678	12.7
16KB Data Cache Banks (2)	36.9	590	11.0
Floating–point Unit	33.4	534	10.0
NIF/2-D Router	26.8	429	8.0
I/O Pads	26.6	426	8.0
Instruction Caches (3)	17.7	283	5.3
EMI + 64 entry TLB	8.3	133	2.5
Clock drivers	5.7	91	1.7
Event Queue	5.6	90	1.7
GCFG	3.3	53	1.0
Switch drivers	3.1	50	0.9
Misc. Control/Wiring	68.3	1089	20.5
Total	334	5340	

Table 2.3: Area costs for the components of the MAP chip.

2.3 Design Methodology

The design and implementation of the MAP chip was truly a team effort, with cooperation between MIT and several industrial partners. The core team at MIT consisted of one faculty member, three PhD students, and one research staff member. All of the logic design, logic validation, circuit design, and timing analysis was performed at MIT. Over the course of the project, six masters and undergraduate students contributed to the design as well. A portion of the physical design and layout was performed by the Microelectronics Center of North Carolina. Cadence Spectrum Design Services in Rancho Bernardo, California was the principal industrial partner. Cadence did the clock distribution design and analysis, provided tools and methodology that smoothed the design flow, and completed the bulk of the physical design, including cell layout, chip assembly, and verification.

In order to reduce the amount of effort for our small team, we tried to use an efficient design methodology, consisting of some full-custom and some cell-based layout. All of the SRAM arrays, including the instruction and data caches, as well as the TLB, are full-custom designs with circuits that were carefully evaluated using the HSPICE circuit simulator. Most of the design uses static CMOS gates, but some custom high-speed circuits were needed. The floating-point multiply array uses dynamic domino circuits to accumulate the partial products. Other custom circuits are found in the simultaneous bidirectional pads as well as in the clock recovery circuitry, which adjusts data arriving from the network into the local clock domain.

Datapath components such as the 64-bit adders and the floating-point multiply array are full custom layouts and are treated as custom macrocells. Other datapath modules were built from a cell library that matched the datapath cell pitch. The cells were placed using the Cadence Smartpath floorplanning tool, and the routing was completed automatically using the Cell3 router. All of the control logic was synthesized from the Verilog register transfer level (RTL) model of the MAP chip, using the Cadence Synergy logic synthesis tool, and mapped to our control cell library. The cells were then assembled and connected using the Cell3 place and route tool. The top level wiring is a combination of manual and automated routes. In all, the MAP chip includes approximately 231,038 placed cells, including datapath, control, and custom macro cells.

Schematics were drawn using the Cadence Composer schematic entry tool, and layout was generated using the Virtuoso layout editor. Dracula was used to perform the design rule checks (DRC) and the layout versus schematic (LVS) comparisons. We constructed a timing model of the entire chip from the MAP's schematics and a library of timing models from individual layout cells. This model was annotated with the wiring parasitics from the top level layout. We then used a static timing analyzer to identify long paths and underdriven nodes. The final routing of the chip is complete and tapeout is scheduled for May 1998.

2.4 Evolution of the MAP Design

During the design and implementation of the MAP chip, we were forced to reduce the scope of the project in order to make all of the components fit on chip and to complete it in a timely fashion. The original MAP architecture was much more aggressive than what we were able to implement. It called for four clusters, each with a floating-point unit and an 8 KByte instruction cache, four 32 KByte cache banks, a 3-dimensional network, and larger event and message queues. As the components were assembled, we realized that most modules were larger than anticipated and that they would not all fit on the chip. In our first reduction, we eliminated one cluster, cut each cache bank in half, and reduced the network to two dimensions. In a subsequent chip amputation, we removed two floating-point units, two cache banks, and all of the control registers for one thread

Component	Area ($M\lambda^2$)	% of total area
Integer Units (4)	1192	10.0
Memory Units (4)	900	7.6
32KB Data Cache Banks (4)	2700	22.7
Floating–point Unit (4)	2136	18.0
NIF/3-D Router	644	5.6
I/O Pads	597	5.0
Instruction Caches (4)	755	6.3
EMI + 128 entry TLB	213	1.8
Clock drivers	182	1.5
Event Queue	180	1.5
GCFG	53	0.5
Switch drivers	100	0.8
Wiring	2288	18.7
Total	11940	

Table 2.4: Area costs for original MAP architecture.

slot on each cluster. This left us with three clusters, seven total execution units, one floating-point unit, and five thread slots.

We also made several design decisions intended to reduce the amount of engineering effort required for the project. Our choice to use a datapath cell methodology reduced the amount of custom layout required, but as a result, each module grew in size. The clock rate was another area in which we decided to reduce our engineering effort. All of the datapath logic is designed and verified to run at 100MHz. However, we chose not to optimize any of the critical paths in the control logic that emerged from our logic synthesis. Consequently, the control logic will run at 40MHz, according to our static timing analysis.

With a state-of-the-art process, many of the MAP's area constraints would be reduced or eliminated. Table 2.4 summarizes estimates for the area of the original MAP specification based on the three-cluster implementation. The resulting area required is approximately $11.9G\lambda^2$ or about twice the area of the actual MAP chip. To implement the original MAP in a chip of the same dimensions (18.25×18.3mm) requires a feature size of approximately $0.35\mu m$, which is surpassed already by today's $0.25\mu m$ chips. Additional area savings are possible by re-implementing the cell-based datapath modules. While full custom datapaths would have required more time to design, they would also be substantially smaller and faster. The speed of the chip could be dramatically increased with further optimization. Since the MAP is partitioned into clusters, all of the automatically generated wires are short, while the long wires, such as those in the Memory and Cluster Switches, could use low voltage swing circuits or could be pipelined if additional speed were required. The control logic has a small number of critical paths that could be hand designed to reduce the depth of logic and the number of gates on the paths.

2.5 Scalability of the MAP Architecture

By partitioning the on-chip execution units into independent processor clusters, the MAP architecture can be easily extended to larger chips by adding more clusters. However, while most of the wires are short, some global communication is still required. The wires that implement the Memory Switch, the Cluster Switch, global condition code registers, and the cluster barrier instruction all are global since they connect remote cluster or memory modules. Because the MAP chip allocates an entire clock cycle for signals to traverse any of these paths, the global wire delay is not a limiting factor for clock rate for the current design. However, with future process technologies and higher clock rates, any long wire will be a problem.

Figure 2.7 outlines MAP-2007, which is a projection of the MAP architecture targetted for a 0.1μm technology. The implementation statistics from Table 2.3 are used to estimate the area of the processor and memory components. With roughly 80 times the silicon area of the 0.5μm MAP chip, the scaled MAP chip may contain 80 processor clusters, each with its own 32KB level-1 cache for fast local memory access. Multiple processors would be clustered around a larger 256KB level-2 cache. This organization is a variant of the existing MAP implementation with the most notable difference being a private level-1 cache for each processor. Because of the influence of wire delay, sharing a level-1 cache among multiple processors could result in an 8 cycle access latency due to arbitration and switch traversal. Using copper interconnect and a low permittivity dielectric a signal can travel only about 2.5mm in one 2GHz clock cycle. Widening and thickening the wires beyond minimal width would further reduce wire resistance and help speed up global communication. However it is clear that the global wires designed for the 3 cluster MAP chip are not appropriate for the 80-cluster MAP chip.

Aside from local memory organization, the primary influence of global wire delay is in interpro-

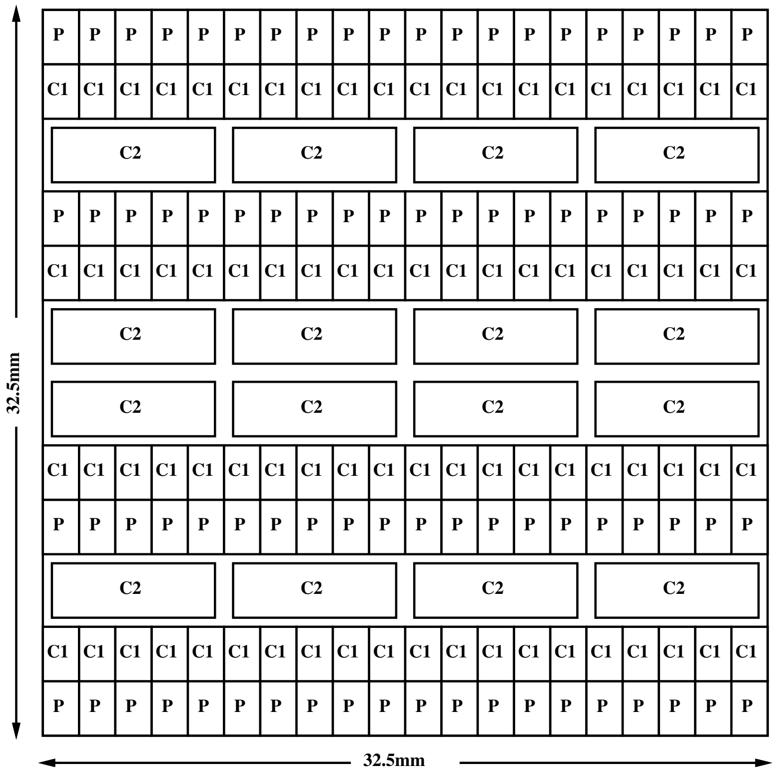


Figure 2.7: Modified MAP architecture scaled for 0.1μm CMOS technology. The chip might contain 80 processor clusters (P), each with it own 32KB level-1 cache (C1). Five processors are clustered around a 256KB level-2 cache (C2), and global cache coherence could be maintained using a hardware protocol.

cessor communication. A single global Cluster Switch would require at least 26 cycles to transmit a word from processors at remote corners of the chip. Furthermore using one global bus per cluster to implement the Cluster Switch would require 6400 global wires and unreasonably complicated arbitration logic. Even if wire resistance was miniscule, the wire bandwidth required for this architecture would be prohibitively expensive. To implement direct communication between processor clusters, the Cluster Switch can be replaced by a hierarchical network in which a small number of processor clusters would share the equivalent of a local Cluster Switch.

Communication between processors in different groups would go through additional levels of switch hierarchy. Alternatively, the processor groups can be connected in a mesh network to their nearest neighbors. Remote writes would travel over or through other processor groups to get to their destination. Communication between processors can take place at wire speeds, namely four 2GHz clock cycles for 5 local processors. Communication between remote processors would depend on a combination of wire and switch delay. Communication between processors using the global condition code registers would require similar scalable communication implementations and could perhaps be incorporated into the data communication network. To encode enough identifiers to capture 80 processors would require a total of 7 bits in each arithmetic instruction format. Increasing the MAP's instruction encoding to 48 bits would provide ample space for remote processor identifiers as well as for 64 registers per register file.

The cluster barrier instruction (`cbar`) also requires global wiring to connect all of the processor clusters that are synchronizing. A natural extension to the `cbar` instruction that is well matched to the groupings described above is to allow synchronization across a subset of the processors. A barrier across an entire 5-processor group would require four 2GHz clock cycles. Barriers across multiple groups can take place in hardware via a hierarchical synchronization network. The latency of the barrier depends on how many processors are involved and how far away they are. A full 80-processor barrier will still require at least 26 cycles of wire delay to accumulate and distribute the barrier requests. Barriers across subsets of processors can be encoded directly into the instruction set using bitmasks. Since all 80 processors can not be named individually in a limited size bit field, barriers on subsets of processors can be encoded on a processor group basis. A processor could then synchronize with individual processors within its own group or with remote processor groups as a whole.

The least scalable component of the current MAP architecture is the centralized memory system. If the processors are located at one end of the chip and the memory is at the other, then each memory reference can take as many as 26 cycles to resolve, just from wire delay. The scaled Multi-ALU processor in Figure 2.7 solves this initial problem by allocating a small cache to each processor. This eliminates switch traversals between processors and memory and can reduce the minimum memory access time from 3 cycles in the current MAP architecture to 1 cycle. A subsequent complication is associated with data sharing among different processors. In the MAP chip, each on-chip processor sees exactly the same memory system hierarchy. The scaled MAP architecture would require hardware support to keep the caches coherent. This is relatively easy to accomplish among a processor group with a shared level-2 cache. However, global data sharing is much more complicated and may ultimately be too expensive to implement. With the high degrees of concurrency that will be available, a better memory system organization may eliminate cache hierarchies and instead use high density DRAM for the primary storage at each processor. Data communication would then be explicit between processors using register-register or register-memory transfers. The changing balance between on-chip computation and communication latencies will present a rich set of challenges for programming systems and software support for scheduling and resource management.

In order to take advantage of the increasing numbers of transistors made possible by shrinking device sizes, scalable mechanisms must be used to increase the on-chip computation power and memory capacity. Unfortunately, current architectures are distinctly not scalable. Superscalar microprocessors rely on centralized instruction issue logic to control its execution units. In the example 0.1μm process, an instruction would require 6.5ns just in wire delay to travel from an instruction window in the center of the chip to an execution unit located at a corner. If the chip is to run at 2GHz, the delay is equivalent to 13 clock cycles, which would require extremely deep pipelines in order to run at full speed. With such deep pipelines, any mispredicted branch or jump will result in a substantial penalty due to flushing the pipeline of useless instructions that have already been fetched. An equally unattractive alternative to pipelining is to reduce the clock rate to less than 200MHz to allow more time for the signals to propagate. One final issue with today's ILP architectures is that the notion of massive scaling of superscalar processors requires a dubious assumption of an abundance of parallelism in primarily sequential applications.

Using the silicon to build a single-chip multiprocessor with large shared caches has its own set of problems. Large monolithic cache structures will be unacceptably slow due to long propagation delays in the word and bit lines. Furthermore, as will be described further in Chapter 4, large wire latencies and memory access delays between the execution units and storage at the other end of the cache hierarchy will render memory communication useless for fine-grain parallelism. The instruction overhead associated with synchronizing through memory also contributes to the overall performance cost of memory only mechanisms. Partitioning chips into independent processors with local memory is necessary in order to scale the number of on-chip execution units and still maintain high clock rates. The direct communication and synchronization mechanisms will then allow the multiple processors to work together efficiently and effectively.

2.6 Lessons from the Implementation

In implementing the MAP chip, we faced a very different set of challenges from those in an industry product development group. As a result, we have learned a lot of lessons, both technical and in project management, about tackling a project of this magnitude in an academic environment.

Our early architectural studies included estimates of the chip area required to implement each of the major modules of the MAP. When the chip assembly began, we realized that our estimates were overly optimistic, and we had to eliminate an entire cluster and reduce the sizes of the on-chip memories. Some of the inaccuracy in our estimates resulted from choosing a cell-based design methodology in the datapath regions, rather than full-custom. While this decision was necessary to reduce design time, the datapaths in the semicustom regions grew by up to 60% due to the increased cell area and wiring requirement [Cha98]. In addition, our estimate for control logic cell utilization rates was optimistic, and the additional latches and multiplexors to implement multithreading required more wiring tracks than predicted. As a result, the control logic regions are approximately twice as large as anticipated.

Wiring also constituted a larger than estimated fraction of the area in the control logic regions. The cell–area utilization in different control logic regions varied unexpectedly. We found that the architecture greatly affected the estimation accuracy, and that a single cell density metric is not sufficient for all control logic regions. In our arithmetic control regions, 55% of the area is occupied by logic cells, with the remaining area dedicated to interconnect. In the pipeline control, which replicates the pipeline registers to implement multithreading, the logic could only occupy 40% of the area and still be routable.

Another key lesson is that the design effort needs to be balanced on all aspects of the chip. Our early datapath pitch selection of 14 wire tracks per bit was exactly the correct number required by the physical implementation. This allowed us to complete the datapath logic design and begin the circuit design at a very early stage in the project. Most of the datapaths, such as the register files and arithmetic blocks were straightforward, but additional circuit design was required for the the data cache and the floating-point unit. In order to support the memory synchronization bits, the data cache needs to implement a read-modify-write cycle. This required that the data cache be clocked by a special delay line in order to finish the write at the beginning of the subsequent cycle. The floating-point multiply array required a self-timed domino design and aggressive time borrowing in order to complete all of the partial product summations. As a result of these circuit optimizations, the datapath modules all met the 100MHz clock rate target.

The full-chip logic design was not completed until much later and, in order to meet our deadlines, we focused primarily on logic validation. In retrospect, it would have been beneficial to spend less time optimizing the datapaths, and instead complete the control logic sooner. Because of the greater complexity of the control logic and the need to use automated design tools to generate the circuits, a clock rate of 100MHz was much more difficult to reach. According to our static timing analysis, the worst case path is in the exponent calculation of the floating-point divide/square-root unit, which is 15ns over budget and limits the clock rate to 40MHz. This path and the others that restrict the clock rate could be shortened at the expense of a substantial amount of design time. In retrospect, a better balance between the time spent optimizing the datapath and control circuits could have resulted in a higher overall clock rate. However, it must be noted that a functional prototype is far more important than a fast clock rate for an academic project such as this.

One of the most important factors in the success of the implementation was our partnership with Cadence Spectrum Design. In addition to taking responsibility for the physical design, layout, and chip assembly, they provided a substantial amount of technical feedback and expertise. Their review of our cell density metrics and their place and route experiments allowed us to easily identify which MAP modules to eliminate in order for the remaining modules to fit on chip. In addition, Cadence's experience with the particular fabrication process was vital in the design and implementation of the clock and power distribution, as well as in the chip assembly process. Finally, the MAP chip's size and large number of transistors pushed the limit of the verification tools. In order to perform layout verification, Cadence developed new methodologies and used some new tools. In short, our industrial partnership allowed our team at MIT to focus more heavily on the novel architecture mechanisms, while relying on the skills of Cadence to perform the physical design.

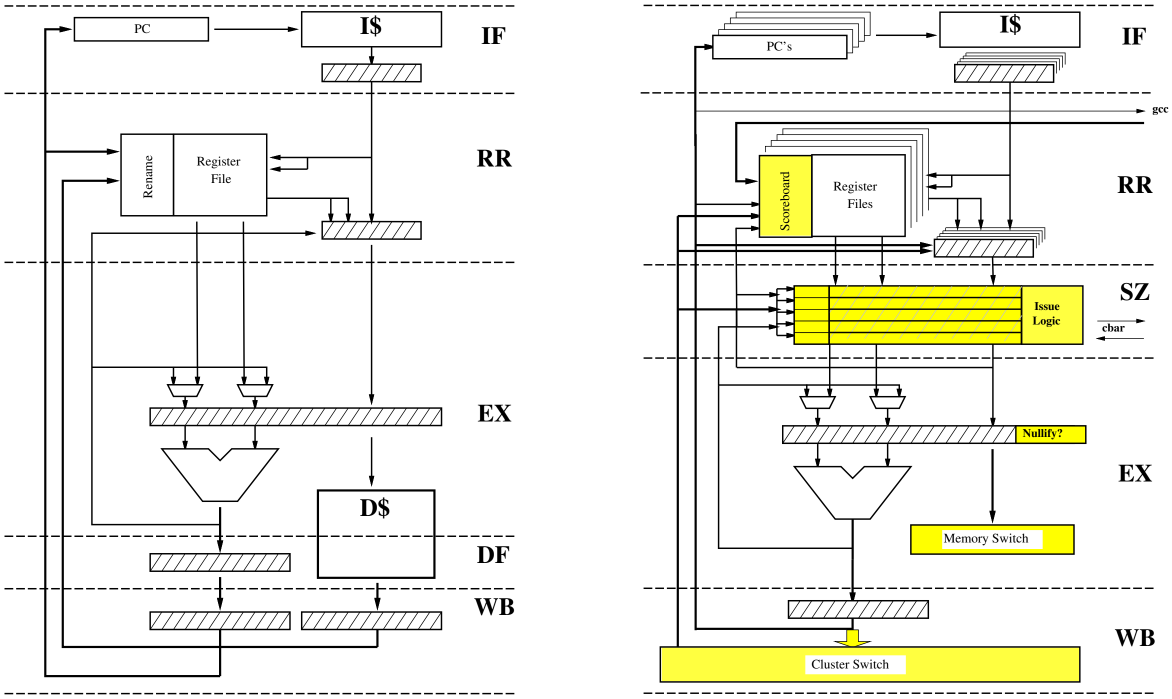
Chapter 3

MAP Chip Pipeline Design

The processor pipeline of the MAP chip is designed to exploit parallelism within each MAP cluster and to enable fine-grain parallelism between clusters. The pipeline includes mechanisms to execute multiple instructions concurrently and to interleave instructions from multiple threads. In addition, the intercluster communication and synchronization components are integrated tightly into a cluster's pipeline.

Figure 3.1 highlights the differences between a simple reduced instruction set (RISC) pipeline and the MAP pipeline. The example RISC pipeline includes five stages: Instruction Fetch (IF), Register Read (RR), Execute (EX), Data Fetch (DF), and Write Back (WB). A program counter accesses the instruction cache in the IF stage, and passes the next instruction to the RR stage where registers are renamed and the register file is read. The subsequent EX stage executes the instruction and, for loads and stores, begins the data cache access. In the DF stage, the cache access completes and the result is sent to the WB stage where it is written back into the register file.

The MAP pipeline is a variant of the basic RISC pipeline with several novel features to support parallelism within a cluster and between clusters. First, the pipeline and data registers in the first three stages are replicated to enable multiple threads to be loaded simultaneously in a cluster. A new synchronization pipeline stage (SZ) interleaves the instruction streams of up to six threads over the execution units, and can switch threads on a cycle-by-cycle basis with no switching overhead. The SZ stage includes reservation stations where the instructions from multiple threads wait until all of their operands are available. Thread scheduling can be controlled in software through a



Basic RISC Pipeline

MAP Cluster Pipeline

Figure 3.1: Comparison of basic RISC and MAP pipelines.

combination of programmable thread priorities and mechanisms to prevent thread starvation. A novel instruction fetch unit supplies instructions for all of the threads and expands instructions on the fly from a dense encoding that has the NOPs compressed into the instruction format used within the pipeline.

Intercluster communication is integrated into the pipeline by enabling the arithmetic units to write directly to the Cluster Switch during the Write Back (WB) stage. In addition, single bit condition code values may be broadcast to other clusters using the global condition code (GCC) registers. The pipeline uses a scoreboard in the register read stage to track data dependencies and enable data synchronization on register writes between clusters. The scoreboard can be manipulated by a new `empty` instruction which marks a register invalid pending arrival of remote data. Control synchronization across the clusters is implemented with a cluster barrier instruction. Finally, the MAP employs a shared lockup-free cache in which memory requests are decoupled from the processor pipeline, eliminating the data fetch stage from the processor pipeline. Load and store

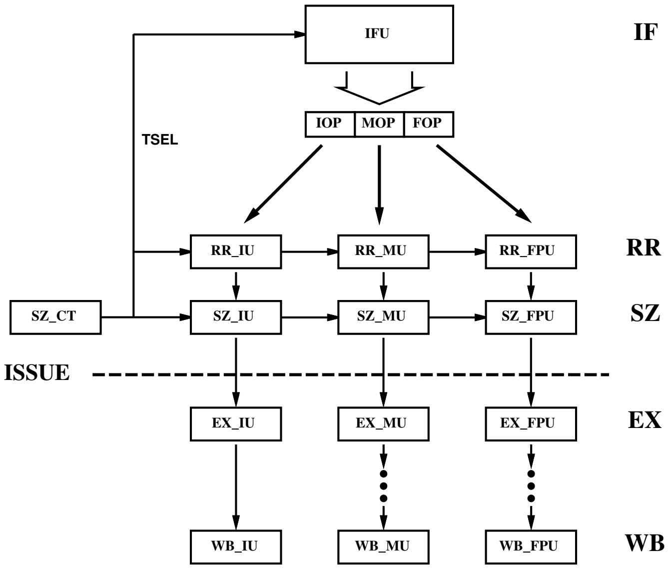


Figure 3.2: Block diagram of all cluster pipeline modules.

operations are sent to the unified cache via the Memory Switch, and the scoreboard ensures that subsequent instructions that need the result do not issue until the data is returned.

This chapter describes the MAP pipeline and the novel mechanisms that enable parallelism both within a cluster and between clusters. Section 3.1 details the design and implementation of each pipeline stage. The use of the scoreboard to enforce data synchronization between instructions is discussed in Section 3.2. Section 3.3 characterizes the impact of multithreading including the additional interaction between pipeline stages and the flexibility of the thread selection logic. Finally, Section 3.4 describes the integration of mechanisms for interthread interaction into the MAP pipeline.

3.1 Pipeline Components

Figure 3.2 shows the pipeline stages for all of a cluster's execution units. A single instruction fetch unit (IFU) delivers operations to each of the integer, memory, and floating-point pipelines. Each execution unit has its own register read (RR), synchronization (SZ), execute (EX), and write back (WB) stages. A central synchronization stage controller (SZ_CT) determines which instruction to

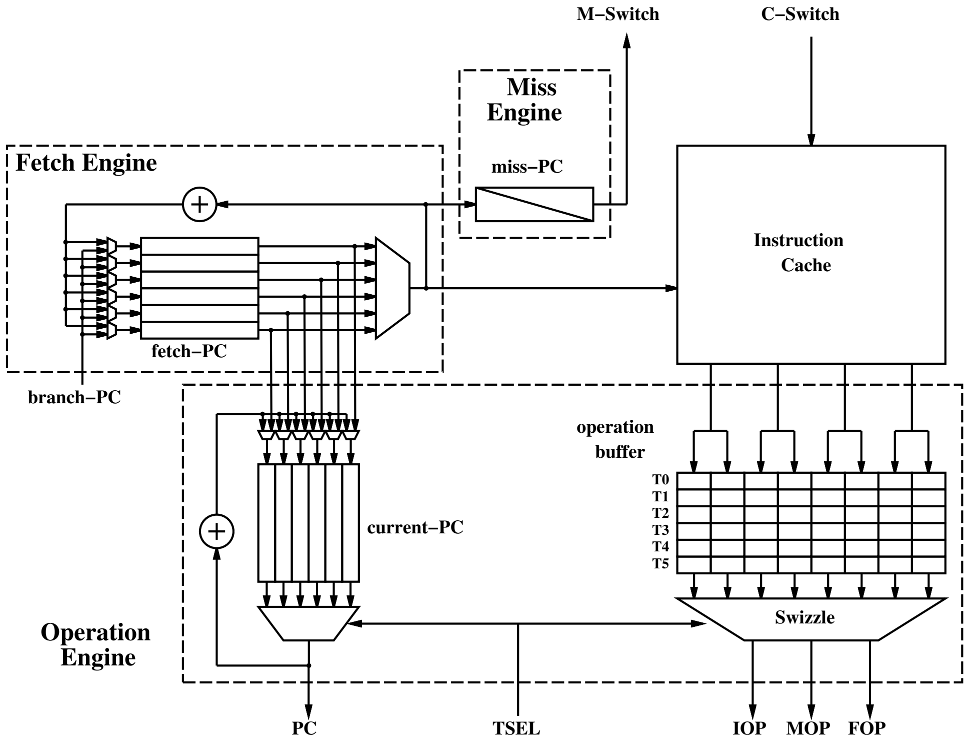


Figure 3.3: The MAP Instruction Fetch Unit.

---

issue from the SZ stage to the EX stage. The SZ_CT logic delivers the thread selection decision on the TSEL (thread select) wires to the IFU and to all of the execution pipelines.

3.1.1 Instruction Fetch

The instruction fetch unit (IFU) provides instructions from the cluster's instruction cache to the pipeline and refills the instruction cache from the unified cache when a miss occurs. As shown in Figure 3.3, each thread has its own program counter (current-PC) which points to the next instruction to leave the IFU and enter the RR stage. Each thread also has a fetch program counter (fetch-PC) which is used to optimistically fetch instructions from the instruction cache into the operation buffer. The thread select (TSEL) signal determines which thread will deliver the next program counter and instruction. Since all threads reside in the same global virtual address space, the instruction cache can be shared among them.

	Pack Bits	Meaning
First operation	00	IU operation only
	01	MU operation only
	10	FPU operation only
	11	Multi-operation instruction
Second operation	00	IU-MU instruction
	01	IU-FPU instruction
	10	IU-MU-FPU instruction
	11	MU-FPU instruction

Table 3.1: Instruction pack bits to compress NOPs from instruction stream. If the instruction contains two or three operations, the pack bits from the second operation are required.

The IFU consists of three state machines that are largely independent: the fetch engine, the operation engine, and the miss engine. The fetch engine contains a bank of fetch program counters (Fetch-PC), one for each thread. On each cycle, the fetch engine selects a PC, uses it to fetch four operations (128 bits) from the cache, and places the operations in the operation buffer. The selected PC is then incremented and returned to the fetch-PC bank. In order to prevent a thread from stalling due to instruction fetch, the fetch engine tries to keep the operation buffer full at all times.

The operation engine contains a bank of current program counters (one for each thread) and the operation buffer, which is a queue of eight 30-bit operations for each thread. The MAP instruction format provides a dense encoding to eliminate NOPs in the instruction stream when not all three units are needed by a particular instruction. Table 3.1 shows how two *pack* bits from each 32-bit operation are used to encode the length of an instruction (one, two, or three operations), and the execution units that are included. If the instruction has only a single operation, the pack bits from the operation determine whether it belongs to the integer, memory, or floating-point unit. If the instruction has more than one operation, then the pack bits from the second operation are necessary to complete the encoding. NOP operations are inserted automatically when the instruction leaves the instruction fetch unit. In response to TSEL, the operation engine retrieves the next program counter and the next three-operation instruction for the selected thread and delivers them to the register read pipeline stages. The pack bits are decoded and the instruction swizzler directs each operation from the operation buffer to the correct execution pipeline. The program counter is then incremented by the number of operations that were included in the instruction.

When a cache miss occurs, the miss engine captures the miss address in one of two miss-PC registers and initiates an instruction cache refill. The refill engine sends load requests to the global mixed cache to fetch subsequent instructions in the program. A refill consists of eight fetches of 64-bit words, and each word is requested by a memory operation similar to a load. Each request reaches the memory system via the Memory Switch and the resulting data is returned via the Cluster Switch. Control bits in the request and reply packet return the result to the instruction cache, rather than to the register file. Since each operation is 32-bits long, an instruction cache refill cycle loads 16 operations into the IFU. Two miss handlers are implemented, with one dedicated to the TLB miss thread and the other shared among the remaining five thread slots. A dedicated miss handler for the TLB thread is necessary to prevent deadlock when an instruction cache miss causes a TLB miss. An instruction cache miss that occurs while the miss handler is already busy is delayed until the miss handler becomes free again.

The interface between the instruction fetch unit and the rest of the pipeline includes thread select (TSEL), instruction-fetch available (`if_avail`), and the branch program counter (`branch_PC`). The `if_avail` signals indicates which threads have four operations in the operation buffer. If `if_avail` for a thread is deasserted, then the synchronization stage will not advance that thread since there may not be enough operations ready to enter the register read stages. When a branch instruction is executed, the target instruction pointer is delivered to the IFU, where it is immediately loaded into the fetch-PC. On the next TSEL, the operation buffer for that thread is cleared, an instruction cache fetch is immediately initiated, and the new instruction pointer is loaded from the fetch-PC into the current-PC.

3.1.2 Register Read

The register read (RR) stage contains register files and pipeline registers for each thread. The integer register file is accessed by the integer and memory pipelines, while the floating-point register file is read by the memory pipeline and the floating-point pipeline. In addition to the 64-bit data registers, the RR stage also holds the condition code (CC) registers, which are used for conditional branches or predicated execution. The CC registers are divided into local and global; four CC registers are purely local and may only be written by the cluster in which they reside. The remaining

12 CC registers are global, with four assigned to each cluster. A cluster may only write to one of its four assigned global CC registers, and that update is broadcast to the corresponding global CC register in every cluster.

After accessing the register files, incoming instructions wait in the RR stage pipeline registers before advancing to the SZ stage. The thread select (TSEL) signal from the synchronization stage controls the RR stage. When TSEL is asserted, the RR stage outputs the designated thread's next instruction and latches the subsequent instruction for the same thread coming in from the instruction fetch unit. Instructions waiting in the RR stage have results bypassed directly to them from local write-back stages as well as from memory or remote clusters via the Cluster Switch. The register files are only accessed once per instruction.

The RR stage also contains the register scoreboard, which has one full/empty scoreboard bit for each register. The scoreboard indicates whether the data in the register is valid and can be used as an instruction operand. Operations mark their destination registers empty upon issue and full upon completion. When an `empty` instruction executes, the bit vector that identifies which registers to empty is sent to the RR stage which then modifies the scoreboard.

3.1.3 Synchronization

The synchronization (SZ) pipeline stage is the central control unit of the cluster and contains both instruction reservation stations [Tom67] and instruction issue logic. Each of the execution pipelines has its own reservation station, which holds one operation for every thread. Operations wait in the reservation station until their instruction is selected by the issue logic in the synchronization control module (SZ_CT). Both data and register validations are bypassed directly into the reservation stations. As with the register file, the reservation stations can be written from the local execution units as well as from the Cluster Switch. The SZ_CT module communicates with the reservation stations in the integer, memory, and floating-point pipelines to determine which thread's instruction is ready to issue. When an instruction issues, the instruction fetch and register read stages for the selected thread advance and a new instruction enters the SZ stage. Once a thread's instruction issues, it will complete without any further intervention from the scheduler in the synchronization stage. As shown in Figure 3.4, thre process of selecting a threads includes checking for operands, masking out low priority threads, and arbitrating among the remaing contenders.

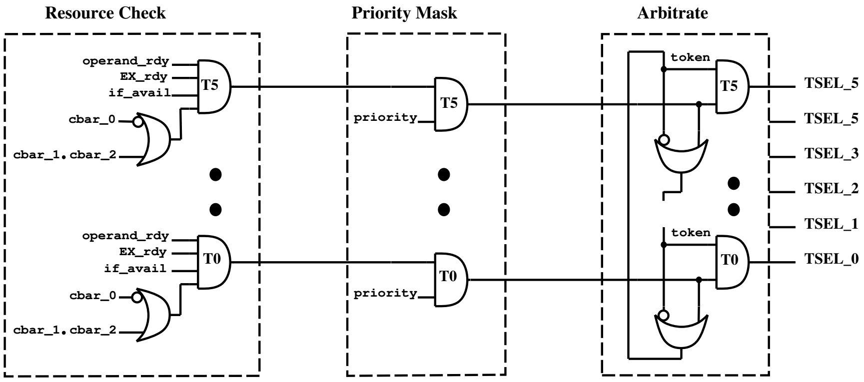


Figure 3.4: MAP synchronization stage.

Resource Check: The Resource Check module examines all of the operations in the synchronization stage reservation stations and determines which threads are ready to issue. In order for an instruction to be eligible to issue, it must satisfy the following conditions:

- All of the instruction's operands are present (`operand_rdy`).
- All of the arithmetic units needed by the instruction, such as the floating-point divide/square-root unit, are available, and none of the execution units are stalled waiting for the Memory or Cluster Switches (`EX_rdy`).
- The instruction fetch unit is able to deliver at least three operations from the operation buffer to the register read stage for the chosen thread (`if_avail`).
- If the instruction contains a cluster barrier, the other clusters have also reached the barrier.

Priority Mask: The priority mask module examines each thread that is ready to issue and gives preference to those threads that have a higher priority. The thread slots are divided into two categories, system and user, and each thread has two priority levels. The system threads include the event and exception handlers running in slots 3, 4, and 5, while the user threads run in slots 0, 1, and 2. A simple three state priority scheme is used to determine which threads are the most important. High priority system threads are the most important and can monopolize the execution resources when necessary. Low priority system threads and high priority user threads are at the same level and have equal access to the execution units. At the bottom of the hierarchy are low priority user threads. The two user levels enable applications to run their critical sections at a higher priority, which can increase overall performance [FD95]. Section 3.3.2 describes other features of the SZ_CT that ensure a thread is not starved of issue slots indefinitely.

Arbitration: Placed after the priority mask, the arbiter considers only those threads that are both ready to issue and have priority. The arbiter uses round robin scheduling to allocate the cluster execution resources fairly among the threads. A token is inserted into the arbiter at the stage whose thread was last granted. The token flows between arbitration stages until it reaches a requesting thread. The arbiter's decision is broadcast to the rest of the cluster in the thread select (`TSEL`) signals. Performance monitoring counters are built into the SZ_CT module to track when a thread is selected and when a thread has all of its operands ready but is not selected.

3.1.4 Execution Units

The Execute (EX) stage contains three execution pipelines, including an integer unit (IU), a memory unit (MU), and a floating-point unit (FPU). The integer unit executes all of its arithmetic instructions in a single cycle, but the memory and floating-point units encompass multiple pipeline stages. The arithmetic units receive instructions and operands from the SZ stages, execute the instructions, and forward the results to the write-back stage. Data from previous instructions is bypassed directly to the front of each execution unit. Unlike the first three pipeline stages, the EX stage pipeline registers are shared by all of the threads. Arbitration for the Cluster Switch is performed during the EX stage, one cycle before the result is ready. If the Cluster Switch is not granted, the entire cluster stalls and arbitration is performed again on the next cycle. Since the IU and FPU share a port to the Cluster Switch, a simple local arbiter determines which unit will be granted access. Exceptions are detected at the beginning of the EX stage; if an exception occurs, the pipeline stalls while a state machine in the hardware updates the registers in the exception thread slot. Figure 3.5 details the arithmetic units contained within the EX stage. An abbreviated description of the MAP's instruction set architecture can be found in Appendix A.

Integer Unit: The integer unit (IU) contains a 64-bit adder, a barrel shifter, and a boolean logic unit. In addition to executing integer arithmetic instructions, the IU performs byte insertion

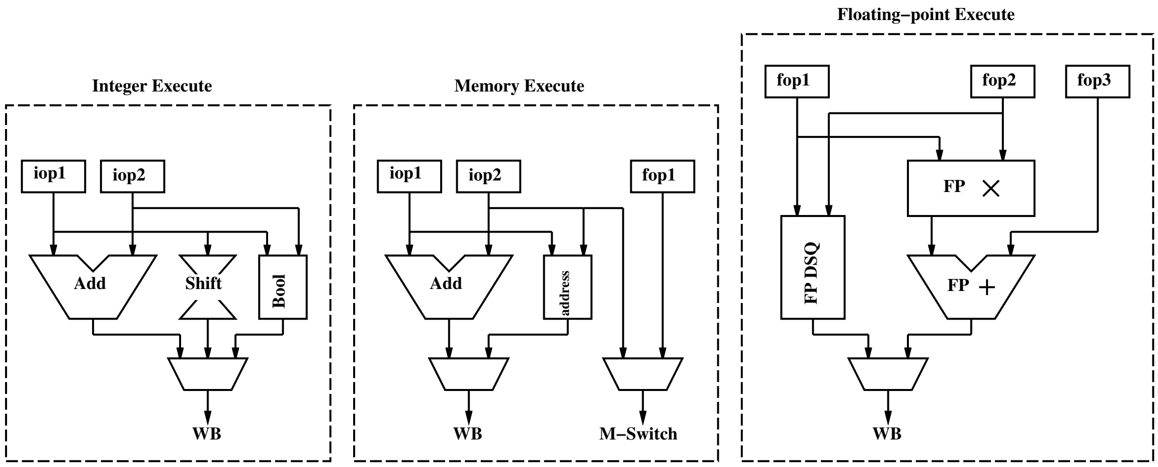


Figure 3.5: MAP execution units.

into and extraction out of a 64-bit word. The IU also has a mask unit to detect segmentation violations that occur during address calculation. Although not shown in the diagram, the IU includes a send unit to transmit a message that has been composed in the integer register file to the network interface for delivery to a remote MAP chip. The send state machine sequences the message contents from the registers to the network interface unit via the Cluster Switch.

Memory Unit: The memory unit (MU) includes a 64-bit adder, an address calculation unit, and a path to the Memory Switch. The MU can execute a subset of the integer arithmetic instructions, but its primary function is to issue loads and stores. Since indexed and displacement addressing are not supported in the MAP chip, the address for the memory operation is known immediately when the instruction enters the MU. This allows the MU to arbitrate for a Memory Switch path to the correct cache bank during first half of the MU execute cycle. If the switch is granted, the address (and data for a store) is transferred on the second half cycle. If the switch is not granted, the pipeline blocks. The MU has two sets of pipeline registers, one for *normal* requests, and one for *TLB* requests. The TLB handler can continue executing even if all of the other threads are blocked. Load and store operations can perform address calculations concurrently with the Memory Switch access. A post-increment value is added to the address using the adder in the MU, and the result is written back the local register file.

Floating-point Unit: The floating-point unit (FPU) includes a multiply-add unit (MULA) and a divide/square-root unit which both adhere to the IEEE floating-point format [IEE85]. The MULA unit executes floating-point add, subtract, and multiply instructions, as well as integer multiply. The four-stage pipeline consists of a multiplier followed by an adder to implement a fused multiply-add ($A \times B + C$). Multiplies and multiply-adds need all four pipeline stages, but other instructions, such as floating-point add, only use the last two stages. If the upper pipeline stages are empty, a low latency operation may skip them and drop directly into the stage it needs. However, the MULA pipe will not reorder instructions. The multiplier uses a radix-8 booth encoder and two-branch Wallace tree to accumulate the partial products. The carry-save adders in the multiply array are implemented in domino-logic, while the surrounding datapath logic is in static CMOS. A more complete description of the circuit implementation of the MULA unit can be found in [Har96].

The divide/square-root (FDSQ) unit implements floating-point divide, floating-point square-root, and integer divide. The FDSQ unit is not pipelined, but uses a radix-4 iterative SRT algorithm. The divide and the square-root functions share much of the same datapath logic, but use different lookup tables to determine the next quotient bits [Fan87, EL90]. Each iteration requires half a clock cycle; latches and multiplexors are implemented to allow the iterative logic to be used during both halves of a clock cycle. The floating-point divide/square-root instructions have a 20 cycle latency, while integer divide requires 23 cycles. When the FDSQ unit is busy, the SZ stage is prevented from issuing any divide or square-root instructions. Since the MULA and FDSQ units share the same write-back register, the FDSQ stalls the MULA pipeline when a divide or square-root instruction completes.

3.1.5 Write Back

During the write-back stage (WB), data is written into the local register files and bypassed into the RR, SZ, and EX stages. For a remote register write, the data is transmitted on the Cluster Switch during the WB stage and can be used by the remote cluster on the subsequent cycle. The Cluster Switch is also used to transfer data between the integer and floating-point register files. A dedicated path between the integer and floating-point datapaths was purposely omitted from the design in order to reduce the wiring complexity within the cluster.

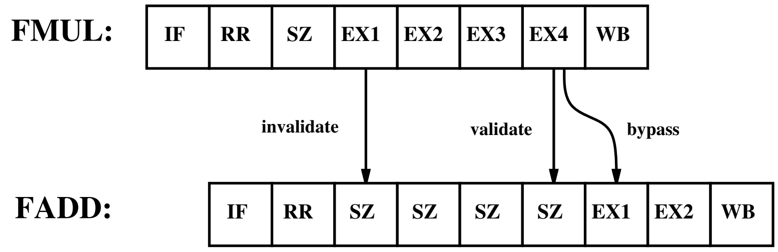


Figure 3.6: Data synchronization and delivery in the MAP pipeline.

3.2 Data Synchronization

The MAP pipeline enforces all data dependencies between instructions using register scoreboarding for pipeline interlocks and bypassing for data delivery. Figure 3.6 shows a pipeline timing diagram for a four cycle floating-point multiply (`fmul`), followed by a floating point add (`fadd`) that consumes the result. When the `fmul` issues from the SZ stage to the EX stage, it empties its destination register in the scoreboard and invalidates the `fadd` in the SZ stage. One cycle before the result is available (during the last EX stage), the register is validated in the scoreboard and in the SZ stage, allowing the `fadd` to issue and meet the result of the `fmul` at the beginning of the EX stage. For single cycle latency instructions, the destination is validated in the EX stage, but it is not invalidated. A `load` instruction also marks its destination register empty upon issue, and the register is marked full when the data is returned on the Cluster Switch.

In addition to data synchronization, the MAP chip provides support for detecting when memory operations have completed. Because the memory system is heavily pipelined, a program may not be able to determine when a store instruction has written its data to memory. Furthermore, a protected procedure call must be able to prevent the code that called it from leaving an outstanding load that may corrupt the register file when it returns. To enforce memory ordering and detect completion of memory operations, the MAP implements a memory barrier (`mbar`) instruction which stalls until all outstanding memory references for a thread have finished. When a load or store instruction is issued, a thread's `mbar` counter is incremented; when data returns on the Cluster Switch, the counter is decremented. Even though store operations do not return any data, they do include a Cluster Switch transaction to decrement the counter. When an `mbar` instruction enters the SZ stage, it stalls until the `mbar` counter is zero, meaning that all outstanding loads and store have completed. Up to 31 memory references for each thread may be outstanding. If the maximum number has been reached, subsequent load and store operations must wait in the SZ stage.

3.3 Multithreading

Multithreading has typically been exploited in multiprocessors to tolerate memory latencies. In block multithreading, as found in the Alewife machine [ALKK90], a thread switch is typically triggered by a cache miss or a write-hit to shared data. Fine grain multithreading, which is employed in the Tera machine [ACC+90], typically requires a thread switch on every cycle. The MAP chip implements fine-grain interleaving of instructions based on data availability, rather than a strict round robin scheduling scheme. Thus, the MAP chip can tolerate both long memory latencies and short instruction latencies, while still executing at full capacity even if only one thread is present. The term *zero-cycle multithreading* indicates that no dynamic overhead is incurred to switch among threads when programs are executing in the pipeline.

3.3.1 Pipeline Overhead

In order to implement zero-cycle multithreading, the register files and pipeline registers in the IF, RR, and SZ stages are replicated. In the MAP chip, the additional state in both the datapath and control logic increases the size of the cluster by 80% over a similar pipeline without multithreading. A substantial amount of this area is in the control logic regions, where more gates and wiring are needed. In addition to the state registers, multiplexors and demultiplexors are required at each interface to the RR and SZ stages so that the instruction information can pass from stage to stage on a common set of wires. Multithreading also requires more communication between pipeline stages as the thread select signal (TSEL) must be broadcast to the IF, RR, and SZ pipeline stages.

3.3.2 Thread Selection

The MAP chip uses a three-level priority scheme to enable important threads to use more of the execution resources. However, a low priority thread may be starved by higher priority threads. The synchronization stage implements a preempt state to guarantee that a thread cannot be starved indefinitely. Each thread has a preempt state machine which includes a counter and a limit register. The counter is incremented for every cycle that the thread is ready to issue but is not granted the pipeline. If the thread issues, the counter is reset to zero. When the counter reaches the value in the limit register, the thread enters the preempt state and elevates to the highest priority. This essentially guarantees each thread at least one issue slot every N cycles, where N is the value in the limit register. In the MAP, the counter and limit register are 8 bits each, so a ready thread will stall for at most 255 cycles.

In addition, a second set of counters keeps track of the number of cycles that a thread idles while waiting for its data to become available. This *stall cycle counter* is incremented on every cycle that any operand is invalid. It is reset to zero when all of the operands become available. A system software scheduler can use the counter to monitor the threads and determine if they are making progress. The thread priority levels, preempt limit registers, and stall cycle counters can all be modified through the configuration space interface, which allows applications and operating systems to efficiently control hardware thread scheduling.

3.4 Pipeline Mechanisms for Intercluster Interaction

The MAP chip implements mechanisms that enable fast communication and synchronization between clusters. Arithmetic units on one cluster can write directly into a register file of another cluster. The data is also delivered to the remote SZ and EX stages using bypass paths so that a remote arithmetic unit can use the data immediately. A cluster barrier `cbar` instruction can synchronize the pipelines in all of the clusters. Because these mechanisms are integrated tightly into the arithmetic pipelines, they can be used with very little overhead.

3.4.1 Register Synchronization

When a value is sent to a remote cluster, the destination must be notified of its arrival. The MAP chip incorporates this synchronization into a register scoreboard that is already needed for data synchronization within a cluster. When data arrives from a remote cluster, its destination register is marked full. However, unlike local arithmetic and memory operations, the destination register is not automatically emptied when the remote register write instruction issues. Instead, an `empty` instruction must execute at the destination cluster prior to data arrival. The `empty` takes

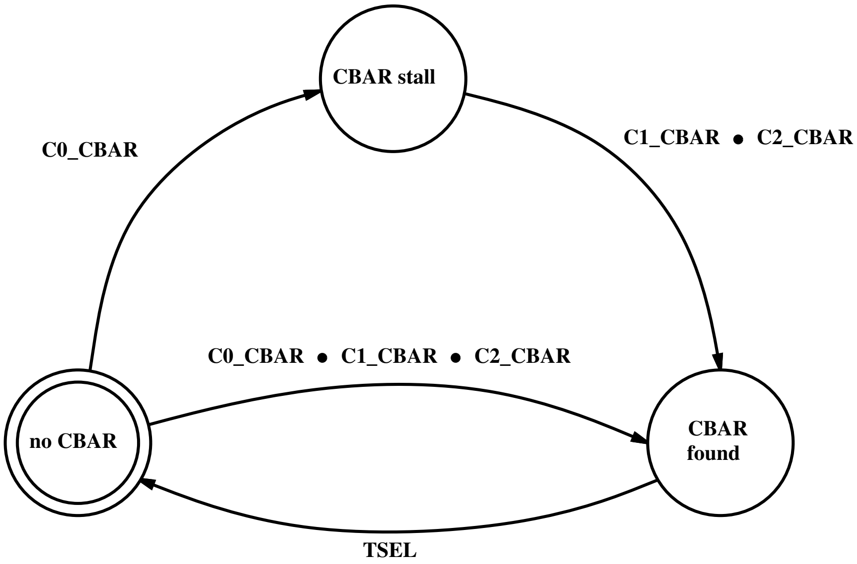


Figure 3.7: Cluster barrier state machine.

a bit vector as an argument which allows the remote cluster to manually invalidate one or several registers. An instruction at the destination that needs the data from the source cluster will stall until the data arrives. The `empty` instruction eliminates the need for any global wires for remote invalidation. Even if instructions could invalidate remote registers, an additional synchronization between the consumer and producer clusters would still be necessary to prevent the producer from delivering the data prematurely.

3.4.2 Cluster Barrier

Synchronization between threads on different clusters is supported in the MAP chip using a cluster barrier (`cbar`) instruction. The `cbar` acts as a gate such that no thread can proceed past the barrier until all threads have reached it. The state machine diagram used to implement `cbar` is shown in Figure 3.7. When a `cbar` enters the SZ stage, the state machine transitions to the *CBAR stall* state which causes the thread to wait. When both of the other clusters reach their barrier instructions, the state machine transitions to the *CBAR found* state in which the thread is again enabled to issue. When the thread is selected (`TSEL`), the state machine returns to the *no CBAR* state. If all three threads reach a barrier at the same time, the *CBAR stall* state can be skipped, and they can all issue immediately. The memory execute unit converts the the `cbar` into a NOP after it issues. Three states and six global wires are required to prevent back to back `cbar` instructions from becoming misaligned across the clusters. Each thread has its own state machine and global barrier wires so that cluster barriers and threads are independent.

3.5 Summary

The MAP chip pipeline employs novel features to implement zero-cycle multithreading and fast intercluster synchronization. A synchronization (SZ) pipeline stage orchestrates instruction execution across all three arithmetic units within a cluster. Instructions from each thread wait in the reservation stations of the SZ stage until all operands are present. The SZ stage examines the instructions from each thread and selects one to issue based on data availability, thread priority, and arbitration. The MAP also uses valid bits in a register scoreboard and in pipeline registers to unify nearly all instruction synchronization through data dependence, eliminating additional pipeline interlocks. The scoreboard tracks the data from local arithmetic operations, memory operations, and remote register writes. Intercluster synchronization takes place through data transfer or by using a cluster barrier instruction which is implemented in the SZ pipeline stage. The one drawback of the SZ stage is that it is an additional pipeline stage between the instruction fetch and execute stages, which leads to a larger penalty when a branch is mispredicted.

The architecture and implementation of a cluster are driven mainly by wiring constraints. The SZ stage is a central cluster resource and must communicate with most of the pipeline stages within the cluster. Thus there is a limit to the number of execution units that can be incorporated into a cluster without causing wiring delays to become a significant fraction of the cycle time. Wiring track limitations also influence the partitioning of the cluster into distinct integer and floating-point components. A unified integer and floating-point register file is not attractive because of the need for more register file ports. In addition, pitch-matching the register file to all three execute datapaths, which would reduce wiring complexity, is difficult.

Chapter 4

On-chip Interaction Mechanisms

Parallel speedup of applications is typically limited by the amount of concurrency available and the overhead of the mechanisms provided by the computer system to exploit it. Traditional multiprocessors have very expensive mechanisms for managing parallelism, typically requiring hundreds or thousands of cycles to invoke a thread, or to communicate and synchronize between threads. With long interaction latencies, thread communication, synchronization, and invocation must be infrequent in order to prevent the execution time from being dominated by interaction overheads. For example, if a communication costs 1000 cycles, then threads must spend more than 10,000 cycles executing between communications to keep the overhead under 10%. Parallelism that requires more frequent communication cannot be exploited effectively.

The MAP chip is designed to reduce the overhead of on-chip thread management, providing hardware support for thread communication, synchronization, and invocation. Communication between clusters takes place via direct register-register transfers which require only one cycle to deliver the data. A global cluster barrier (`cbar`) instruction enables a barrier synchronization across clusters, with only one cycle of overhead. A new thread can be forked into a separate MAP cluster in 14 cycles, using a user level `hfork` instruction. As shown in Chapters 5 and 6, these fast mechanisms can be used by independent processors to exploit instruction-level parallelism and fine-grain thread-level parallelism that has not previously been available. This chapter describes each of these thread management mechanisms, evaluates their overhead, and compares them to alternatives that lack hardware support.

4.1 Experimental Evaluation Tools

Specific microbenchmarks are used to directly evaluate the fine-grain thread control, communication, and synchronization mechanisms of the MAP chip. The microbenchmarks are written in MAP assembly code [DKC+94], and an executable is generated using the MMAS assembler and linker [Gur94]. MMAS is derived from the Multiflow assembler, and inherits the Multiflow macro-processor and instruction format.

The test programs are run on both MSIM and the MAP chip register transfer level (RTL) simulator. MSIM, a functional-level simulator of the MAP chip, is implemented in C. It executes 400-1000 MAP cycles per second, depending on the number of active clusters. In addition to executing programs, MSIM includes a breakpoint facility and mechanisms for debugging and profiling programs. MSIM can simulate an M-Machine with a network of multiple MAP chips, and has been parallelized to run on a multiprocessor.

The RTL is the logic design of the MAP chip, implemented in Verilog [TM91]. The RTL model, composed of 810 unique modules and 78,000 lines of code, was used to verify the MAP chip schematics. It accurately represents all of the modules of the chip and is exactly cycle accurate to the silicon. The control logic schematics of the MAP chip were synthesized directly from Verilog modules. Due to its detail, the RTL is much slower than MSIM, executing less than 15 cycles per second on a 300MHz Sun Ultra 2 workstation. The cycle accuracy of MSIM was determined by comparing it to the RTL model using the MAP chip verification suite, consisting of 663 programs containing 174,000 lines of assembly test code. The test programs were generated by hand, by our compiler, and by an automated random test generator. Over that test suite, MSIM and the RTL differ in cycle count by less than 5% per test.

4.2 Communication

In coarse grained multiprocessors, communication between threads is exposed to the application through memory references or messages. Today's distributed shared memory multiprocessors use a hardware cache coherence protocol to automatically transfer data between processors. A remote read that does not initiate any additional protocol messages between remote processors requires nearly 140 cycles on a high end symmetric multiprocessor [CPWG97]. Most message passing

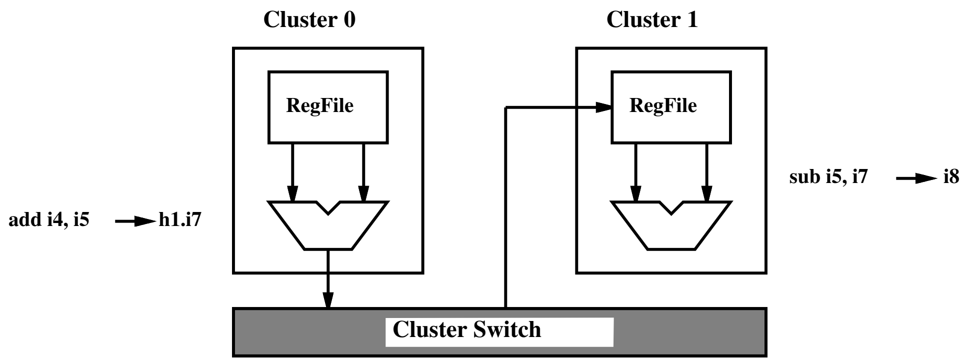


Figure 4.1: A remote register write via the Cluster Switch. When the `add` instruction on Cluster 0 executes, it writes its result into integer register 7 (`i7`) on Cluster 1. The `sub` instruction on Cluster 1 can later use the result.

multiprocessors lack integrated support for message injection and extraction, which results in even slower communication latencies (500 cycles on an Intel Paragon [CLMY96]). However, a message can convey more information than a single word transmission through shared memory. As a result of slow communication, today's multiprocessors are unable to exploit fine grain parallelism. In contrast, by incorporating multiple processors onto a single integrated circuit, the MAP chip enables fast communication between different threads. Threads may communicate through the on-chip cache, through the shared off-chip DRAM, or through registers. Since the data need not leave the chip to be transferred from one thread to another, communication is fast and well suited to fine–grain parallelism.

4.2.1 Communication Mechanisms

The MAP chip implements two mechanisms for communicating between clusters. Using the shared on-chip cache and local DRAM, threads can communicate with one another by loading and storing data to the same memory locations. Alternatively, a thread on one cluster can write directly into the register file of another cluster, via the Cluster Switch. As shown in Figure 4.1, the result of any arithmetic operation may be sent directly to a remote register, without interfering with memory references or polluting the cache. The `add` operation on cluster 0 writes into the register file of cluster 1, where it can subsequently be used by the subtract. Register–register transfers are extremely fast, requiring only one more cycle to write to a remote register than to

Operation	Producer Overhead	Consumer Overhead	Transfer Latency
Memory (cache miss)	2	2	36
Memory (cache hit)	2	2	10
Register	1	0	2

Table 4.1: Communication latencies between threads on different clusters.

a local register. Since the size of the register file limits the storage for communicated values, register communication is particularly suited to passing small amounts of data quickly, such as transferring signals, arguments, and return values between threads. One drawback is that register communication requires an additional synchronization between consumer and producer to prevent values in the destination cluster from being illegally overwritten. Memory communication is less prone to this because of the abundance of communication locations.

4.2.2 Communication Costs

Communication latency and overhead are evaluated with a producer-consumer microbenchmark. Both memory and register mechanisms are examined by passing a value back and forth between two clusters. The memory version uses two memory locations, one for each communication direction. Spin locks using the MAP's memory synchronization bits implement the synchronization between the threads. The producer stores its value to the target location and marks the memory location full, while the receiver spins on the location, waiting for the data to arrive. The register version uses the `empty` instruction and remote register writes. The producer empties its receiving register and writes the value to the consumer's register file. The consumer stalls on the register until the value is written and the scoreboard is marked full.

Three components contribute to the latency of cross-cluster communication. The *producer overhead* is the number of cycles that the producer must spend initiating the transfer. The *consumer overhead* is the number of cycles that the consumer must spend executing instructions to synchronize with the data arrival. The *transfer latency* is the total time from the producer initiation to the use by the consumer. Table 4.1 shows the producer overhead, consumer overhead, and transfer latency for memory and register communication. Before transferring the data, the memory versions must

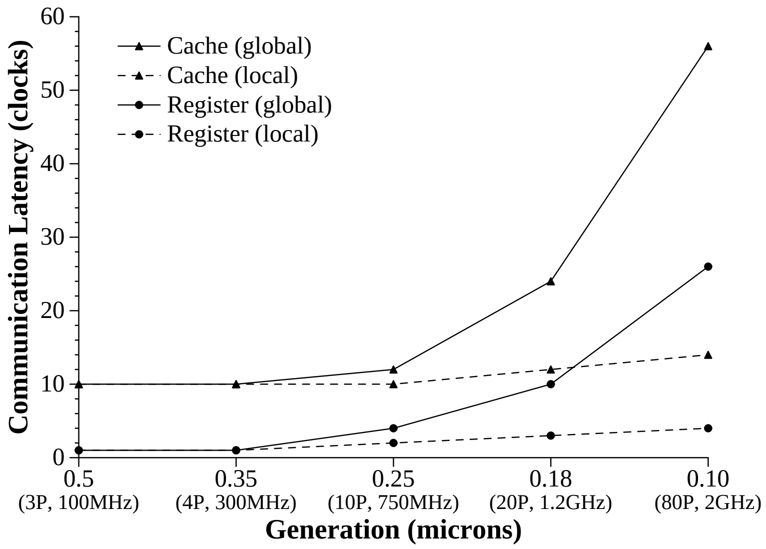


Figure 4.2: Technology scaling of communication mechanisms. As devices become smaller, more processors can be incorporated on-chip and the latency between remote processors increases.

---

first compute the address of the communication location, which takes 3 cycles. In the register version, the remote location for the data is encoded in the instruction performing the transfer, resulting in only a single cycle producer overhead. In fact, if the data is delivered by an arithmetic instruction, the producer overhead is zero cycles, as the instruction is necessary regardless of its destination. If all memory accesses hit in the cache, memory communication has a 10 cycle transfer latency, including the producer overhead and two memory latencies, one each by the producer and consumer. If both producer and consumer memory references miss in the cache, then the total transfer latency can be as long as 36 cycles. Register communication has only one additional cycle of latency for the Cluster Switch traversal, and the consumer is able to use the data immediately.

Figure 4.2 shows how register and memory communication mechanisms are likely to scale with advances in silicon process technology. As described in Section 2.5 future process technologies will enable the number of processor clusters to grow from three in a 0.5μm technology (3P) to eighty in a 0.1μm process (80P). For the three largest chip models (10, 20, and 80 processor clusters), the processors are divided into groups of 5 processors sharing a second level cache. Latencies are shown for both global communication between two remote processors (global) and local communication between two processors in the same group (local). The Register lines show the latency to transmit a values between processors using remote register writes. As this latency is dominated by wire delay, it increases linearly with distance. The latency to communicate within a processor group increases only to a maximum of four cycles as the decrease in physical area of a processor group due to smaller devices counterbalances the increasing wire delay. Global communication rises more dramatically, reaching 26 cycles in a 0.1μm process, because both the effective distance between remote processors and the wire delay is increasing with advancing technology.

The latency to communicate between remote processors using a cache (T_{cache}) is indicated by the Cache lines. In this model for T_{cache}, a source processor writes the shared data into a local memory module. A remote processor then loads the data and must wait a full round-trip delay to fetch the data from the source. This delay can be expressed by the following equation:

$$
\begin{aligned}
T_{cache} &= \text{2 cycles for store overhead at source} + \\
&\quad \text{2 cycles for load overhead at destination} + \\
&\quad 2 \times (\text{round trip communication latency})
\end{aligned}
$$

Figure 4.2 shows both the global and the local cache communication latencies using this simple wire delay dominated model. Similar to the local register latencies, memory communication within a group of processors does not increase dramatically. The increasing wire delay and the shrinking group size offset one another. However, the latency for remote cache communication grows at approximately twice the rate as direct register communication and reaches 56 cycles for 80 processor clusters in a 0.1μm process. There are opportunities for optimizing cache communication by selecting the communication location to be close to the destination. In this case only one corner-to-corner delay is in the critical communication path as the source can store directly into a remote memory location. In reality, though, using caches can be even worse than shown in the graph due to additional overheads of indirect communication. Traversing the memory hierarchy can add overhead at each level because the data must drop down into a location in a memory array which may require multiple cycles to access. Memory communication requires also requires spinning or polling which incurs both instruction and memory bandwidth overheads. Direct communication can be synchronized explicitly using the scoreboard, which causes the destination processor to stall rather than poll while waiting. Another overhead of memory communication is that the memory location which is being used as the communication point may not be on a direct path between the source and the destination processors. Thus the wire delay can be much larger than just the corner-to-corner latency. The gap between indirect communication through memory and direct communication through registers is already significant and is increasing with faster transistors and slower wires of future process technologies.

4.3 Synchronization

In a concurrent system, synchronization must be used to indicate when a task is to be started, when it is complete, or when two running threads must communicate. Multiprocessors, such as Alewife [AKK+93], have typically provided memory based synchronization instructions and used those to build barriers and producer/consumer locks. Some, such as the CM-5, implement a global barrier mechanism in hardware using a hierarchical barrier network [LAD+96]. The MAP chip allows on-chip threads to synchronize through memory, registers, and a hardware barrier instruction, while threads on separate MAP chips synchronize using messages.

4.3.1 Memory Synchronization

In the MAP chip, every memory location has a synchronization bit that exists both in the off-chip DRAM and in the cache, enabling locking on a location-by-location basis. Special load and store operations allow atomic testing and setting of the bit. The code fragment below shows how a spin-lock may be implemented using a memory synchronization bit. The load and synchronize operation (`ldsu`) loads the value at the address held in register `i8`, into `i9`. In the memory system the synchronization bit is compared to the precondition `pre_1`. If they are the same, the operation succeeds: the synchronization bit is set to `post_0`, the contents of the location are returned to `i9`, and the value `true` is returned to condition code register `cc0`. Otherwise, the location remains unchanged, and `false` is returned to the condition code register. The subsequent branch will cause the loop to spin until the operation succeeds.

```
_loop:
   instr memu ldsu pre_1, post_0, i8, i9, cc0;
                                    /* load from address in i8,
                                       compare memory synchronization bit to pre_1,
                                       set memory synchronization bit to post_0,
                                       return result of test in cc0 */
   instr ialu cf cc0 br _loop;      /* if test fails, try again */
```

A similar operation is used to store a value and set the synchronization bit. This synchronization mechanism can be incorporated with memory communication between threads, allowing synchronization on a word by word basis. However, a consumer thread waiting for a producer will continue to make memory requests while spinning, which can slow down other threads trying to access the memory system. As an alternative to spinning, the MAP chip can implement blocking and automatic retry. When the memory system detects a synchronization failure from a specific set of synchronizing load and store instructions, it triggers a trap to software. When the user code tries to use the result of a synchronizing load that trapped, it stalls in the pipeline waiting for the data to return. The trap handler can run in parallel with the user code and can retry to faulting reference or swap out the waiting thread.

4.3.2 Instruction Synchronization

The MAP chip uses full/empty bits in a register scoreboard to determine when values in registers are valid. When an operation issues, it marks the scoreboard for its destination register invalid. When the operation completes, it writes its result to the destination register and marks the scoreboard valid. Any operation that attempts to use the register while it is empty will stall until the register is valid. To reduce the amount of interaction between physically distant clusters, an operation that writes to a remote cluster does not mark its destination register invalid. Instead, the consumer must execute an explicit `empty` instruction to invalidate the destination register prior to receiving any data. When the data arrives from a remote register write, the scoreboard is marked valid and any operation waiting on the register is allowed to issue. Using register–register communication fuses synchronization with data transfer in a single operation and allows the consumer to stall rather than spin.

The simplest synchronization mechanism implemented by the MAP is the cluster barrier instruction `cbar`. The `cbar` instruction stalls a thread's execution until the threads on the other two clusters have also reached a barrier. Threads waiting for cluster barriers do not spin or consume any execution resources, so other threads can use the execution units instead. Figure 4.3 shows how `cbar` can be used to orchestrate intercluster interactions. In order to guarantee correct data synchronization, the `cbar` ensures that the `empty` of register `i7` on cluster 1 executes before the `add` that transfers the data from cluster 0. In addition, `cbar` can be used to enforce order between

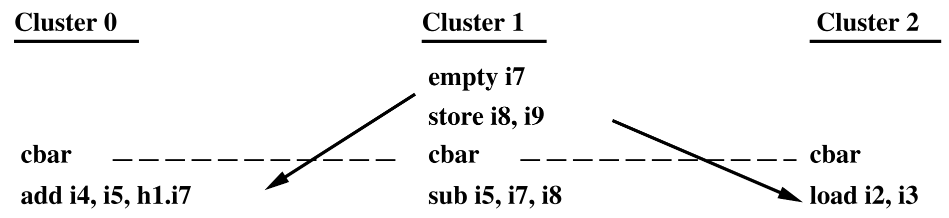


Figure 4.3: The cluster barrier `cbar` instruction enforces synchronization across all three clusters.

load and store instructions on different clusters that may reference the same address.

For purposes of experimenting with instruction-level parallelism, the MAP provides a tightly coupled (TC) mode bit that enables the MAP to simulate a VLIW machine across all three clusters. In tightly coupled mode, each instruction contains an implicit `cbar` and instructions on separate clusters issue in pseudo lock-step. Aligned instructions do not issue simultaneously across the clusters, but instruction $i+1$ on one cluster will not issue until instruction i on all of the other clusters has issued. This provides the compiler with guarantees about the execution order of instructions across the clusters so that it can use VLIW scheduling techniques. Tightly-coupled mode will be discussed further in Chapter 5.

4.3.3 Synchronization Costs

Not all synchronization can be easily expressed using a producer-consumer model. A barrier can be used to conglomerate several synchronizations into a single action. Fast barriers reduce the overhead of using parallelism, which is vital if the parallelism to be extracted has short task execution times between synchronizations. Four implementations of barriers across three clusters are examined: memory, register, condition-code, and CBAR. The memory implementation uses four memory locations; one location holds the barrier counter, and each thread has its own location on which to spin. Upon reaching the barrier, each thread performs a fetch and increment on the counter, using the MAP's memory synchronization bits. If the barrier count is less than 2, the thread begins spinning on its own memory location. If the barrier count equals 2, then the other threads have already reached the barrier. The last arriving thread resets the counter to zero, and releases the spinning threads by marking their memory locations full.

Barrier Method	Latency
Memory (cache hit)	61
Register	6
Condition Code	5
CBAR	1

Table 4.2: Latency to execute a barrier across all three clusters. Even with an on-chip cache, synchronizing using memory is more than ten times as expensive as using registers or the `cbar` instruction.

The register barrier microbenchmark consists of an even phase barrier, followed by an odd phase barrier. Upon reaching the barrier in an even phase, a thread empties its odd phase registers, and writes into the even phase registers of both of its neighbor threads. It then reads from its own even phase registers, stalling until they have been written by the neighbors. Two registers per phase are necessary to allow each of the neighbors to communicate independently. The Condition Code barrier is similar except that with the broadcast capability of global condition code registers, only one instruction is required to signal to both neighbor threads. The CBAR barrier uses the `cbar` instruction, without requiring any registers or auxiliary instructions to be executed.

Each mechanism is implemented in a simple program that does 100 successive barriers. The time per barrier in the steady state is measured and shown in Table 4.2. The `cbar` instruction is the fastest and can complete a barrier every cycle. The register and condition code barriers are similar, with Condition Code being one cycle faster since only one write is necessary to communicate with both neighbors. The memory barrier requires 61 cycles, even with all accesses hitting in the cache. For each thread, approximately 20 cycles are needed for the control overhead of testing the barrier counter, while the remaining cycles are consumed contending for the on-chip cache and waiting for the other threads to arrive at the barrier. In order to exploit fine–grain parallelism with task lengths in the 10s of cycles, long latency memory-based barriers cannot be used.

Figure 4.4 shows estimates of the scalability of barrier synchronization to larger numbers of processor clusters for the same process technologies described in Section 4.2. Latencies are shown for global barriers encompassing all of the on-chip processor clusters (global) and for local barriers synchronizing within a processor group (local). The CBAR lines represents the time to perform

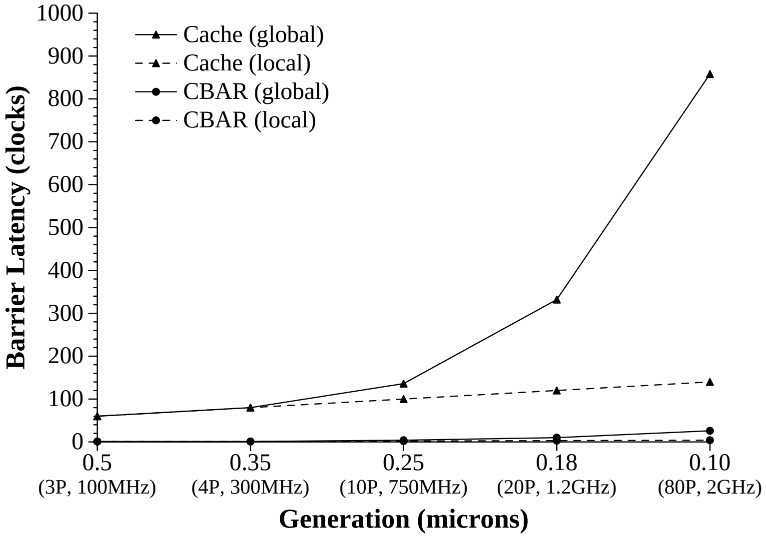


Figure 4.4: Technology scaling of barrier synchronization.

---

a barrier in hardware using a global barrier instruction. This latency is composed primarily of wire delay and increases linearly with the maximum distance between synchronizing processors. The time to complete a local barrier among processors in a common group increases only slightly (from 1 to 4 cycles) as the shrinking of a processor group counterbalances the increasing effect of wire delay. For a global barrier, the barrier time is equivalent to the on-chip corner-to-corner transmission latency, which is 26 cycles for a 0.1μm process with 80 processors.

The latency to synchronize via on-chip memory (Cache) requires a more complicated software protocol. The model for software barriers is that the processor clusters are placed at the leaves of a tree, where each internal node of the tree has 3, 4, or 5 children depending on the total number of clusters. A barrier is performed locally first by all clusters with a common parent in the tree. Subsequent barriers are performed hierarchically up the tree toward the root. After synchronization at the root has been completed, the barrier status is distributed down the tree until it reaches the leaves. The latency to perform this barrier can be expressed by the following equations.

$$\begin{aligned}
T_{cache} &= \textstyle\sum_{i=0}^{m-1} n\times(8+4\times(\frac{latency_{cc}}{m-i})) \\
m &= \text{depth of tree} \\
n &= \text{number of clusters grouped at leaf level} \\
latency_{cc} &= \text{corner-to-corner wire latency}
\end{aligned}$$

T_{cache} is the sum of barrier times across all levels of the tree and accounts for the barrier latency at each level. The time to execute a barrier at each level is approximated by 8 instructions and 4 memory references. The time to execute a memory reference depends upon the distance between the components participating in the barrier which is in turn determined by the level of the barrier in the tree. Barriers among a local group of processor clusters can use local memory references with short latencies. The local barrier time starts at 60 cycles for 3 processors in the 0.5μm process and only increases to 140 cycles for 5 processors in a 0.1μm process. The primary factor in this increase is a larger delay to the memory shared by the processors in the group. A global barrier across all of the on-chip processors requires global communication. This simple model shows that the cost to execute a barrier using memory for communication between processors is already quite expensive and will become even more costly in future technologies. At least 850 cycles will be required to synchronize 80 processors in a 0.1μm process. The increase is due both to longer global wire delays and to more processors participating in the barrier. More direct mechanisms, such as a cluster barrier instruction that can synchronize directly between processors, require far less time to complete a barrier.

4.4 Thread Creation

Invoking a thread on a remote processor is typically an expensive operation. The source thread must send a message that contains a pointer to the invoked function as well as all of the necessary arguments. At the destination, a kernel call is used to set up the stack and to initialize all of necessary data structures for the new thread. Finally, the function arguments are unmarshalled and passed to the thread for execution. The overhead to initialize the thread can be in the 10s of microseconds, which renders conventional methods inappropriate for fine-grain parallelism in which threads are invoked frequently.

4.4.1 New Threads

In the MAP chip, a new thread can be invoked on a remote cluster either by modifying the thread control state through a series of `store` instructions, or by executing a single user level `hfork` instruction. A privileged thread can update thread control state by writing to the thread control registers through the memory mapped global configuration space interface. Figure 4.5 shows the

```
 #define C1_PC   0xc0240000000011a8            /* Cluster 1 program counter */
 #define V0_HRUN 0xc028000000000408            /* H-Thread Run bits */
 #define V0_HACT 0xc028000000000808            /* H-Thread Active bits */
 #define C1_INIT 0xc024000000001150            /* Cluster 1 initialize */

 instr ialu imm (_c1_proc - _here), i4;        /* offset for procedure */
_here:
 instr ialu lea i1, i4, i6;                    /* create instruction pointer */

 /* Generate GCFG address for cluster 1 program counter */
 instr ialu imm   ##XTR(C1_PC,48,63), i9;
 instr ialu shoru ##XTR(C1_PC,32,47), i9;
 instr ialu shoru ##XTR(C1_PC,16,31), i9;
 instr ialu shoru ##XTR(C1_PC,0,15),  i9;
 instr ialu setptr i9, i9;

 /* Generate GCFG address for cluster 1 initialization */
 instr ialu imm   ##XTR(C1_INIT,48,63), i10
       memu st i6, i9;                         /* store IP into C1_PC */
 instr ialu shoru ##XTR(C1_INIT,32,47), i10;
 instr ialu shoru ##XTR(C1_INIT,16,31), i10;
 instr ialu shoru ##XTR(C1_INIT,0,15),  i10;
 instr ialu setptr i10, i10;

 /* Generate GCFG address for H-Thread Run bits */
 instr ialu imm   ##XTR(V0_HRUN,48,63), i11
       memu st i0, i10;                        /* initialize cluster 1 */
 instr ialu shoru ##XTR(V0_HRUN,32,47), i11
 instr ialu shoru ##XTR(V0_HRUN,16,31), i11;
 instr ialu shoru ##XTR(V0_HRUN,0,15),  i11;
 instr ialu setptr i11, i11;

 /* Generate GCFG address for H-Thread Active bits */
 instr ialu imm   ##XTR(V0_HACT,48,63), i12
       memu mov ##0x3, i3;
 instr ialu shoru ##XTR(V0_HACT,32,47), i12
       memu st i3, i11;                        /* set cluster 1 H-Thread run */
 instr ialu shoru ##XTR(V0_HACT,16,31), i12;
 instr ialu shoru ##XTR(V0_HACT,0,15),  i12;
 instr ialu setptr i12, i12;

 instr memu st i3, i12;                        /* set cluster 1 H-Thread active */
```

Figure 4.5: Fork a thread into Cluster 1 using the Global Configuration Space controller.

```
  instr ialu imm (_c1_proc - _here), i4;     /* offset for procedure */
_here:
  instr ialu lea i1, i4, i6;                 /* create instruction pointer */

  instr memu hfork i6, #1, cc0;              /* fork procedure into cluster 1 */
  instr ialu cf cc0 br _hfork_fail;          /* jump to fail if cluster 1 is busy */
```

Figure 4.6: Fork a thread into Cluster 1 using an `hfork` instruction.

MAP assembly code executed in thread slot 0 of cluster 0 to generate the configuration space addresses and update the remote thread state in thread slot 0 of cluster 1. First, the pointer to the remote procedure `_c1_proc` is created in `i6` using an immediate offset from the current instruction pointer (stored in `i1`). The global configuration space address for the remote program counter is generated through a series of immediate (`imm`) and shift-and-or-unsigned (`shoru`) operations. The `XTR` macro extracts a 16 bit field from the constant `C1_PC`. The 64-bit constant is turned into a guarded pointer using the set pointer (`setptr`) instruction. After storing the function pointer into the remote program counter, the pipeline registers from the remote thread slot are cleared by storing to `C1_INIT`. Finally, the `HRUN` and `HACTIVE` bits for thread slot 0 of cluster 1 are set to allow the instructions from the new thread to be fetched and executed. Both `HRUN` and `HACTIVE` fields consist of three bits, one for each cluster. These fields are set to `(011)` (which is equivalent to `0x3`) to allow both both cluster 0 and cluster 1 to run. The entire operation requires 23 instructions and must run in system mode to allow the privileged `setptr` instruction to execute.

As a much faster alternative, the MAP chip introduces an `hfork` instruction, shown in Figure 4.6. To fork a thread, the new instruction pointer must be created as before, but all of thread management is encapsulated in the `hfork`. The `hfork` instruction specifies the instruction pointer to run (`i6`) and the cluster in which the new thread will reside (`#1`). The `hfork` also specifies a return condition code register (`cc0`) in which a single bit indicating the success or failure of the fork is written. If cluster 1 is already executing, then zero is returned in `cc0`; otherwise the `hfork` succeeds and one is returned. The `hfork` is treated as a store operation by the cluster and delivered directly to the global configuration space controller. There it starts a simple state machine which initializes the target pipeline registers, writes the program counter, and updates the thread active and run bits. This operation requires only 4 instructions at the source and can be executed completely in an unprivileged mode.

```
_master_code:
     instr ialu imm (_c1_proc - _here), i4;     /* offset for procedure */
   _here:
     instr ialu lea i1, i4, i6;                 /* create instruction pointer */
     instr ialu imm _slave_loc, i8;             /* offset for shared memory location */
     instr ialu lea i3, i8, i8;                 /* generate pointer to memory */
     instr memu stsu ua, 1, i6, i8, cc3;        /* store IP and mark location full */

_slave_code:
     instr ialu imm _slave_loc, i8;             /* offset for shared memory location */
     instr ialu lea i3, i8, i8;                 /* generate pointer to memory */
  _slave_spin:                                  /* spin wait for master */
     instr memu ldsu ct, 0, i8, i3, cc0;        /* load and set location empty */
     instr ialu cf cc0 br _slave_spin;          /* spin if location still empty */
     instr;                                     /* branch delay slots */
     instr;
     instr;

     instr ialu jmp i3;                         /* jump to _c1_proc */
     instr;                                     /* branch delay slot */
     instr;
     instr ialu lea i1, #4, i4;                 /* calculate procedure return pointer */

     instr ialu br _slave_code;                 /* branch to _slave_code for next call */
```

Figure 4.7: Fork into a waiting thread in Cluster 1 using memory communication.

```
_master_code:
     instr ialu imm (_c1_proc - _here), i4;     /* offset for procedure */
   _here:
     instr ialu lea i1, i4, i6;                 /* create instruction pointer */
     instr ialu mov i6, h1.i5;                  /* write IP to slave thread */

_slave_code:
     instr ialu jmp i5;                         /* jump to _c1_proc */
     instr;                                     /* branch delay slot */
     instr;
     instr ialu lea i1, #4, i4;                 /* calculate procedure return pointer */

     instr ialu br _slave_code;                 /* branch to _slave_code for next call */
     instr ialu empty ##0x0020;                 /* empty i3 for call */
```

Figure 4.8: Fork into a waiting thread in Cluster 1 using register–register communication.

4.4.2 Waiting Threads

A second method of fast thread invocation can be implemented by installing a simple dispatch handler on the remote thread. The dispatcher waits until a source thread sends an instruction pointer. Figure 4.7 shows the assembly code to invoke a remote thread using the shared on-chip cache, with the master code running on cluster 0, and the slave code running on cluster 1. The dispatch handler in the slave thread (starting at `_slave_code`) spins on a memory location waiting for it to be marked full. The master thread must generate the pointer to the shared communication location with the `_slave_loc` offset from the shared heap pointer in `i3`. The master then stores the program counter into that memory location, and marks it full with a synchronizing store operation `stsu`. When `_slave_loc` is full, the slave jumps to the instruction pointer and stores the procedure return pointer in `i4`. When the subroutine is complete, the slave resumes its spinning on `_slave_loc`.

The MAP chip can avoid using the memory system for thread invocation by sending the instruction pointer from the master to the slave using a remote register write. As shown in Figure 4.8, the slave stalls on an empty register `i5`, waiting for it to be written by the master. After creating the instruction pointer, the master writes it directly into `i5` on cluster 1, marking the destination register full. The slave then jumps to the invoked procedure, and writes the function return pointer in `i4` as before. When the subroutine is complete, the slave re-empties `i5` and branches back to wait for a new instruction pointer.

4.4.3 Invocation Costs

Four methods for starting a thread on a remote cluster, including two cold start and two standby, are examined. The first cold start method invokes the thread using the global configuration space controller, while the second uses the `hfork` instruction. The standby methods already have a slave thread running in the remote cluster waiting for a new task from the master. The two standby methods differ in how the master and slave communicate with one another, using either registers or memory.

Figure 4.9 shows the four components of a null thread call and return. The *master call* overhead is the number of cycles that the master must spend executing instructions to create the new thread. The *slave invoke* latency is the time from the beginning of the master call to the execution of the

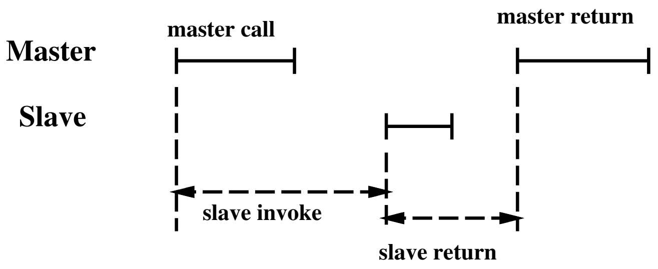


Figure 4.9: Components of thread invocation and return.

slave's first instruction. The *slave return* latency is the time for the slave to signal to the master. Finally the *master return* is the overhead for the master to resynchronize with the slave.

Table 4.3 summarizes the components of latency for each of the four methods. GCFG is the most expensive due to the address calculation required. In addition, GCFG needs several registers to perform the address calculation, and may ultimately require spilling some live values to memory if no free registers are available. The `hfork` instruction and the *standby register* method are the most efficient, with only 1 cycle of overhead for the master at the call and return. *Standby register* is a little faster overall as the slave invocation time is shorter. *Standby memory* is almost three times worse than the register version because of the memory spin loops the master and slave use to synchronize.

Although *hfork* is faster than *standby register*, both standby versions have an advantage when the thread is invoked more than once. When an `hfork` instruction executes, the target thread's registers are cleared automatically. Thus any stack or heap pointers that the target thread needs, must be transmitted from the master to the slave on every invocation. When the thread idles in the standby methods, only the function pointer and its arguments need to be transferred, since the other registers can be persistent across invocations.

4.5 Summary

In order to efficiently execute fine–grain parallel programs, the underlying system, including both the hardware and system software, must support fast thread interaction. Traditional multiprocessor interactions based on shared memory or messaging mechanisms are inadequately slow. In a single

Operation	Master Call	Slave Invoke	Slave Return	Master Return	Total
GCFG	21	27	10	9	46
hfork	1	11	2	1	14
standby memory	3	21	6	9	36
standby register	1	7	2	1	10

Table 4.3: Latencies for the overheads associated with thread invocation. The total time is end–to–end latency of a null remote invocation. Using the `hfork` instruction or register communication yields an overhead three times smaller than using memory operations.

---

chip multiprocessor, interactions can become much faster if certain novel features are employed. The MAP chip introduces fast on-chip interprocessor mechanisms such as single cycle communication, a single cycle barrier instruction, and 10 cycle thread invocation. Microbenchmark studies show that these mechanisms allow communication that is 10 times faster, and synchronization that is 60 times faster than mechanisms that use only an on-chip cache. As more processor clusters are incorporated in future chips with smaller feature sizes, the gap between direct interaction mechanisms and interacting through memory will continue to widen. With 80 on-chip processors in a 0.1μm process, cross chip communication will require at least 56 cycles using the memory operations and only 26 cycles with direct register writes. Both of these communication latencies scale linearly with the distance between the processors, but cache communication will incur additional constant software overheads. The latency for memory based global synchronization will increase to more than 850 cycles. With direct hardware support for barrier synchronization, such as the `cbar` instruction, the latency scales linearly with the maximum distance between processors participating in the barrier and can be as low as 26 cycles. Synchronizing through memory scales approximately at a rate of $d{\cdot}\lg(n)$ where d is the global wire delay and n is the total number of processors to synchronize. The additional software overhead for coordinating a hierarchical barrier is a substantial contributor to the overhead of the barrier. Thread invocation can also be significantly faster with a small amount of hardware support. Using the `hfork` instruction or invoking a remote procedure by sending the function pointer and arguments to the registers of a standby thread is three times faster than the corresponding methods that uses load and store instructions.

The fast communication, synchronization, and thread invocation mechanisms described in this chapter are extremely important to future single-chip computer systems. The increasing number of transistors combined with the constraint of slower wires between the active devices dictates architectures that partition their execution units into a large number of independent processors. This creates a tremendous demand for concurrency that cannot be met using traditional coarse-grained application parallelism. With fast and scalable communication and synchronization mechanisms, fine-grain thread level parallelism can be extracted from the applications that people run every day. In today's technology, the MAP implementation shows that threads need only to execute 10 cycles between interactions to keep the overhead below 10%. Fine grain threads that execute for less than 100 cycles are already feasible. In future technologies, physical locality will become even more important, requiring processors that communicate frequently to be located close to one another. Hardware support for direct communication and synchronization will allow the interaction latency between processors to increase only linearly with the distance between interacting processors. As will be shown in Chapter 5, communication and synchronization is fast enough to enable execution of instruction level parallelism across physically distributed and independent processors.

Chapter 5

Instruction-Level Parallelism

Instruction-level parallelism (ILP) has long been an attractive method for improving computer system performance, as it typically can be used without modifying applications. Today's dynamically scheduled superscalar microprocessors examine instructions held in a scheduling window, determine which can be executed concurrently, and distribute them to multiple execution units. To avoid some of the limits of ILP [Wal91], superscalars employ increasingly complicated microarchitectures which automatically expose more parallelism in existing binary programs. Techniques such as register renaming, adaptive branch prediction, and out-of-order execution are all intended to increase the number of instructions that can be considered for execution.

At the other end of the spectrum from dynamic scheduling are Very Long Instruction Word (VLIW) architectures, which have been used in scientific supercomputers such as the Multiflow Trace [CNO+88] and more recently incorporated into digital signal processors like those in the Texas Instruments TMS320C6x family [Dil97]. VLIW machines statically schedule their instruction streams across multiple arithmetic units, avoiding the hardware complexity of superscalars. A VLIW compiler also uses a notion of an instruction window for scheduling, but the software window size can be much larger than one fixed in hardware. Thus with compiler scheduling, instructions from very different parts of the program can execute concurrently. Since software scheduling is also limited by control dependencies, a number of compiler algorithms, such as trace scheduling [LFK+93], software pipelining [Lam88], and hyperblocks [MLC+92], have all been used to increase the amount of ILP visible to the compiler. These compilation techniques have also proven to be useful in scheduling code for superscalar machines.

With the faster gates and slower wires of future silicon process technologies, microprocessor chips must be partitioned into components that exploit local communication and limit global communication. The global control of superscalar processors will prevent them from being scaled beyond 8–16 arithmetic units. Static VLIW machines are better suited for scalability, but still require a central control unit.

This chapter discusses how instruction-level parallelism can be exploited on the independent on-chip processors of the MAP chip, using its fast intercluster communication and synchronization mechanisms. Section 5.1 describes the constraints that limit ILP and result in an uneven amount of parallelism in different parts of the program. Section 5.2 discusses the relaxed synchronization across the clusters of the MAP and compares it to both superscalar and VLIW architectures. Finally, Section 5.3 uses a synthetic benchmark and two application kernels to evaluate the loose coupling between MAP processors on ILP code, by comparing it to a lock-step VLIW execution discipline.

5.1 Limits of ILP

A major factor that limits the availability of instruction-level parallelism is uncertainty in the control and data flow of a program. Control uncertainty stems from conditional branches which determine the dynamic path through a program. As the direction of a branch is often not known until only slightly before the branch is executed, the instructions that can be considered for scheduling may be restricted. Both hardware and software branch prediction techniques are currently used by computer systems in an attempt to increase the window of instructions that can be examined for scheduling or execution. Predicated execution of instructions, in which each instruction may be conditionally executed based on the value of a condition code, may also be used to eliminate some conditional branches and prevent the linear execution of an instruction stream from being interrupted.

Data uncertainty is typically a result of being unable to determine in advance whether a load and a store instruction reference the same address. If the address is different, then the load and store may be reordered, but if they reference the same location, the instructions must be executed sequentially. Compilers have had some success with memory disambiguation, particularly with languages that use structured array accesses [GKT91]. Performing memory disambiguation in

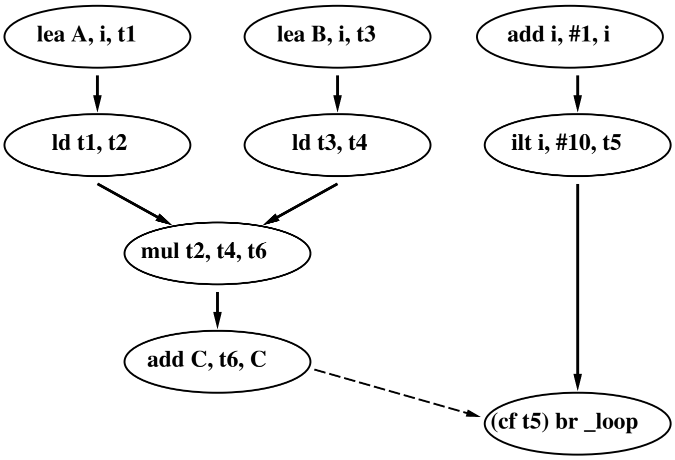


Figure 5.1: Dependence graph for inner loop of dot product. Data dependence is indicated with the solid arrows, while control dependence is indicated with the dashed arrow. The performance of the schedule is limited by the critical path, and the instruction-level parallelism varies throughout the program.

hardware has also been proposed [FS96], but this is a cumbersome process that requires complicated address comparisons to be performed dynamically.

If all of the control and data uncertainty could be correctly predicted, the only limit to the amount of ILP would be the sequential data dependent instructions that form the critical path of an application. The degree of ILP can vary throughout different parts of the program. Figure 5.1 shows the dataflow graph for the inner loop of the dot product code shown below:

```
for(i=0; i<10; i++) {
  C = C + A[i]*B[i];
}
```

In this example without loop unrolling, the degree of ILP varies from one to three instructions. The critical path for data dependencies has 4 instructions, including an address calculation (`lea`), a load from memory (`ld`), and two arithmetic operations. The loop control instructions are aligned in a separate 3 instruction path, but control dependencies may require the branch to be executed after the final add. The solid data arcs in the graph indicate communication between instructions, and if instructions are placed on different execution units, the communication overhead can limit the amount of instruction-level parallelism that can be exploited. A scalable system for executing ILP code must employ fast communication mechanisms between the execution units, and if possible schedule the program so that the amount of communication needed between instructions on different execution units is minimized.

5.2 Instruction-Level Parallelism on the MAP chip

In the MAP chip, each processor cluster has its own program counter, and the integrated communication and synchronization mechanisms enable fast intercluster interactions. A single ILP instruction stream can be partitioned into substreams that are placed on different clusters. Data is transferred explicitly from one stream to another by writing into a remote cluster's register file. Results of comparisons can be broadcast to all clusters allowing each of them to perform the same conditional branch. Clusters synchronize only when necessary for control or data dependencies. If a stream stalls while waiting for a long latency operation, the instruction streams on the other clusters can proceed until they reach the next synchronization point.

5.2.1 Loosely Coupled Execution Streams

Instructions streams on different MAP clusters are *loosely coupled*, which means their instructions do not execute in lock-step. Instead, synchronization is explicit in the program and is only inserted when needed to coordinate control or data dependencies between instructions. Figure 5.2 details the assembly code for two MAP clusters to execute the inner-loop of a linear relaxation:

```
for(i=0; i<imax; i++) {
  A[i] = (A[i+1] + A[i+2] + A[i-1] + A[i-2]) / 4;
}
```

Cluster 0 performs the address calculations and arithmetic operations for `A[i+1]` and `A[i+2]`, while cluster 1 does the same for `A[i-1]` and `A[i-2]`. In this partitioning of the work, three communications are required between the clusters and a total of 15 instructions lie on the critical path. At the beginning of the loop, the index variable `i` is passed to cluster 1 in register `i7`. Remote registers are named by prefixing a cluster identifier to the register number. Destination clusters are indicated with relative names so that `h1` is next numerically named cluster. After computing its

Cluster 0

```
_top_0:

    instr memu cbar;
    instr ialu mov i7, h1.i7
          memu lea i6, i7, i8;
    instr ialu add i7, #1, i7
          memu add i7, #2, i11;
    instr ialu lea i6, i7, i9;
          memu lea i6, i11, i10;
    instr memu ld i9, i9;
    instr ialu empty {i12}
          memu ld i10, i10;
    instr memu cbar;
    instr ialu add i9, i10, i10;
    instr ialu add i10, i12, i10;
    instr ialu lsh i10, #-2, i10;
    instr memu st i10, i8;
    instr memu cbar;
    instr ialu ilt i7, i3, h0.cc0;
    instr ialu ct h0.cc0 br _top_0;
```

Cluster 1

```
_top_1:
    instr ialu empty {i7}
    instr memu cbar;
    instr ialu sub i7, #1, i8
          memu sub i7, #2, i9;
    instr ialu lea i6, i8, i8
          memu lea i6, i9, i9;
    instr memu ld i8, i8;
    instr memu ld i9, i9;



    instr memu cbar;
    instr ialu add i8, i9, h2.i12;
    instr ialu empty {h2.cc0}

    instr memu cbar;

    instr ialu ct h2.cc0 br _top_1;
```

Figure 5.2: Assembly code for 4-point relaxation on 2 MAP clusters.

sum, cluster 1 returns it by writing the result of the `add` directly into `i12` in cluster 0. Cluster 1 writes to registers in cluster 0 by prefixing the `h2` to the register number. At the end, cluster 0 tests `i` against the loop limit and uses the integer less than (`ilt`) instruction to broadcast the comparison result to cluster 1. Each communication is preceded by an `empty` instruction and a barrier (`cbar`) to guarantee that the consumer receives the correct data. Without the synchronization, the programs running on the two clusters may end up operating on different iterations and pass the wrong values data between them.

The communication and synchronization overhead can be reduced in two manners. First, the multiple barrier and empty instructions can be coalesced into a small number of `empty` instructions and a single barrier. Second, some variables can be replicated and maintained independently on different clusters. Figure 5.3 shows an optimized schedule for the relaxation that requires only one communication and one barrier for the entire loop. Here the index variable `i` is stored and incremented on both clusters, eliminating its transfer at the beginning of the loop. In addition, the comparison to determine if the end of the loop has been reached can now be performed on each

Cluster 0

```
_top_0:
   instr ialu empty {i12};
   instr memu cbar;
   instr memu lea i6, i7, i8;
   instr ialu add i7, #1, i7
         memu add i7, #2, i11;
   instr ialu lea i6, i7, i9
         memu lea i6, i11, i10;
   instr memu ld i9, i9;
   instr memu ld i10, i10;
   instr ialu add i9, i10, i10;
   instr ialu add i10, i12, i10;
   instr ialu lsh i10, #-2, i10;
   instr memu st i10, i8;
   instr ialu ilt i7, i3, cc0;
   instr ialu ct cc0 br _top_0;
```

Cluster 1

```
_top_1:

   instr memu cbar;
   instr ialu sub i7, #1, i8
         memu sub i7, #2, i9;
   instr ialu lea i6, i8, i8
         memu lea i6, i9, i9;
   instr memu ld i8, i8;
   instr memu ld i9, i9;
   instr ialu add i7, #1, i7
   instr ialu ilt i7, i3, cc0;
   instr ialu add i8, i9, h2.i12;


   instr ialu ct cc0 br _top_1;
```

Figure 5.3: Optimized assembly code for 4-point relaxation on 2 MAP clusters.

cluster and the result need not be broadcast. Finally, the empty of `i12`, which receives the result of the add from cluster 1, can be pushed to the top of the loop. If enough registers are available, all of the barriers can be eliminated by using and odd-even strategy in which the loop is unrolled and the communication registers are split into two sets. During the even section of the iteration, the odd registers are emptied for the next part of the loop. The phases of computation can be kept separate by an interlocking producer–consumer relationship between the two threads.

The number of required synchronizations depends on the number of registers needed to hold live variables. If the code has many live variables, then reserving registers for communication may require more register spills to memory. However, by exposing the interactions explicitly to the programmer or compiler, tradeoffs can be made between communication and storage that make the best use of the hardware resources.

5.2.2 Comparison to Superscalar

By partitioning the execution units into independent clusters, making interactions explicit, and requiring compiler scheduling of the communication, the MAP chip implementation is far simpler than dynamically scheduled superscalar architectures. Since the instruction issue logic in the MAP is distributed throughout the clusters, scaling to more execution units has no impact on a single cluster's complexity. Adding more execution units to a superscalar architecture faces problems in both the issue logic and in the register file. Since a superscalar's issue logic must schedule instructions from the instruction window to the execution units, its complexity is proportional to the product of the window size and the number of execution units. By increasing the number of execution units, the instruction window must grow so that more instructions can be evaluated simultaneously. Thus the issue logic complexity increases with the square of the number of execution units, which is not an attractive equation for scalability. In addition, the wire delay required to deliver an instruction from the central instruction window to a remote execution unit is likely to limit the clock rate.

The second problem in scaling superscalar microprocessors stems from the demand for more register bandwidth to supply all of the execution units with data. One alternative is to maintain a single monolithic register file in which all registers are accessible from all execution units. This is attractive because a dynamic scheduling algorithm need only to be concerned with instruction placement and not data placement. However, each additional register file port requires one more horizontal and vertical track per register cell to connect to the word and bit lines. As a result, the size of the register file grows with the square of the number of ports. With a large number of execution units, the register file becomes too large and too slow. Perhaps even more severe is the complexity of the bypass logic to deliver data from the output of one execution unit to the input of another. To do full bypassing among N execution units requires N^2 busses, which results in designs that become wire limited even when N is small.

An alternative to a superscalar's monolithic register file is to partition it into multiple banks, such as in the DEC 21264 [Gwe96]. However, this places additional burdens on the dynamic instruction scheduling logic as instruction and data placement must both be managed. Combined with a larger scheduling window, this scheduling problem is likely to be too difficult to accomplish in an aggressive clock cycle. Incorporating the MAP's mechanisms is attractive for scaling superscalar technology to larger numbers of execution units. Each MAP cluster could contain a dynamically scheduled superscalar processor, but clusters would still independent and use explicit communication and synchronization.

5.2.3 Comparison to VLIW

A VLIW processor has a single program counter and a single instruction stream which issues in lock-step across the execution units. The data dependencies are enforced by static knowledge of all of the instruction latencies. If the latencies change, a given program must be rescheduled in order to run correctly. In the MAP chip, the register scoreboard enforces the data dependencies between operations. Therefore, changing the memory or instruction execution latencies has no effect on the correctness of the program. In addition, since the MAP clusters are not synchronized in lock-step, long latency operations on different clusters can be overlapped, instead of sequentialized.

The Multiflow Trace/500 VLIW supercomputer used a distributed instruction cache to hold its instructions. When a branch was taken, the target was broadcast to all of the execution units. In order to reduce the memory size of the executable program, instructions were stored in a dense format which eliminated null operations (NOPs) corresponding to unused execution units. A complicated instruction cache refill engine was required to expand from the main memory format into the sparse format that was executed by the arithmetic units. The independent clusters of the MAP simplify the hardware required for instruction sequencing. Branch instructions are purely local operations within a cluster, and if an ILP program using all of the clusters changes control flow, then all of the clusters branch independently. Because synchronization between instructions on different clusters is explicit, NOP placeholders are not required. If a cluster's execution units are not used, then the MAP's synchronization pipeline stage will automatically issue NOPs while the thread waits at the next synchronization point.

5.3 Evaluation of Loose Coupling

In strictly statically scheduled VLIW machines, no hardware interlocks are used to enforce data or control dependencies. The compiler schedules the instructions, taking into account all of the hardware latencies at compile time. Unpredictable latencies, such as those associated with a cache memory system, are difficult to statically schedule. Either the compiler can assume the worst-case for all memory latencies, or the hardware can implement interlocks, such as a scoreboard, to enforce the data dependencies. However, if strict lock-step instruction execution is preserved, latencies on different execution units may be sequentialized, resulting in a longer critical path.

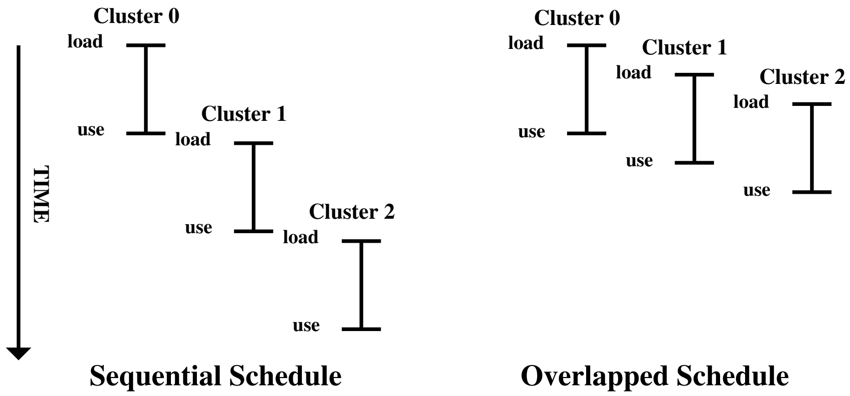


Figure 5.4: The effect of overlapping memory latencies.

Figure 5.4 shows the execution of two VLIW schedules consisting of three loads and three uses. In each case an instruction stream must wait at the point of use until the data from the load has returned. In the sequential schedule, a cluster's load follows the previous cluster's use, resulting in multiple waiting times. The overlapped schedule allows the loads to proceed in parallel and the clusters wait together for the data to return.

The MAP chip automatically overlaps latencies of all types by allowing its independent processing clusters to slip relative to one another. Synchronization is not implicit and lock-step, but instead is performed only when needed. However, synchronization is not free, as additional instructions are needed to coordinate the clusters. This section evaluates the advantages and disadvantages of loose coupling on the MAP chip compared to a VLIW emulated by the MAP's tightly coupled mode which enforces lock-step synchronization across the clusters. A simple synthetic benchmark is used to determine the benefit of slip in a program with unpredictable latencies. The cores of two applications are then examined to quantify the effect of slip and the overhead of explicit synchronization.

5.3.1 Synthetic Benchmark

Figure 5.5 shows a synthetic benchmark program in which each of 3 clusters sum the contents of a series of memory locations. The contents of the address increment register `i6` can be set to make the loads either hit or miss in the cache. When executed with lock-step instruction synchronization,

Cluster 0	Cluster 1	Cluster 2
`_lp_c0:`	`_lp_c1:`	`_lp_c2:`
`instr memu ld i5, i7;`	`instr;`	`instr;`
`instr ialu add i10, i7, i10;`	`instr;`	`instr;`
`instr ialu lea i5, i6, i5;`	`instr memu ld i5, i7;`	`instr;`
`instr;`	`instr ialu add i10, i7, i10;`	`instr;`
`instr;`	`instr ialu lea i5, i6, i5;`	`instr memu ld i5, i7;`
`instr;`	`instr;`	`instr ialu add i10, i7, i10;`
`instr;`	`instr;`	`instr ialu lea i5, i6, i5;`
`instr ialu add i8, #1, i8;`	`instr ialu add i8, #1, i8;`	`instr ialu add i8, #1, i8;`
`instr ialu ile i8, i9, cc0;`	`instr ialu ile i8, i9, cc0;`	`instr ialu ile i8, i9, cc0;`
`instr ialu ct cc0 br _lp_c0;`	`instr ialu ct cc0 br _lp_c1;`	`instr ialu ct cc0 br _lp_c2;`

Figure 5.5: Sequential memory access program.

Cluster 0	Cluster 1	Cluster 2
`_lp_c0:`	`_lp_c1:`	`_lp_c2:`
`instr memu ld i5, i7;`	`instr memu ld i5, i7;`	`instr memu ld i5, i7;`
`instr ialu add i10, i7, i10;`	`instr ialu add i10, i7, i10;`	`instr ialu add i10, i7, i10;`
`instr ialu lea i5, i6, i5;`	`instr ialu lea i5, i6, i5;`	`instr ialu lea i5, i6, i5;`
`instr;`	`instr;`	`instr;`
`instr;`	`instr;`	`instr;`
`instr;`	`instr;`	`instr;`
`instr;`	`instr;`	`instr;`
`instr ialu add i8, #1, i8;`	`instr ialu add i8, #1, i8;`	`instr ialu add i8, #1, i8;`
`instr ialu ile i8, i9, cc0;`	`instr ialu ile i8, i9, cc0;`	`instr ialu ile i8, i9, cc0;`
`instr ialu ct cc0 br _lp_c0;`	`instr ialu ct cc0 br _lp_c1;`	`instr ialu ct cc0 br _lp_c2;`

Figure 5.6: Overlapped memory access program.

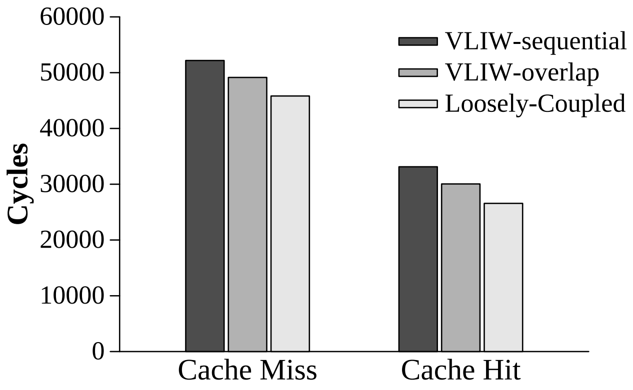


Figure 5.7: The effect of slip among instruction streams. A better VLIW schedule to overlap memory references improves performance 6–9%, while removing lock-step synchronization entirely results in a 12–20% improvement.

---

none of the memory latencies are overlapped, and each load must wait until the previous load completes. Figure 5.6 shows a similar program, in which the memory references are scheduled to execute in parallel in the same instruction and are overlapped. The null instructions are maintained so that the only difference between the sequential and overlapped schedules is the waiting time. Each loop is run for 1536 iterations and the experiment is run twice, once in which all memory references hit in the cache, and again with all references cache missing.

Figure 5.7 shows the effects of overlapping the memory latencies. VLIW-sequential is the lock-step execution of the sequential program and VLIW-overlap is the lock-step execution of the overlapped program. Loosely-coupled uses the normal mode of the MAP chip in which no implicit synchronization is enforced across the clusters and executes the same program as VLIW-sequential. Compared to VLIW-sequential, the better scheduling in VLIW-overlap results in a 6% speedup for cache misses and a 9% speedup for cache hits when memory latencies on different threads are overlapped. Loosely-coupled performs 12% faster for cache misses and 20% faster for cache hits than VLIW-sequential because the lack of synchronization allows overlap not only of memory operations, but also of other loop overhead such as instruction fetch unit stalls. In addition, the slip skews the load instructions so that fewer conflicts to the on-chip cache banks occur. The improvement from overlapping the memory latencies is smaller when the references miss in the cache, as each cluster must spend a greater fraction of its time waiting.

Program	Total Cycles	CBAR	Remote Write	GCC broadcast
MG-core	147161	5498	12115	2041
CG-core	277752	5654	10691	2041

Table 5.1: Intercluster interactions in MG-core and CG-core with explicit synchronization.

5.3.2 Application Kernels

Two application kernels are used to evaluate execution of instruction-level parallelism using both lock-step and explicit synchronization. MG-core is the relaxation subroutine of a multigrid benchmark, consisting of a triply nested loop that computes a 27-point weighted sum on all points in a 3-dimensional 12×12×12 space. CG-core is the primary subroutine of a 3-dimensional conjugate gradient that consists of a triply nested loop implementing a wavefront of computation across the diagonal of a 12×12×12 cube. Each kernel was hand coded and optimized for a single MAP cluster (SEQ), and was hand-scheduled for three MAP clusters using the intercluster communication and synchronization mechanisms. One version of the three cluster program has all of the explicit synchronization required for loosely coupled execution (Loosely Coupled), while another uses implicit lock-step synchronization (VLIW).

Table 5.1 shows the total cycles required to execute the two application kernels in Loosely Coupled mode, and includes the number of intercluster interactions required for explicit synchronization. On average, MG-core synchronizes using the cluster barrier every 27 cycles, but the clusters communicate every 10 cycles. Likewise, CG-core synchronizes every 49 cycles, and clusters communicate every 22 cycles. The interaction frequencies show that the data flow graph for each kernel contains somewhat independent instruction sequences that can be placed on separate clusters. In addition, the optimization of emptying multiple registers at a given cluster barrier is evident by the ratio 2.3–2.5 communications per barrier.

Figure 5.8 shows the breakdown of the execution time for both MG-core and CG-core. The number of cycles spent executing instructions is shown by *Execute*. Stalls due to instruction fetch, including both instruction cache misses and branch penalties, are signified by *IFU*. Stalls due to memory latencies, including both cache hits and misses, and data dependent instruction latencies are included in *Memory*. *Sync* comprises the time spent while waiting at a barrier or waiting for

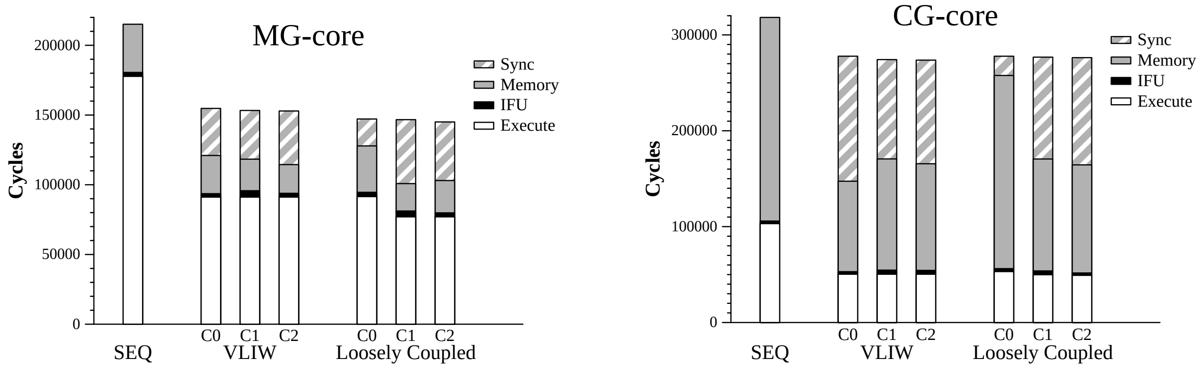


Figure 5.8: ILP cycle breakdown for MG-core and CG-core.

data to be delivered from a remote cluster.

For MG-core, lock-step VLIW synchronization yields a speedup of 28% while Loosely Coupled speeds up the kernel by a total of 32%. A perfect factor of three speedup on the parallel versions is not attained, due in part to overhead and contention. Both VLIW and Loosely Coupled execute more total instructions than SEQ because some code, including as branches and address calculations, is replicated on multiple clusters. The cumulative time spent waiting for memory requests on all clusters is larger than SEQ for both parallel versions. More memory instructions are required, due to replication, and there is more contention for the shared on-chip cache. As a result, the sequential version spends 83% of its time executing useful instructions, while VLIW and Loosely Coupled spend only 52–62%. The overall speedup of the parallel models is also limited by imperfect parallelization, as neither loop unrolling nor software pipelining was used to scheduling the inner loop. Since these techniques require many registers to hold variables from multiple iterations, the MAP chip is somewhat disadvantaged with its limited register set. In Loosely Coupled, cluster 0 does slightly more work, which results in more time spent waiting for synchronization on clusters 1 and 2. In VLIW, all execution units execute the same number of instructions, but some of these are NOPs where parallelism is not able to fill instruction slots in all of the clusters. Finally, due to its lock-step execution, VLIW spends slightly more time synchronizing than Loosely Coupled, resulting in a 5% performance advantage for the explicitly synchronized program.

For CG-core, the performance advantage of instruction-level parallelism is much smaller, only 13% for both VLIW and Loosely Coupled. In its inner loop, this kernel contains accesses to multidimensional arrays, requiring several memory references for each array element, and a 20 cycle latency floating-point divide. As a result, the sequential version spends only 32% of its cycles executing instructions, and the time to complete the program is limited by the memory and pipeline latencies. Both parallel versions are load balanced and the sum of the instructions executed across all clusters is 45% more than SEQ. However, the load instructions and the divide in the inner loop of cluster 0 cause the memory and synchronization overheads to skyrocket. In VLIW, each cluster spends as much as 41% of the total execution time waiting for the other clusters. Although the divide is executed in cluster 0, the time spent waiting for it to return is incorporated into *Sync*, as some of the divide latency is covered by time waiting for other clusters. For Loosely-Coupled, the divide latency is a part of *Memory* because cluster 0 does not need to wait for the other clusters before using the divide result. The other component of the long *Memory* time of cluster 0 is contention for the Memory Switch and the on-chip cache.

5.4 Summary

This chapter demonstrates the viability of exploiting instruction-level parallelism on the MAP chip. The communication and synchronization mechanisms are fast and are easily exposed to a compiler. Orchestrating the communication between clusters is straightforward and can be accomplished using a barrier and an empty instruction to guarantee that the synchronization will occur on data delivery. The overhead resulting from inserting the synchronization instructions is low, and can be reduced through a variety of optimizations, including emptying multiple registers at a given barrier. These intercluster mechanisms on the MAP are scalable as only the intercluster communication network and global barrier busses would require modification in a system with more clusters.

The performance of explicitly synchronized ILP is competitive with implicitly synchronized VLIW architectures. Loosely coupled clusters allow instruction streams to slip relative to one another and to overlap, instead of sequentialize, their long latency operations. This slip is also beneficial when the instruction streams need to access a limited resource, such as the on-chip cache. The different streams will naturally align themselves so that they do not all access the resource at the same time. Explicitly synchronizing the streams only when necessary is not only simpler to implement in hardware, but also faster (up to 5%) on the application cores than implicitly synchronized streams, even when counting the overhead for executing the synchronization instructions.

Chapter 6

Thread-Level Parallelism

In traditional parallel computers, the communication latencies between threads on different processors can be as high as 100 to 1000 cycles. Each processing node is typically a commodity microprocessor which is connected to a custom network through the memory interface pins. For each transfer between processors, a substantial penalty must be paid to traverse the deep on-chip memory hierarchy just to get to the network. The software overhead to receive and synchronize with an incoming transfer can also be quite large. Because of these high overheads, most parallel applications use only coarse-grain threads with many thousands of cycles between interactions.

The MAP chip changes the relationship between computation and communication in a single chip multiprocessor. Fast interaction mechanisms are integrated directly into the processor pipeline, enabling threads on different clusters to communicate and synchronize in a single cycle. Fine-grain threads can use these mechanisms to parallelize applications that require frequent communication, and those that have small data sets and large computational needs, such as electronic circuit simulation.

This chapter focuses on the effectiveness of the MAP's on-chip communication and synchronization mechanisms at exploiting on-chip fine and coarse-grained parallelism on a range of scientific applications. The coarse-grained parallelism comes from the outer loops of the applications, mainly by dividing the data set across the processors and assigning independent loop iterations to them. The fine–grain parallelism is added by examining the inner loops of the applications to find subroutines and expressions that can be executed concurrently. Section 6.1 discusses the encapsulation and invocation of parallel tasks using a parallel procedure call (PPC). Section 6.2 compares register

C code	**Assembly**	
`fork(eval_node, cur_node, i);`	`instr memu empty i14;`	`/* for join */`
	`instr ialu mov i6, h1.i6;`	`/* cur_node */`
	`instr ialu mov i7, h1.i7;`	`/* i */`
	`instr ialu mov i8, h1.i3;`	`/* function pointer */`
••• compute •••	••• compute •••	
`join(cost);`	`instr ialu mov i14, i6;`	`/* wait for result */`

Figure 6.1: Parallel procedure call fork and join.

communication to memory communication between threads, using a fine-grain synthetic benchmark. Section 6.3 details the set of parallel applications used in this study to explore the benefits of the MAP's integrated communication and synchronization mechanisms. Sections 6.4 examines fine-grain inner-loop parallelizations of the applications, while Section 6.5 looks at more traditional outer-loop methods of exploiting concurrency.

6.1 Parallel Procedure Call

In this study, thread level parallelism is exploited using a parallel procedure call (PPC) which is similar to a *future* [KHM89]. A master thread runs on cluster 0 and controls the flow of the program, while slave threads run on clusters 1 and 2 and wait to be forked by the master. When the master encounters a parallel procedure call, it forks the procedure to a slave thread and continues executing. The slave executes the procedure and returns the value to the master. At a join, the master must wait until the slave has completed its task.

Figure 6.1 shows the C code and the corresponding assembly code that runs on the master thread during a parallel procedure call. The `fork` macro expands into a sequence of assembly language instructions to transfer the pointer to the `eval_node` function and the two arguments to cluster 1. Register `i14` is first emptied to prepare for the synchronization at the join. The procedure's parameters (`cur_node` and `i`) are transferred into cluster 1's argument registers (`i6`, `i7`), matching the compiler's function calling protocol. Finally, the master delivers the `eval_node` function pointer to cluster 1 in register `i3`. When the master reaches the join, it resynchronizes with

```
slave_loop:                      /* slave code */
    instr ialu jmp i3;           /* wait for new IP, then jump */
    instr;                       /* delay slot */
    instr;                       /* delay slot */
    instr ialu lea i1, #4, i4;   /* calculate return IP */

••• compute function •••

slave_return:                    /* slave code */
    instr ialu br slave_loop
          memu mbar;             /* commit all memory accesses */
    instr ialu empty i3;         /* empty IP register */
    instr ialu mov i6, h2.i14    /* transfer result to master */
    instr;                       /* delay slot */
```

Figure 6.2: Slave standby handler for parallel procedure call.

the slave, by reading `i14`. If the slave has already completed, `i14` will be full, and the synchronizing move operation will execute immediately. Otherwise the master will stall until the slave completes and delivers the function's return value.

Prior to any parallel procedure call, the master starts a standby handler in the slave thread on cluster 1, giving it a stack pointer and a heap pointer. These remain persistent throughout execution of the program, and are not passed from the master to the slave at every call. The assembly code for the standby handler, shown in Figure 6.2, waits for `i3` to be marked full. When the master writes the function pointer into `i3`, the slave handler jumps to the pointer, and in the last branch delay slot, computes the function return pointer and places it in `i4`. When the parallel procedure call completes, the slave handler resumes control. It first executes an `mbar` instruction that waits for any outstanding memory references so that no registers are inadvertently overwritten. The handler then empties `i3` for the next invocation and passes the function's return value from `i6` to the master's synchronization register, `i14`. Finally it returns to `slave_loop` to wait for the next call.

The same master/slave protocol can also be implemented using only the local on-chip cache to transfer data between threads. At the fork, the master empties the synchronization bit associated with a memory location in preparation for the return value. The arguments and function pointer are written to memory and the locations are marked full to signal to the slave. The slave spins until the memory location holding the function pointer is full, loads the arguments into registers for the

```
for(i=0; i<global_num; i++) {
  res1 = sub_loop(sub_num);
  res2 = sub_loop(sub_num);
  res3 = sub_loop(sub_num);
  total_res = res1 + res2 + res3;
}
```

Figure 6.3: Pseudocode for synthetic benchmark. Each instance of `sub_loop` is executed on a different cluster for the parallel measurements.

function call, and jumps to the forked procedure. Upon return, the slave handler first empties its input argument memory locations and stores the return value into the designated return location, marking it full. At the join, the master spins on the return location until the slave has completed. The experiments throughout this chapter compare the performance of communicating between the master and slave using registers or memory.

6.2 Synthetic Benchmark Study

A synthetic benchmark is first used to further examine the effect of the interthread register and memory communication latencies of the MAP chip. With fast mechanisms for thread invocation and communication, extremely fine–grain thread parallelism can be exploited. If the mechanisms are slower, as is on-chip memory communication, fine–grain parallelism can still be exploited, but the granularity of the tasks must be larger. The synthetic benchmark, shown schematically in Figure 6.3, consists of a single loop containing three function calls, each of which may be run in parallel. Varying `sub_num` changes the time to execute each of the function calls (affecting both grain size and problem size), while `global_num` dictates the number of outer loop iterations.

In the parallel versions, the master thread invokes one instance of `sub_loop` on each of the neighboring clusters, using a parallel procedure call (PPC), and executes the third instance itself. The slave threads operate in standby mode waiting to be signalled by the master. When a slave completes, it returns its result to the master, which performs the join before beginning another iteration of the outer loop. Each of the different versions is implemented in hand generated assembly code and is itemized in Table 6.1.

Synthetic Program	Description
SEQ	Baseline sequential
PPC_REG	Parallel with register synchronization
PPC_MEM	Parallel with memory synchronization

Table 6.1: Synthetic benchmark execution models.

6.2.1 Granularity

The communication frequency and cost has a substantial impact on the effectiveness of parallelization over multiple clusters. When task granularity is small, communication between the master and the slaves is frequent. The MAP's fast register-register communication is required in order to see performance improvements when fine grain tasks are used. Figure 6.4 shows the time for one iteration of the outer loop as a function of the granularity of the inner loops, normalized to the sequential execution time. The granularity, in turn, is a function of the number of inner loop iterations, which is varied from 0 to 30. When no iterations are executed within `sub_loop`, the procedure call overhead and test inside the slave function still requires 19 cycles. Each increment in grain size corresponds to an additional loop iteration in each subroutine. At the smallest grain size, PPC_REG is 1.6 times faster than SEQ, while PPC_MEM is 1.2 times slower, due to the additional cost for the master to store the arguments into memory and for the slave to retrieve them. Both PPC_REG and PPC_MEM improve substantially as more work is done inside the inner loops. However, their execution time relative to sequential flattens out above granularities of 110 cycles as they approach the maximum of 3 times speedup. PPC_REG still maintains an advantage over PPC_MEM, but that diminishes as the granularity increases.

Scaling the problem size may be an acceptable method of increasing the grain size and reducing communication overhead. However, if the problem to be solved is a fixed size, then the speedup that can be attained through parallelism is limited directly by the cost for threads to interact. Adding more processors decreases the grain size, but the interaction overhead overwhelms any parallelism benefit. The second experiment examines the total execution time of a constant problem size as a function of variable grain size. The number of outer loop iterations is varied from 1 to 100, while the number of iterations in each `sub_loop` call is varied from 100 to 1, with the sum of all of the inner

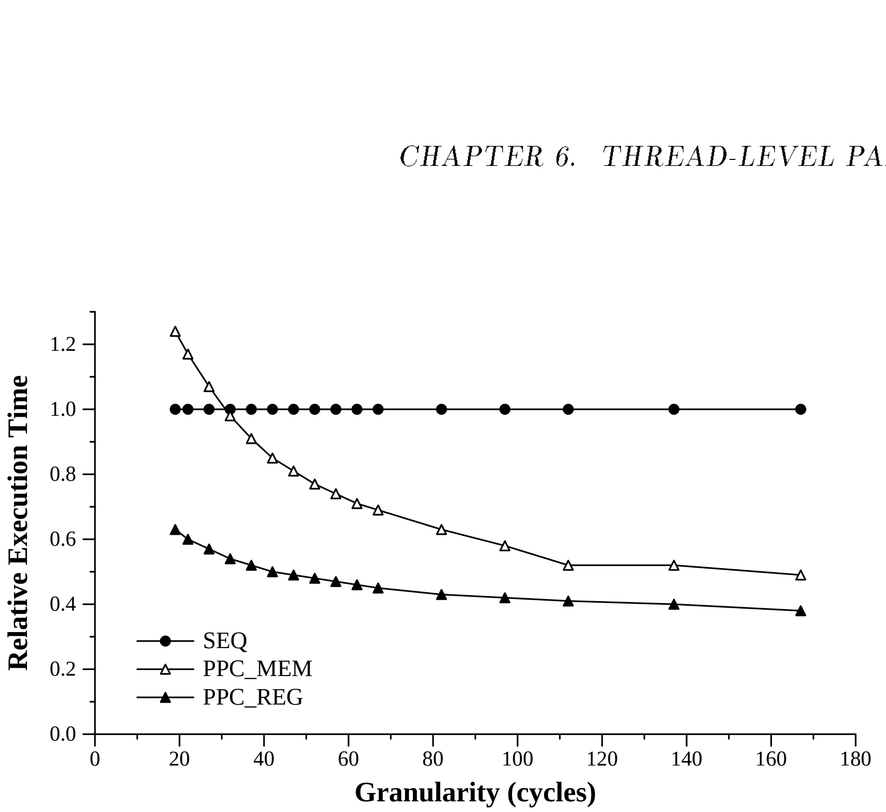


Figure 6.4: Outer loop iteration time as a function of inner loop grain size, normalized to sequential. At the smallest grain size (19 cycles of work in slave threads) PPC_REG is 1.6 times faster than SEQ. PPC_MEM becomes faster than SEQ at grain sizes of greater than 30 cycles.

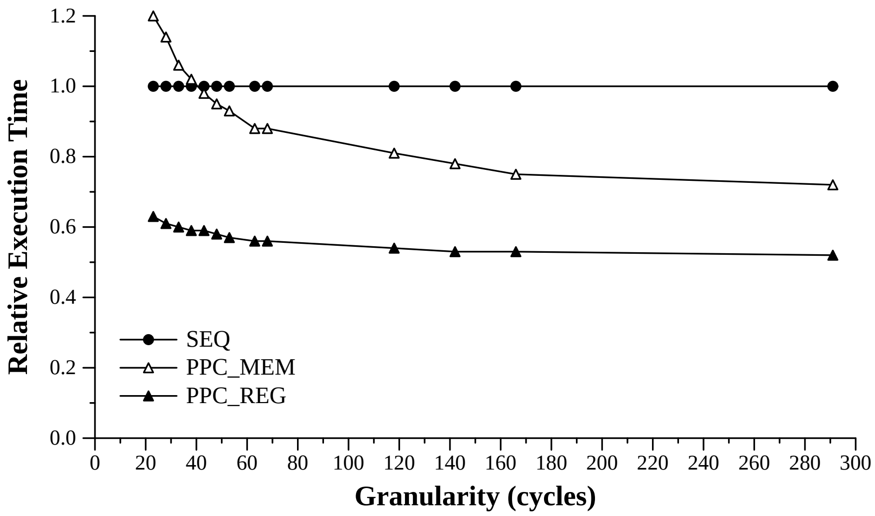


Figure 6.5: Normalized execution time for a fixed problem size as a function of grain size. At low granularities, the high overhead in PPC_MEM results in no speedup, while it approaches PPC_REG as the granularity increases.

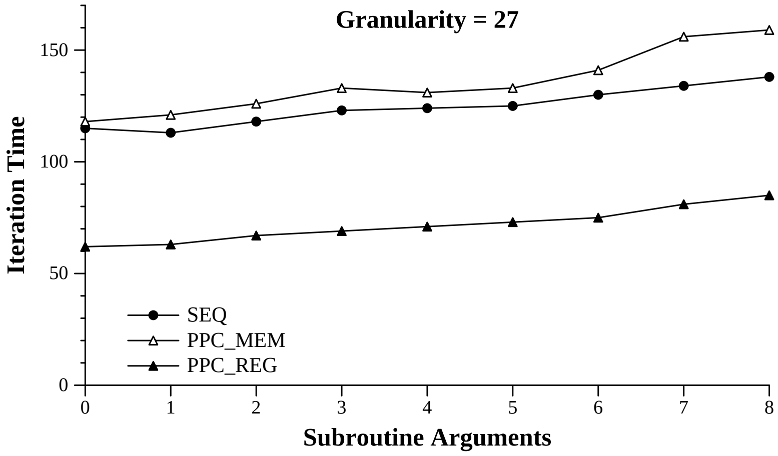


Figure 6.6: Outer loop iteration time as a function of the number of arguments passed from master to slave with a grain size of 27 cycles. PPC_REG requires two additional cycles per argument, one cycle for each slave thread. PPC_MEM requires almost four cycles per additional argument, two cycles for each slave.

loop iterations fixed at 300. Figure 6.5 displays the total execution time normalized to the execution time of SEQ for each of the experimental versions. When the granularity is small (ie. the number of `sub_loop` iterations per outer loop iteration is small), the sequential overheads, including loop and thread call overhead, dominate for each of the parallel versions. As a result of the lower parallel procedure call and return overheads, PPC_REG is almost twice as fast as PPC_MEM. Because the register communication mechanisms are so fast, the relative execution time of PPC_REG suffers very little as grain size decreases. Using more on-chip processors to increase parallelism in fixed size problems is a viable option as the communication overhead is not substantial. At granularity greater than 50 cycles, PPC_REG levels out at 1.8 times as fast as SEQ. PPC_MEM reaches a limit of 1.4 times as fast as SEQ at the largest grain size. Even with as large a grain size as possible and fast communication, a perfect factor of three speedup cannot be achieved due to the unparallelized control overhead of the program.

6.2.2 Argument Count

The cost to invoke a procedure on a remote cluster can be divided into the time to invoke a null function and the incremental time to transfer the function's arguments. Figure 6.6 shows the result of an experiment that examines the relative overhead of starting a thread versus passing it arguments. In this experiment, the program executes two inner loop iterations for each outer loop iteration, which corresponds to a grain size of 28 cycles. The number of arguments passed between the master and the slaves is varied from 0 to 8. For PPC_REG, approximately two cycles are required for each additional argument, one cycle for each slave thread. PPC_REG experiences a slight upturn when going from 6 to 8 arguments. Since only six arguments can be passed in registers, arguments 7 and 8 must be passed on the stack, which requires additional address calculation and memory operations. PPC_MEM requires almost four cycles per additional argument, two cycles for each slave to perform an address calculation and a store. However, the most significant component of the overhead for PPC_MEM is clearly in starting the slave threads. When zero arguments are passed, PPC_MEM requires 56 more cycles (28 cycles per slave thread) for each outer loop iteration. Combining these results with the thread creation experiments in Chapter 4 gives a simple model for cost of starting a remote thread. Using register communication takes 10 cycles for invocation and 1 cycle for each parameter. Creating a thread using the on-chip cache requires 36 cycles for invocation and 2 cycles for each parameter.

6.3 Parallel Applications

As shown in the synthetic benchmark, the communication and synchronization mechanisms of the MAP chip allow threads to be invoked quickly and communicate efficiently with one another. The next sections explore the utility of these mechanisms in applications using inner–loop and outer–loop parallelism. Inner–loop parallelism is discovered by examining the inner loops of the applications to find subroutines and expressions that can be executed concurrently. Outer–loop parallelism comes from the outer loops of the applications, mainly by dividing the data set across the processors and assigning independent loop iterations to them.

The applications in this study are compiled using MMCC, the MAP C compiler, which is a derivative of the Multiflow C compiler [LFK+93]. The compiler is able to generate a schedule for all three processor clusters from a sequential program. However, for the experiments reported in this paper, MMCC produces sequential single cluster code, using all three execution units within a cluster as a 3 instruction wide statically scheduled machine. MARS, the runtime system for the M-Machine, is used to provide system services, including memory allocation, terminal I/O, and file

Benchmark	Description	Source	Problem Size
MG	Multigrid	Alewife [CLB+96]	64–2744 doubles
FFT	Fast-Fourier Transform	Alewife [CLB+96]	4–128 complex doubles
EM3D	Electromagnetic simulation	UC Berkeley [CDG+93]	6–30 node pairs
CG	Conjugate Gradient	Yeung [YA93]	27–1728 doubles
EAR	Cochlea simulation	Spec92 [SPE92]	10–100 doubles

Table 6.2: Application benchmark summary.

I/O [Gur95]. While both MARS and the MAP support virtual memory, all experiments were run in a physical address space, with no translation lookaside buffer (TLB) miss handling required.

Inner-loop parallelism is implemented by encapsulating independent expressions and function calls inside procedures, which are then forked from a master thread to a slave thread using a parallel procedure call. Outer-loop parallelism is explicit in the applications and exploits concurrency at outer loops with data dependent phases separated by barriers. The applications are detailed below and summarized in Table 6.2.

MG is a solution to a 3D Poisson partial differential equation. It is based on the multigrid kernel from the NAS parallel benchmarks and SPEC95. The outer–loop parallel code assigns a subset of the three dimensional data space to each processor, and the different computation phases are separated by barriers. For inner-loop parallelism, two versions with different thread granularities are used. **MG-E** parallelizes only the contents of the innermost loop of the `Relax` (relaxation) subroutine by placing independent arithmetic expressions on different clusters. **MG-L** parallelizes the `Relax` subroutine by executing different iterations of the inner loop concurrently. The volume of the cubic space to be solved is varied from 64 to 2744 double precision floating-point numbers.

FFT solves a 1-dimensional partial differential equation using forward and inverse FFTs. With outer-loop parallelism, each processor is assigned a subsection of the array and computes one level of the butterfly on its subarray before placing the result into a temporary array. After a barrier, each processor copies its section of the temporary array to the global array and barriers again. Inner-loop parallelism is extracted by executing inner-loop expressions and subroutines concurrently. The size of the input array is varied from 4 to 128 complex double precision floating-point numbers.

EM3D simulates electromagnetic interactions and consists of alternating phases of computation on `e-nodes` and `h-nodes`. To exploit outer–loop parallelism, each processor is assigned a subset of the nodes and at each timestep computes new values for its `e-nodes`, barriers, computes new values for its `h-nodes`, and barriers again. Inner–loop parallelism is exploited by computing all of the interactions for a given node concurrently. Each of the MAP's clusters is assigned a subset of the connecting nodes. After computing its subset's local contribution, a cluster delivers the result to cluster 0 to be accumulated with the results from all three clusters. The EM3D initialization routines are not included in any results. The problem size is varied from 6 `e-node/h-node` pairs to 30 pairs, and each node is connected to 5 other nodes.

CG implements a Modified Incomplete Cholesky Conjugate Gradient method for 3-D boundary value problems. The outer–loop parallelism profile forms a wavefront across the central diagonal of a cube that forms the problem space. At each iteration, a processor computes its assigned portion of the wavefront and then executes a barrier. The inner–loop version only parallelizes the innermost computation loop, which consists of a set of arithmetic operations combined with boundary checks to handle corners, edges, and faces of the cube. The volume of the cube is varied from 27 to 1728 double precision floating-point numbers.

EAR, from the SPEC92 suite, simulates the propagation of sound in the human cochlea (inner ear). The application consists of a sequential outer loop, containing a sequence of 12 parallel inner loops. Iterations of the outer loop must execute sequentially, and iterations from different inner loop nests cannot be run concurrently. Thus, EAR consists only of inner-loop parallelism. Ten time steps are simulated and the size of the input vector is varied from 10 to 100 double precision floating-point numbers.

6.4 Inner-Loop Parallelism

This section examines the task granularity of inner-loop parallelism and compares the effectiveness of register and memory communication methods in exploiting it. The applications in this study have parallel task lengths that are as short as 70 cycles. For the memory and communication latencies in the MAP chip, register communication is approximately 15% faster than using the on-chip cache, and overall speedups of up to 2.4 times can be achieved using only inner-loop parallelism.

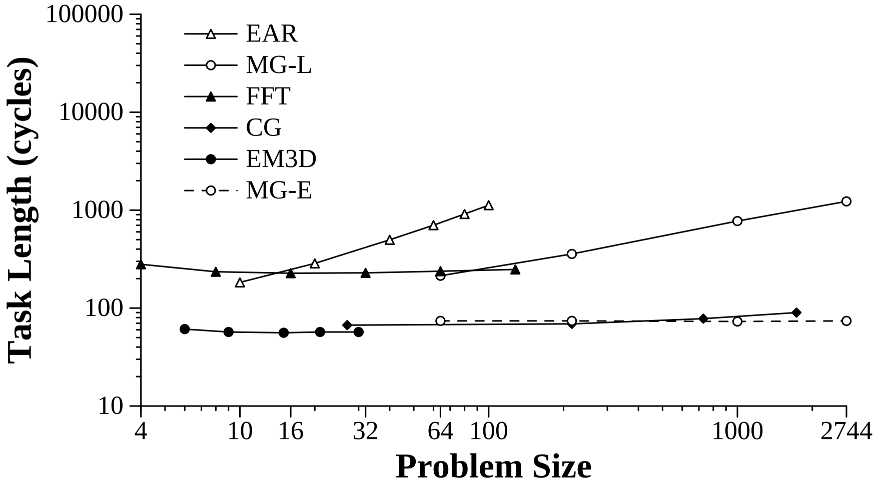


Figure 6.7: Inner–loop task length versus problem size. The task length is the average time for the slaves to execute their parallel tasks. FFT, CG, EM3D, and MG-E exploit expression oriented parallelism in the inner loop, with granularity independent of problem size. EAR and MG-L exploit inner loop level parallelism and have granularities that increase with problem size.

6.4.1 Task Granularity

The granularity for inner-loop parallelism exploited using the fork/join model is defined as the average time for a slave thread to execute a parallel task. Figure 6.7 shows the inner-loop task granularity for all five applications as a function of problem size on a log–log plot. The problem sizes are indicative of the relative amount of work for each benchmark, but cannot be compared across different applications. For EM3D, CG, and MG-E, the task granularity is less than 100 cycles, with EM3D as low as 70 cycles. FFT also has a constant granularity curve, but the average task length is approximately 280 cycles. In order for the slaves to provide a benefit, the overhead for forking and joining must be low. For these four applications, the granularity remains essentially constant, regardless of problem size. Each application is parallelized by partitioning expressions and subroutines across the clusters. Consequently, the overall work within the inner loop is independent of the size of the data set. For example in EM3D, since each node of the data graph is connected to exactly five other nodes, so that regardless of the total number of nodes the amount of work in the inner loop does not increase with problem size.

The inner-loop parallelism for EAR and MG-L is exploited by running loop iterations concur-

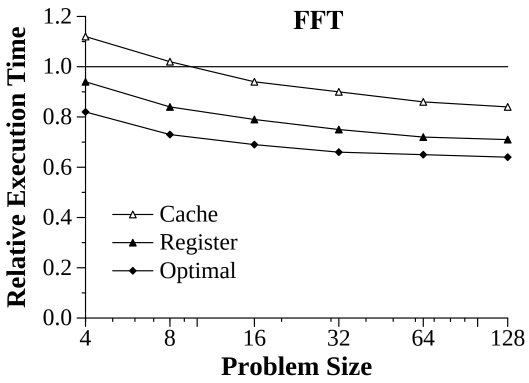


Figure 6.8: Normalized execution time versus problem size for Inner–Loop FFT. The higher interaction latencies of **Cache** cause it to be consistently 15% slower than **Register**.

rently. Increasing the data set size results in more loop iterations and more work for each slave thread. Thus the task granularities for EAR and MG-L start out at about 200 cycles for the smallest problem sizes, and increase linearly to about 1000 cycles at the largest problem sizes. If the data sets were large enough, both of these applications could be reclassified as coarse grain. The application granularity is not necessarily inherent, but rather it is a function of the method of parallelization. However, with efficient communication mechanisms, fine grain parallelism can be extracted using methods that have been previously infeasible.

6.4.2 Communication Comparison

The cost to communicate between the master and slaves has a direct impact on the performance of inner-loop parallelizations. Figure 6.8 shows the execution time for FFT across all of the problem sizes, normalized to the sequential execution time on a single MAP cluster. The **Cache** line shows the relative execution time when using the on-chip cache to communicate between the master and the slave threads, while **Register** uses the MAP's register communication mechanisms. **Optimal** is a measure of the execution time if all of the communication between the master and slaves occurs instantaneously. All three versions of the application improve relative to the sequential code as the problem size increases, with a 1.4 times speedup for **Register** at the largest data set. This is due to the application spending a greater fraction of the execution time in the parallel sections of the program and less time in the sequential sections. For all problem sizes, register communication is

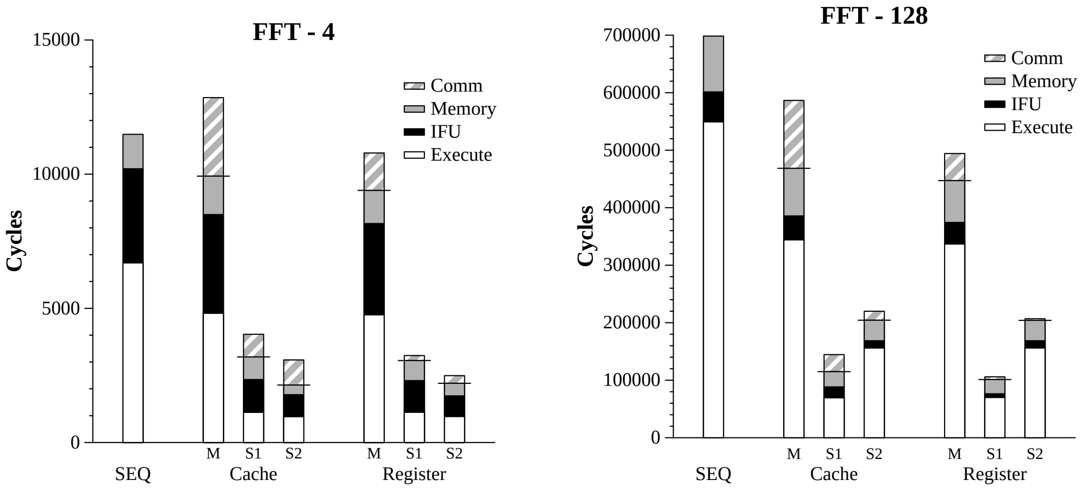


Figure 6.9: Cycle breakdown of execution time for Inner–Loop FFT.

approximately 15% faster than using the on-chip cache. However, the speedup of using multiple clusters is limited by the amount of parallelism in the application and the method of extracting it, rather than by the communication overhead. Even when communication is free (**Optimal**), only an additional 10–15% performance improvement is attained. The speedup for FFT is minimal at small problems sizes and improves as the size of the data set increases. With a 4 element input vector, FFT executes only 6 iterations of its inner loop. The total execution time is dominated by the sequential component of the application.

Figure 6.9 illustrates these limitations by decomposing the running time of FFT with problem sizes of 4 and 128 into execution and overhead components. The cycle breakdown is shown for a single cluster (**SEQ**) as well as for the parallel versions using the on-chip cache or registers to communicate. For the parallel versions, both the master (**M**) and two slaves (**S1**, **S2**) are shown. The master runs on cluster 0, while the slaves run on clusters 1 and 2. The running time is broken down into the cycles spent executing instructions (**Execute**), waiting for the instruction fetch unit and instruction cache (**IFU**), waiting for data from the memory system (**Memory**), and communicating between the clusters (**Comm**). The primary factor that limits the overall speedup is the load imbalance seen in the parallel versions, as there is significant sequential work performed only by the master. Using a longer input vector shows better load balance because more total time is spent inside the inner loop. However, the performance is still limited, not by the communication

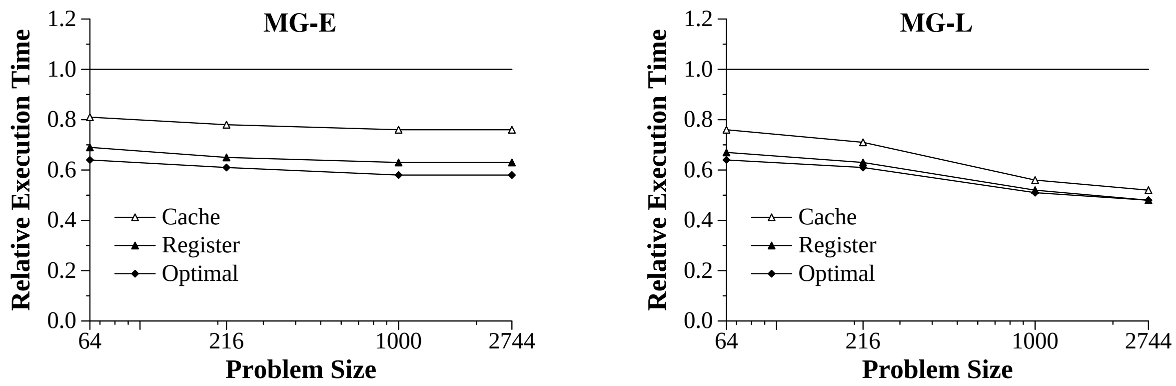


Figure 6.10: Normalized execution time versus problem size for Inner–Loop Multigrid.

mechanisms, but instead by the efficiency of the parallelization. The communication overhead using registers is less than one half that of using the cache, but the overall impact on performance is only 15%.

Similar results are seen in Multigrid, as shown in Figure 6.10. The relative execution time for MG-E stays constant regardless of problem size, because only the contents of the inner loop are parallelized. In this case, Register is also 15% faster than Cache, but it is much closer to Optimal than in FFT. For MG-L, the relative execution time compared to sequential improves as the problem size gets larger, achieving a factor of two speedup at the largest problem size. The increase in speedup at larger problem sizes is due to both more time spent in the inner loops and less frequent communication. As shown above, the task granularity for MG-L increases with problem size since the parallelism comes from executing loop iterations concurrently. Thus the communication frequency decreases and performance of Cache approaches that of Register as the problem size increases. At the smallest problem size, Register is 12% faster than Cache, while at the largest problem size, it only is 8% faster.

These effects can be seen in the cycle breakdown of MG shown in Figure 6.11, for problem sizes of 64 and 2744. The results for using expression oriented parallelism (MG-E), are shown in Cache-E and Register-E. At the smaller problem size, MG-L and MG-E have nearly the same execution time, with the coarser grain parallelization having a slight advantage. At the larger problem size, MG-L shows better load balance across the clusters and less communication overhead. For MG-E,

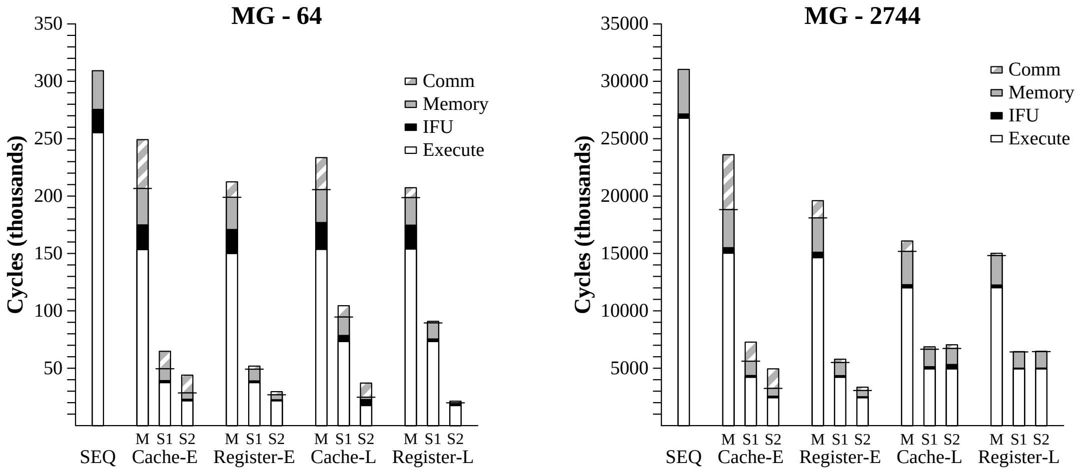


Figure 6.11: Cycle breakdown of execution time for Inner–Loop Multigrid.

register communication has substantially less overhead than memory communication, while MG-L shows a much smaller difference between registers and memory, due to a lower communication frequency. The other applications exhibit similar behaviors which can be seen in Appendix B.1.

6.5 Outer-Loop Parallelism

Outer–loop parallelism is exploited using the multiprocessor parallelizations of each of the applications, in which outer parallel loops are identified and executed concurrently on each of the three MAP clusters. The clusters communicate using the shared on-chip cache and can synchronize either through memory, or using the `cbar` instruction. Memory synchronization, using the memory synchronization bits of the MAP chip, requires threads to spin on locks until all threads have reached the barrier.

6.5.1 Task Granularity

Figure 6.12 shows the outer–loop task granularity on the same scale as the inner–loop granularity of Figure 6.7. Outer–loop task granularity is defined as the number of cycles spent between barriers. The outer-loop parallel tasks are much larger than inner-loop tasks and their granularity increases dramatically with data set size. The gap in grain size between the inner and outer loop

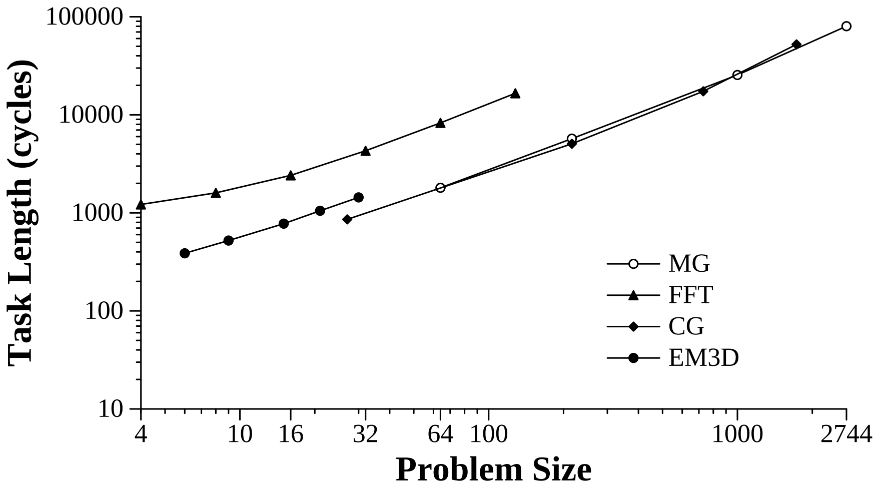


Figure 6.12: Outer–Loop task length versus problem size, where the task length is the average time between barriers.

---

parallelizations is more than a factor of 10 for EM3D, MG, and CG, even on the smallest problem size, and it widens to a factor of 550 at a problem size of 1728 for CG. FFT exhibits the narrowest range, with a factor of 6 at vector length 4, to a factor of 70 at vector length 128. The large task lengths of the coarse-grained applications stem from their original implementation on shared memory multiprocessors, with single word communication latencies requiring hundreds of cycles. Exploiting parallelism in the 70-200 cycle range would be infeasible with such high interaction costs.

6.5.2 Synchronization Comparison

The effect of this increasing granularity can be seen in Figure 6.13, which shows the execution time of FFT and MG as a function of problem size, normalized to the sequential execution time. **Cache** shows the execution time when the barrier is implemented using the on-chip cache, while **CBAR** shows the execution time when the barrier instruction is used. **CBAR** is equivalent to an optimal barrier since the `cbar` instruction is so efficient. Outer–loop parallelism results in shorter execution times than inner–loop, as more of the code is parallelized and the larger grain size requires less communication and synchronization. FFT improves from no speedup on a 4 element vector to 2.4 times speedup on a 128 element vector. MG improves from 1.5 to 2.7 times speedup as the problem size increases. The improvement in speedup is a direct result of both the increasing granularity

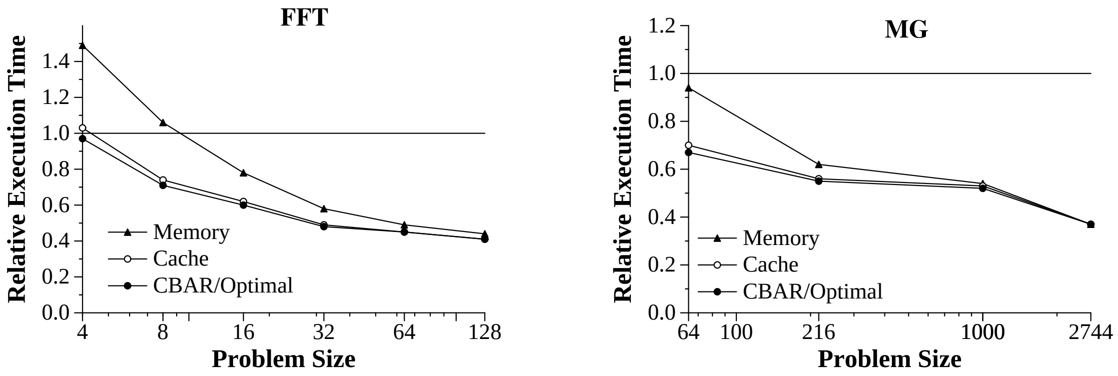


Figure 6.13: Normalized execution time versus problem size for Outer–Loop FFT and Multigrid. As problem size increases, the difference between synchronizing via off-chip memory (**Memory**), the on-chip cache (**Cache**) and the barrier instruction (**CBAR**) diminishes.

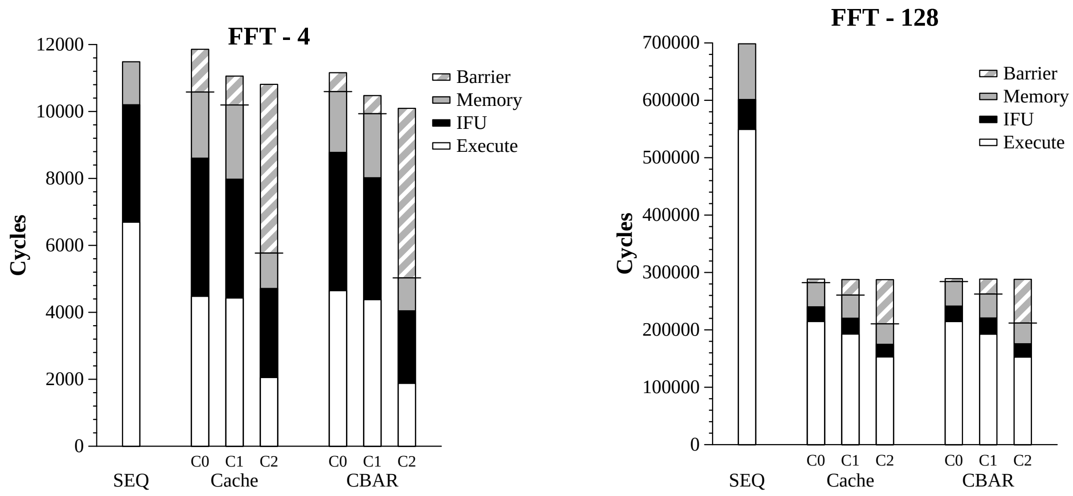


Figure 6.14: Cycle breakdown of Outer–Loop FFT.

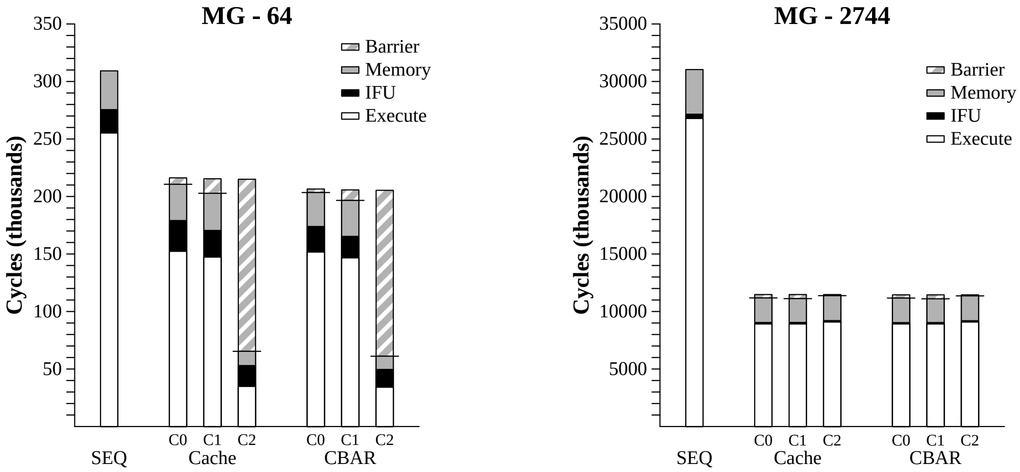


Figure 6.15: Cycle breakdown of Outer–Loop Multigrid.

and the larger fraction of time spent in the parallel sections as the problem size increases. Another consequence of the coarse granularity is that the performance of the fast barrier **CBAR** and the memory barrier **Cache** are practically indistinguishable. Since so much time is spent between synchronizations, the cost of the barrier is inconsequential.

Figure 6.14 shows the cycle breakdowns for FFT at problem sizes of 4 and 128. The Execute, IFU, and Memory represent the same components of running time as in the previous section, but **Barrier** is the time spent waiting for other clusters to reach a barrier. The sequential version uses only cluster 0, while the parallel versions use all three clusters (C0, C1, and C2). At a small problem size, a substantial load imbalance can be seen, which is nearly completely eliminated at problem size 128. The difference between Cache and CBAR, which is small at problem size 4, essentially disappears at problem size 128. The same effects can be seen for MG, as shown in Figure 6.15. A load imbalance is evident at the smaller problem size, as four iterations are spread over three clusters, leaving cluster 2 with little work to do. At problem size 2744, the load is balanced perfectly across the clusters, and the difference in synchronization costs is irrelevant. The results for the other applications are similar and are found in Appendix B.2.

The coarse grained applications see substantial speedups on relatively small problem sizes for two reasons. First, synchronization cost is low, even using memory locks, because all of the accesses are local. Second, all of the data for the threads is shared either in the on-chip cache or in local

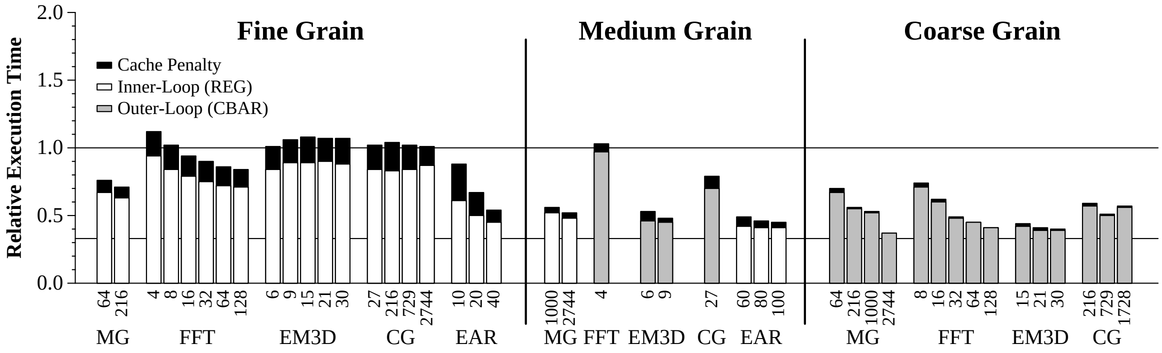


Figure 6.16: Normalized execution time of all 5 applications, including inner and outer–loop parallelization, across all problem sizes. The penalties for interacting using on-chip cache can be substantial, depending on the task granularity.

memory. However, in a traditional multiprocessor, the communication costs are significantly higher. Inter-node barriers are more expensive, and any shared data must be passed from node to node. The **Memory** curve in Figure 6.13 is intended capture some of the effect of additional synchronization cost by increasing the barrier overhead to 1000 cycles. If the grain size is small, as with the small problem sizes, the cost for long latency synchronization can cause substantial slowdowns, rather than speedups. As the problem sizes increase, the synchronization frequency decreases, which renders the synchronization cost irrelevant. However, in an actual system, the cost for threads to communicate with one another through off-chip memory is likely to be the limiting factor for performance.

6.6 Summary

Figure 6.16 summarizes the execution time for all 5 applications, with MG-L representing the inner-loop parallelization of Multigrid. The line at 1.0 indicates the sequential execution time, while the line at 0.33 is the lower bound on speedup using three clusters. The programs can be partitioned based on their task granularity into fine, medium, and coarse grain. Fine–grain tasks are typically less than 300 cycles, medium grain tasks are between 300 and 1500 cycles, and coarse grain tasks are greater than 1500 cycles. The task granularity is a function of the method of parallelization (inner–loop versus outer–loop), as well as problem size. The dark caps on the execution time

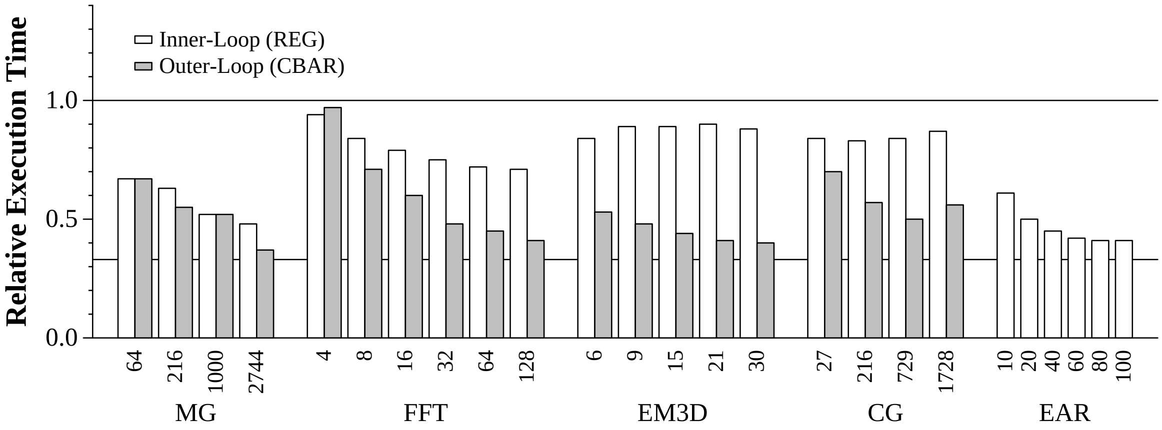


Figure 6.17: Normalized execution time for all applications comparing inner to outer loop parallelization.

---

bars signify the penalty for using the on-chip cache instead of the integrated communication and synchronization mechanisms of the MAP chip. As is evident from the graph, in order to exploit fine-grain tasks, the integrated mechanisms are a necessity. Medium grain tasks suffer only a small performance penalty when using the on-chip cache for communication and synchronization. Coarse grain tasks require no special mechanisms for synchronization since interaction frequency is small. As shown in Chapter 4, the gap between direct communication through registers and indirect communication through memory will increase dramatically in chips implemented in future process technologies. Consequently, to get the benefits of concurrency available using fine-grain threads, architectures must supply applications with fast and direct communication mechanisms.

When outer-loop parallelism is available, it generally yields faster execution times than inner-loop parallelism, as demonstrated in Figure 6.17. However, some applications such as EAR have no outer-loop parallelism and require finer grain parallelism and additional hardware support for communication and synchronization to improve performance. In addition, since inner and outer-loop parallelism exploit concurrency in different components of the program, they can be used in concert to further improve application performance.

The experiments in this section demonstrate that there is considerable fine–grain thread parallelism in typical applications and that register-based communication and synchronization provides sufficiently low overhead to exploit this parallelism efficiently. The MAP's fast interaction mechanisms (10 cycle thread invocation, 1 cycle communication and synchronization) enable application speedups of up to 2.4 on three processors, using only inner–loop parallelism. The granularity of this fine-thread parallelism is typically between 80 and 200 instructions and is largely independent of problem size. Conventional multiprocessor mechanisms with long interaction latencies are unable to exploit fine threads at all. The coarse–thread parallelism that can be exploited in multiprocessors has a granularity of 10^3 to 10^5 instructions and is strongly dependent on problem size. Based on examination of the code, we expect that fine–thread parallelism will continue to scale with more processors and that more aggressive parallelization can yield smaller grain sizes, greater concurrency, and better performance.

Chapter 7

M-Machine Project Retrospective

At the end of a large design and implementation project, it is valuable to look back and try to identify the parts of the project that went well and to evaluate those parts that could have been more successful. This chapter examines the evolution of the MAP chip architecture and takes a retrospective look at the M-Machine project. The architecture of the M-Machine and the MAP chip originated in the studies of Processor Coupling [KD92]. Section 7.1 describes Processor Coupling in some detail and describes three areas in which the MAP chip departs from the early architectural designs. Section 7.2 details the decisions that led to a small number of registers and some of the complications of using a limited register set in the MAP chip. Section 7.3 describes the simulation and software environment for the MAP chip and how it met many of our needs, but lacked some features that would have allowed better evaluation of the system. Finally, Section 7.4 discusses some of the issues associated with undertaking a complex hardware project in an academic setting.

7.1 Processor Coupling

Processor Coupling used compile-time and runtime scheduling to exploit instruction-level parallelism while maintaining high function unit utilization. A compiler scheduled each thread across multiple ALUs to exploit instruction level parallelism. The schedules of several active threads were interleaved at runtime by coupling threads to ALUs on a cycle-by-cycle basis. This interleaving made use of resources that would otherwise have been left idle due to holes in a single thread's schedule and stalls caused by synchronization and statically undetermined latencies. This combination of instruction-level and thread-level parallelism enabled fast execution of single threaded code as well as high instruction throughput and arithmetic unit utilization during periods of low ILP.

The study of Processor Coupling in [KD92] suggested a machine configuration with four clusters, each with an integer and a floating-point unit. The cluster communication experiments indicated that an interconnection network consisting of two global busses per cluster was a reasonable balance between wire costs and execution performance. The MAP chip was built on the results of this study, but some of the arithmetic units were eliminated in order to make the design fit into the available chip area. The cluster interconnection network also was reduced due to physical implementation constraints, resulting in only one global bus per cluster. The other major modifications to the Processor Coupling architecture are in the pipeline design, in the intercluster control, and in the mechanisms for data synchronization.

7.1.1 SZ Stage Placement

The Processor Coupling pipeline was very similar to the MAP's pipeline described in Chapter 3, except that the synchronization (SZ) stage came before the register read (RR) stage. An instruction would wait in the SZ stage's reservation station until its operands were marked valid. When the instruction was selected by the SZ stage issue logic, it read the register file before proceeding to the execution units. In order to reduce pipeline stalls, operands were validated two cycles before the data was produced. An instruction that consumed a previous instruction's result could read the data from the register file, or have it bypassed at the head of the execution stages.

In the MAP chip the SZ stage occurs after the RR stage, reducing the latency between validation and production of data to a single cycle. Data can be transferred more easily between back to back instructions, as the second instruction is enabled to issue when the first instruction enters the execute (EX) stage. By placing it next to the (EX) stage, the MAP's SZ stage clearly defines the instruction issue point. Any instruction that leaves the SZ stage has all of its operands and can begin execution immediately. When an exception occurs, the SZ stage immediately stops issuing instructions. The entire state of each of the user threads is captured in the instruction fetch, register read, and synchronization pipeline stages. Rolling back the pipeline is not necessary because the instruction after the one that caused the exception is still in the SZ stage. After an exception is handled, the user threads are enabled to issue and can pick up immediately where they left off prior to the interruption.

7.1.2 Cluster Synchronization

Like a VLIW, Processor Coupling used a single program counter to control all of the clusters and to exploit instruction-level parallelism. Each cluster had its own instruction sequencer and instruction cache, but branch and jump destinations would be broadcast to all of the clusters. The clusters executed in a loose lock-step discipline, similar to *tightly-coupled* mode described in Chapter 5, in which the all of the operations for a given instruction must issue before any of the operations from the next instruction. This allowed some slip across the clusters and a limited overlapping of long latency operations. Multiple threads were interleaved over all of the clusters, and if a thread stalled on one cluster, another thread could use that issue slot instead.

By using a single program counter and centralizing the control of all of the clusters, the implementation of Processor Coupling became quite complicated. Branch targets were broadcast to all of the execution units, and a global `NOT_DONE` line was required to control the slip across clusters. Since a cluster could be one instruction ahead or behind the others, swapping threads required complex bookkeeping. When a thread was swapped out, the software had to detect whether each cluster had issued the instruction corresponding to the program counter. Swapping a thread back in required setting the pipeline back to the original state which could have some clusters out of synchronization with others. The MAP chip solved all of these complexities by decoupling the clusters and giving each one its own program counter. All branches are detected and taken locally, and thread swapping only requires capturing the program counters on each of the clusters. While tightly-coupled mode can simulate Processor Coupling, we realized that decoupling would reduce the synchronization constraints between the clusters without affecting performance.

7.1.3 Remote Scoreboard Invalidation

Processor Coupling behaved much like a multithreaded VLIW machine except that a scoreboard was added to tolerate both unpredictable latencies from memory operations and transactions for intercluster communication. For transfers between execution units, the remote scoreboard was marked empty by the source unit when the instruction issued. When the data arrived, the register was marked full in the scoreboard so that a consuming operation could use the data. The primary disadvantage to this is that two remote transfers are required, one for the invalidation and one for the validation and data delivery. If two clusters are writing to a third cluster, both sources must invalidate their destination registers, which may require enough scoreboard ports for all possible writers. An alternative is to arbitrate between the writers to determine which is allowed to invalidate a remote register. However, unlike the arbitration for data delivery, the decision of which cluster to grant access to the invalidation port must be made prior to instruction issue. This greatly complicates the issue logic of the synchronization stage and results in pipeline stalls that can be easily avoided by having the destination cluster, instead of the source cluster, empty the target register.

In the MAP chip, automatic invalidation is even less useful, as the clusters are sequenced independently. Automatically invalidating a destination register is meaningless since the source cluster has no control over which instruction the destination cluster is executing. Synchronization is still necessary in order to guarantee that the instruction that consumes the data from a remote write will in fact wait until the data arrives. The `empty` instruction moves the invalidation from the source to the destination, eliminating one cross chip communication per data transfer. However, the `empty` must still occur prior to the transfer to guarantee correct data synchronization. The instruction ordering across clusters can be enforced by inserting a cluster barrier instruction or by using already existing producer-consumer relationships elsewhere in the program. The `empty` instructions have little overhead as they can often be placed in empty issue slots.

7.2 Register Limitations

With multithreading, the register file's effect on chip area can be severe since each register name must have a physical register for each thread on every cluster. In addition to reducing the silicon area required for the register files, the MAP chip limits each thread to 16 integer and 16 floating-point registers to simplify the instruction fetch logic. Operations are encoded in 32 bits so that all instructions, regardless of whether they include one, two, or three operations, lie on 32-bit boundaries. Two bits in each 32-bit operation are reserved to implement the dense instruction encoding to eliminate NOPs. Six more bits are required to implement predicated execution with two bits to determine the predicate and four bits to specify the condition register. With the

MAP's large instruction set, as many as 9 bits are needed to encode each instruction in such a way that makes the hardware decoder simple and fast. Since each instruction can specify a remote destination register, three more bits are required to encode the destination cluster identifier and the destination register file (integer or floating-point). This leaves a total of 12 bits to encode two source registers and one destination register (4 bits each), which limits architecture to 16 named registers.

At the same time, the MAP chip places additional demands on register file capacity. Registers that implement synchronization and communication between clusters cannot be used for other purposes during a program's communication phases. In addition, when a cluster sends a message to a remote MAP chip, it must compose the message in the general register file. With a limited number of registers, this can cause many values to be spilled to memory. Experience with the MAP compiler and assembly programs shows that more registers would greatly improve the code quality. In retrospect, the register files could have been larger, as they currently occupy a relatively small fraction of the MAP chip's area. Changing the instruction encoding to use 42-bit operations would have allowed us to easily encode as many as 64 registers per thread. While this would have increased the complexity of the instruction decode logic and sacrificed code density (only 3 operations per 128 bit packet instead of 4), it would have likely improved overall performance.

7.3 Simulation Environment

The M-Machine simulator (MSIM) was designed for both architectural evaluation and logic validation. As it evolved from a relatively simple architecture simulator to a cycle accurate model of the chip, it became slower and less configurable. MSIM originally used a configuration file to specify the number of registers, register files, threads, execution units, and clusters. A simple prototype compiler used the same configuration information, and the early architectural studies were performed using this parameterizable simulation environment. After the instruction encoding was selected, we began to port an industrial strength compiler and assembler, and for reasons of expedience, chose to eliminate the configurability of the system. In hindsight, we should have maintained a flexible interface between the assembler and simulator, instead of fixing the binary encodings for all of the instructions. Had we integrated the configuration interface into our port of the Multiflow Compiler as well, we would have been able to run more experiments to examine the utility of additional

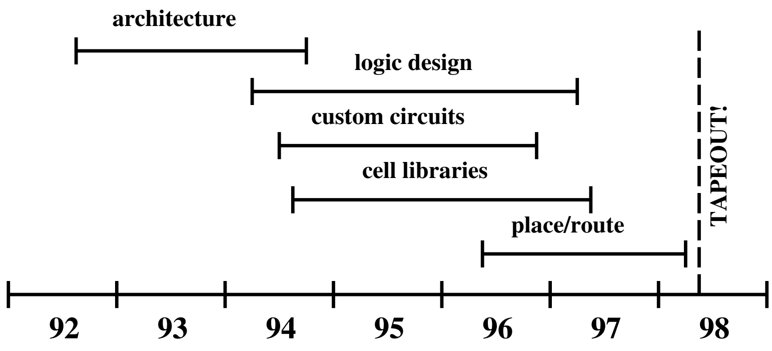


Figure 7.1: Chronology of the MAP chip design.

hardware resources.

MSIM was not only a tool for examining the performance of the architecture. It was also useful for evaluating the complexity of different hardware design decisions. There were many cases in which a particular feature, such as remote register invalidation, was deemed unreasonable for the MAP chip because the implementation in MSIM was too complex. As MSIM became a more accurate representation of the MAP chip, its execution time slowed to approximately 800 simulated cycles per second. Two different approaches could have greatly increased simulation speed. With a substantial amount of effort, we could have optimized MSIM's execution loop and boosted simulation speed to perhaps 2500 cycles per second. In the second approach, we could have built a fast instruction simulator or interpreter in addition to the slower cycle accurate simulator. While not reflecting all of the characteristics of the MAP chip, such a fast simulator would have enabled earlier and easier development of our software infrastructure, including the compiler, runtime system, and applications.

7.4 Project Complexity

The final lesson from the M-Machine project is that a large system building endeavor can be quite time consuming, especially when the design team is small. Figure 7.1 shows a rough timeline of the MAP project, from 1992–1998. Although much of the logical and physical design was performed concurrently, the final placement and routing was contingent on completion and validation of the logic design, which took longer than anticipated. In addition, we did not forsee the amount of time that would be required to iterate between the chip assembly and logic design in order to fit everything onto the chip. With our combination of full-custom and semi-custom cells, we had to choose our process technology relatively early in the project, which locked us into $0.5\mu m$ design rules. As an alternative, we might have selected a purely standard cell plus memory array methodology, which would have reduced our design time and enabled us to have a more advanced process technology. It is possible that the smaller feature size would have offset the density advantage of custom datapaths and would have enabled us to implement the same functionality, with less effort.

Some researchers believe that designing and building hardware like this is not worthwhile in an academic environment. I strongly disagree, as the alternative of performing only architectural simulations can be grossly misleading. Since silicon technology provides a set of fundamental constraints on computer systems, proposed architectures that ignore it produce results that are academically interesting but potentially irrelevant. With the MAP chip implementation, the M-Machine project was firmly grounded in the realities of VLSI. Building the prototype of the chip has been extremely valuable because it has allowed us to validate our assumptions in a way that would not be possible if we had stopped at simulation. In retrospect though, perhaps we were too aggressive in our goals to investigate processor, memory system, and network interface technologies all in the same project. Focusing on one of these components would have drastically reduced the complexity and shortened the design time of the chip. We did find, however, that combining all three subsystems necessitated some novel mechanisms that we might not otherwise have needed. In order to have a successful hardware research project, the right balance must be struck between the research goals and the functionality and design time of the prototype.

Chapter 8

Conclusion

The most significant constraint facing high speed integrated circuit designers is the increase in on-chip wire delay [Sem97]. By 2007, nearly thirty 500ps clock cycles are expected to be needed to send a signal across the diagonal of a single chip. Today's microprocessor designs which require long wires between centralized controllers and distributed execution units will be impractical in future silicon technologies. Emerging microprocessor architectures must minimize global communication and the large latencies they imply. Since traditional multiprocessor architectures face similar constraints in the tradeoffs between local and global communication, incorporating multiple independent processors on a single chip is certainly a promising use of the silicon area. Moreover, adding integrated interprocessor communication and synchronization mechanisms will increase the utility of the execution units. More parallelism can be extracted at finer grain sizes if the latency and overhead for communicating between tasks is greatly reduced.

Instruction-level parallelism (1 cycle tasks) and coarser grained multiprocessor concurrency (10,000 cycle tasks) dominate the parallelism landscape. However, today's 4–8 issue superscalar processors are nearing the limits of ILP, and most applications have limited coarse-grain parallelism, particularly at smaller problem sizes. To continue to achieve higher performance with every generation of microprocessor, new forms of parallelism must be exploited on a single chip. The four orders of magnitude between the task granularities of ILP and coarse grain threads expose a tremendous gap in which parallelism exists but cannot be extracted with today's computer systems. Fine-grain thread-level parallelism, with tasks lengths less than 100 cycles, are made possible by low cost communication and synchronization mechanisms and are well suited to fill this performance gap.

Fine-grain threads are also well matched to the cluster organizations of future microprocessors. Most applications, even those with small problem sizes, have considerable fine-thread parallelism. This parallelism, because of its limited extent, has a smaller cache footprint than coarse-thread alternatives [FD95]. However, the development of fine–grain programs has been a chicken–and–egg proposition. Fine–grain applications are not prevalent because there are no machines with fine–grain mechanisms, and vice versa.

The contribution of this thesis is the design, implementation, and evaluation of a single chip parallel computer that meets the wire latency constraints of silicon process technology and provides opportunities for new forms of parallelism. The prototype is the MIT Multi-ALU Processor (MAP) chip which includes three independent on-chip processors in a 5 million transistor custom VLSI implementation. The key features include fast register-register communication between processors, a global processor barrier instruction, and zero-cycle multithreading. The novel mechanisms used to implement these features include a register scoreboard that can be manipulated by a user level `empty` instruction, and a synchronization pipeline stage to determine when an instruction's operands are present. This thesis has discussed the design issues associated with each of these hardware mechanisms to support on-chip concurrency, and described how they can be incorporated into a real processor pipeline. The interprocessor communication and synchronization mechanisms are used to extend the use of instruction-level parallelism to multiple independent processors and to exploit fine-grain thread level parallelism. In the evaluation of the integrated interaction mechanisms, this thesis has demonstrated how to use fine-grain threads to achieve speedups of up to 2.4 times on three processors when only the code within an application's inner loop are parallelized.

8.1 MAP Chip Summary

The MAP chip, which forms the foundation of the M-Machine, is intended to exploit parallelism at all levels and to extract more parallelism from problems of fixed size, rather than requiring enormous problems to achieve peak performance. On-chip execution units are organized into independent processor clusters which are connected to a shared cache memory system. Each of the three clusters is able to extract instruction-level parallelism using its three execution units. Both instruction and thread level parallelism can be executed across all three clusters using the fast intercluster communication and synchronization mechanisms. Threads on separate clusters communicate by writing into each other's register files via the Cluster Switch, and synchronize using a global barrier instruction. Two memory operations can access the on-chip cache simultaneously and the paths to and from the memory system are deeply pipelined. Multithreading allows the execution resources of a cluster to be used when one thread stalls for a short or long period of time. Coarse-grain thread-level parallelism can be exploited across multiple MAP chips by connecting them together through the integrated on-chip network interface and router.

The MAP chip pipeline employs novel features to implement zero-cycle multithreading and fast intercluster synchronization. A synchronization (SZ) pipeline stage orchestrates instruction execution across all three arithmetic units within each cluster. Instructions from each thread wait in the reservation stations of the SZ stage until all operands are present. The SZ stage examines the instructions from each thread and selects a thread to issue based on data availability, thread priority, and arbitration. The MAP also uses valid bits in a register scoreboard and in pipeline registers to unify nearly all instruction synchronization through data dependence, eliminating pipeline interlocks. The scoreboard tracks the data from local arithmetic operations, memory operations, and remote register writes. Intercluster synchronization takes place through data transfer or by using a cluster barrier instruction which is implemented in the SZ pipeline stage. Because the thread communication and synchronization mechanisms are implemented primarily by augmenting the existing cluster to memory communication paths, their cost is small.

The MAP's highly-integrated processor interaction mechanisms are substantially faster than the alternative of using the shared memory system. Communication between threads on different clusters requires only one cycle of latency when using the Cluster Switch to transfer a word to a remote register file, while communicating though memory takes at least 10 cycles. Threads can synchronize in a single cycle using the cluster barrier instruction, but need 60 cycles to execute a barrier through the on-chip cache. The MAP chip also implements a user level thread invocation instruction (`hfork`), which initiates a thread on a remote cluster. This instruction enables thread invocations at one-third the latency of using load and store instructions to access the MAP chip's thread control registers. With communication and synchronization latencies of a single cycle, threads need only to execute 10 cycles between interactions to keep the overhead below 10%. Thus with such low overhead operations, fine-grain threads that execute for less than 100 cycles are feasible.

With the fast register communication and synchronization mechanisms, the MAP chip can execute instruction-level parallel code on independent processors. Instead of lock-step VLIW-style synchronization, the MAP chip allows these ILP programs to synchronize only when necessary, which enables streams on different clusters to slip relative to one another and overlap their long latency operations. Explicitly synchronizing the streams is simpler to implement in hardware and is up to 5% faster on the application cores than implicitly synchronized streams, even when counting the overhead for executing the synchronization instructions.

In this study, the MAP's fast communication mechanisms are also used to implement fine-grain thread-level parallelism via a parallel procedure call (PPC), in which a master thread dynamically assigns work to the slave threads on the other execution units. Parallelizing the inner loops of several applications using PPC yields performance improvements of 1.2–2.4 times even on small problem sizes. The register communication mechanisms result in a 15-20% improvement over communication via the on-chip cache. To put this performance improvement into perspective, several studies have shown that increasing the number of execution units in a dynamic superscalar processor from 2 to 4 also results in a 15-20% speedup on integer applications [SLH90, TW92]. The measured speedup is limited by both the overhead of thread control, and by the sequential components of the program which are not accelerated. When outer-loop parallelism is available, it generally yields faster execution times than inner-loop parallelism, as less communication and synchronization are required. However, some applications have no outer-loop parallelism and require fine-grain parallelism and additional hardware support for communication and synchronization to improve performance. In addition, since inner and outer-loop parallelism exploit concurrency in different components of the program, they can be used in concert to further improve application performance.

While the performance improvement using direct communication instead of the cache is only 15-20% in today's technologies, it will be much more significant in the future. As shown in Chapter 4, over the next several process generations the latency for global communication using a cache will increase from 10 to 56 cycles, while register communication latency will increase from 1 to 26 cycles. In fact, the actual remote cache communication latency is likely to be more than 56 cycles due to intermediate cache access delays and software overhead for spinning or polling to synchronize on data arrival. The divergence in global synchronization latency between hardware methods and using a cache hierarchy is even more significant. The synchronization latency using caches increases from 60 to 860 cycles, while the latency for a mechanism such as the cluster barrier instruction (`cbar`) only increases from 1 to 26 cycles. As a result of this divergence in costs to communicate and synchronize, future chips that provide hardware support for fast interprocessor interactions are likely to yield significant speedups over chips that only allow interactions through memory.

Since silicon technology provides a set of fundamental constraints on computer systems, architecture research that ignores it produces results that can be grossly misleading. With the implementation of a complex custom VLSI microprocessor, the M-Machine project was firmly grounded in the realities of VLSI. The success of the project has been due to a tremendous amount of hard work by the design team at MIT, as well as a successful collaboration effort with an industrial partner. By combining investigations into processor, memory system, and network interface technologies, we discovered novel mechanisms that enabled better interaction between these three subsystems that we might not otherwise have needed. While pushing the technology aggressively for multiple subsystems required quite a bit of effort, the end result was much more valuable than the alternative of focusing only in one area. In order to have a successful hardware research project, the right balance must be struck between the research goals and the functionality and design time of the prototype.

8.2 Architectures for Future Chips

The driving force behind faster microprocessors has been technology advances that have reduced transistor and wire dimensions. Moore's law has accurately predicted a doubling of the number of MOS transistors on a single chip every 1-2 years [Moo95]. Since the on-chip clock rate has been historically dominated by transistor delay, smaller transistors have led directly to faster cycle times. However, the dramatic change in the balance between transistor and wire delays is placing new constraints on existing microarchitectures. Based on projections for a 2GHz 0.1μm CMOS chip, all wires must be less than 2.5mm long, even if using copper interconnect, in order to have less than one clock cycle transmission latency. This means that large monolithic structures such as high capacity caches and memory arrays will not be feasible as the word and bit lines will be too long and too slow. Furthermore, modules that need to communicate with one another at 2GHz must be located in close proximity. Scaling existing superscalar and VLIW microarchitectures or increasing monolithic on-chip cache or memory arrays in future technologies is not feasible. Even extending a traditional shared multi-level cache memory hierarchy for multiple on-chip processors will be inappropriate as the latencies between processors and remote reactive caches will be too great.

As the number of transistors that can be fabricated on a single chip increases, so will the number of arithmetic execution units. However, the changing balance between gate delay and wire delay will require decentralization of control so that the execution units will be partitioned into independent processors. The processors must be small enough so that all local wires are short, to allow a processor to run at the full clock rate. Likewise, each primary memory array must be small and located very close to a processor to enable high bandwidth and low latency access. The abundance of processors will create a large demand for fine-grain parallelism that cannot be met using existing architectures.

Efficient mechanisms are required to allow processors to interact, to access remote memory, and to overlap local operations with remote accesses. Processors will communicate with one another and with remote memory modules via an on-chip network. The interface to this network will be integrated into the instruction set. Processors will communicate by writing into remote registers or FIFOs that can be read locally by a destination processor. With low overhead interfaces to this on-chip network, communication latency will be dominated by wire delay. Nearby processors will be able to communicate with one another in less than five 2GHz cycles, while remote processor communication may take thirty cycles. A promising approach to further reduce remote memory access time is for a processor to communicate directly and proactively with a remote memory module. Thus a data producer may write directly into a consumer's memory enabling subsequent accesses by the consumer to be local. A hierarchical synchronization network will also be implemented to enable multiple processors to synchronize simultaneously. Localized groups of processors will be able to synchronize at the latency of a local communication (5 cycles), while synchronization across all of the processors will require global communication (30 cycles). Architectures for future process technologies will be motivated by the constraints of wire delay and the opportunities of high on-chip bandwidth resulting from narrow wires and high wire density. Combining proactive communication techniques with the increased on-chip bandwidth may be able to offset the effect of large on-chip latencies. Without partitioning a chip into independent processors and providing fast integrated communication and synchronization mechanisms, the large numbers of transistors in future process generations will not be used effectively to increase overall system performance.

The results of this thesis demonstrate that engineers of future computer systems must design for physical locality even within a single chip. The fundamental importance of wire delay dictates that those components of the chip that communicate frequently with one another must be placed in close proximity. This implies that memory arrays must be small and that execution units and memory modules must be close together in order to have fast memory access. Moreover, the communication latency between on-chip processors increases with the physical distance between them. The granularity of parallelism that can be exploited will depend on the distance between processors to which parallel tasks are assigned. Tasks mapped to nearby processors will be able to execute very fine-grain parallelism, while tasks mapped to distant processors will execute at a coarser grain. The effects of wire delay present both challenges and opportunities to applications since the ability to exploit locality will have a profound impact on a program's performance.

8.3 Software Support

Software support will be required to discover fine–grain parallelism in existing programs. Aside from hand parallelization, compilers may be able to analyze and partition inner loop iterations, procedure calls, and expressions. New research in compiler algorithms and analysis will be required, as current parallelizing compilers assume large communication and synchronization latencies. Other avenues, such as pipelining dependent do-across loop iterations across the on-chip processors, or speculatively executing components of the program in parallel are possible as well. Regardless of the technique, fine–grain threads enable a different and orthogonal type of parallelism than that found in outer loops. Reducing the synchronization and communication costs between parallel tasks will enable fine–grain parallelization of programs, and allow existing problems, such as personal or business applications, to be solved faster without scaling their size.

At a more fundamental level, automatic parallelization of inherently sequential programs has limited benefit. Many of today's applications are sequential due to the selected algorithm and the programming language, rather than because the problem to be computed lacks concurrency. For example, many applications that use linear compression could alternatively employ an algorithm that breaks a data stream into multiple streams and compresses them simultaneously. To effectively exploit the concurrency available on a future chip will require parallelism to be explicit at all levels, ranging from the hardware/software interface to the algorithms developed for solving problems. Explicit parallelism will certainly require innovations in programming languages as well as compile-time and runtime resource management. Dynamic compilation is a promising technique for enabling compatibility across different chip configurations by separating hardware resource management from the programming language target. While retrofitting existing applications to execute well in concurrent environments will be difficult, emerging applications such as media and speech processing will be more amenable for execution in future chips. As they interact in the changing environment of the physical world, these applications have very dynamic behaviors with concurrency at many granularities. Efficient communication and synchronization mechanisms will be critical for exploiting parallelism and enabling high performance and efficient execution of future applications.

Appendix A

MAP Instruction Set Architecture

The MAP uses a custom instruction set architecture that is similar to many of the commercial RISC load/store microprocessor architectures. In addition to the usual memory and arithmetic operations, the MAP provides special instructions for protection, address space management, thread invocation, and off-chip communication. Each MAP instruction consists of three operations, one each for the integer, memory, and floating-point units in a cluster. Each operation is encoded in 32 bits and NOP operations in an instruction are stored in a compressed format in memory. The NOPs are expanded on the fly for instruction execution. This appendix provides a brief description of the MAP instruction set architecture, including a listing of all of the instructions. The entire instruction set architecture description, including all bit encodings, is detailed in [DKC+94].

A.1 Operation Fields

The following instruction sequence shows two sequential MAP instructions. The first contains an integer, a memory, and a floating point operation, while the second contains only an integer operation.

```
instr  ialu add   i5, i6, i7
       memu ld    i11, i12
       falu fadd  f3, f4, h1.f2;
instr  ialu sub   i7, i8, f13;
```

The memory and floating-point NOPs in the second instruction will be inserted on the fly by

	Pack Bits	Meaning
First operation	00	IU operation only
	01	MU operation only
	10	FPU operation only
	11	Multi-operation instruction
Second operation	00	IU-MU instruction
	01	IU-FPU instruction
	10	IU-MU-FPU instruction
	11	MU-FPU instruction

Table A.1: Instruction pack bits to compress NOPs from instruction stream. If the instruction contains two or three operations, the pack bits from the second operation are required.

---

the MAP instruction fetch unit. Integer registers are `i2`–`i15`, and the floating-point registers are `f1`–`f15`. Registers `i0` and `f0` are mapped to the value zero, and `i1` is the value of the program counter for the executing instruction. Each instruction can target a register file in a remote cluster by prefixing the destination register with `h1` or `h2`. Destination clusters are indicated with relative names so that `h1` is the next numerically named cluster. The clusters are numbered 0–2, and the names wrap back to 0 after 2.

The figure below shows a sample operation encoding for the integer arithmetic instructions. Each operation is encoded using 32 bits, and all operations share the packing and predicated execution fields. The other fields may vary among operations, but the format of all operations is similar.

pack	cond	cr	opcode	dh	dr	imm	dest	src2	src1
2	2	4	5	2	1	1	4	7	4

The `pack` field is used to compress NOPs from the instruction stream and reduce the space required to store a program. If an instruction stream has only one operation, the pack bits from that operation determine whether it is for the integer, memory, or floating-point unit. If the instruction has more than one operation, the pack bits from the second operations must be examined. Table A.1 enumerates the possible encodings. In instruction sequence above, the first instruction has pack bits of `11` and `10` in its first and second operations. The second instruction, with only an integer operation, has pack bits of `00`.

The `cond` and `cr` fields are used to implement predicated execution. The `cond` field identifies

Condition	Encoding	Function
CF	00	conditionally execute if FALSE
CT	01	conditionally execute if TRUE
UA	10	unconditionally execute always
UN	11	unconditionally execute never

Table A.2: Predicates used to conditionally execute each instruction.

on what condition the operation will be executed or conditionally nullified, as shown in Table A.2. The `cr` field identifies which condition register to test to determine nullification. In the following example, the branch instruction is executed if the value of `cc1` is `true`. Otherwise the instruction is nullified and turned into a NOP at execution time.

```
instr  ialu ct cc1 br _loop;
```

The `opcode` field identifies the operation to execute. The `dh` field indicates the cluster to which cluster the operation's result is sent. If `dh == 0`, the operation writes its result to a local register file. The `dr` field identifies the destination register file, either integer or floating-point. The `imm` bit determines whether `src2` is an immediate or a register name. Finally, `src1` and `src2` are the operands, and `dest` is the destination register name.

A.2 Integer Operations

A.2.1 Arithmetic Operations

`add`	integer signed add
`addu`	integer unsigned add
`sub`	integer signed subtract
`subu`	integer unsigned subtract
`ash`	arithmetic shift
`lsh`	logical shift
`rot`	rotate
`and`	bitwise logical and
`or`	bitwise logical or
`xor`	bitwise logical exclusive-or

`not`	bitwise logical negation
`ccand`	condition code logical and
`ccor`	condition code logical or
`ccnand`	condition code logical nand

A.2.2 Byte Manipulation

`extb`	extract byte from 8-byte word
`exth`	extract 4-byte halfword from 8-byte word
`insb`	insert byte into 8-byte word
`insh`	insert 4-byte halfword into 8-byte word

A.2.3 Comparison Operations

`ilt`	integer less-than
`ile`	integer less-than or equal
`ult`	unsigned less-than
`ule`	unsigned less-than or equal
`ine`	integer not-equal
`ieq`	integer equal

A.2.4 Data Movement

`mov`	move immediate or register
`empty`	invalidate vector of integer registers
`ccempty`	invalidate vector of condition code registers

A.2.5 Control Flow Operations

`br`	relative branch
`jmp`	absolute jump
`ill`	user generated illegal instruction

A.2.6 Address Calculation

`lea`	load effective address
`leab`	load effective address from segment base
`setptr`	set pointer bit
`unsetptr`	unset pointer bit
`isptr`	test pointer bit
`iserr`	test for *errval*

A.2.7 Immediate Operations

`imm`	create 16-bit immediate
`shoru`	shift-then-or unsigned 16-bit immediate

A.2.8 Configuration Space Operations

`igtwr`	write to global translation lookaside buffer (GTLB)
`igtrd`	read from global translation lookaside buffer (GTLB)
`igprb`	probe global translation lookaside buffer (GTLB)

A.2.9 Communication Operations

`isnd0`	send priority 0 message (*user level*)
`isnd0o`	send priority 0 message and preserve message ordering (*user level*)
`isnd0p`	send priority 0 message using physical address
`isnd0po`	send priority 0 message using physical address and preserve message ordering
`isnd0pnt`	send priority 0 message using physical address with no message throttling
`isnd0pnto`	send priority 0 message using physical address with no message throttling and preserve message ordering
`isnd1pnt`	send priority 1 message using physical address with no message throttling
`isnd1pnto`	send priority 1 message using physical address with no message throttling and preserve message ordering

A.3 Memory Operations

A.3.1 Standard Memory Access

`ld`	load register
`st`	store integer register
`fst`	store floating-point register
`luc`	load register and if a cache miss occurs, place the incoming line into the block buffer instead of the cache

A.3.2 Synchronizing Operations

`lds`	load register and fault if memory synchronization fails

`ldscnd`	load register and return result of memory synchronization test
`ldsu`	load register and return result of memory synchronization test; ignore memory block status failure
`sts`	store integer register and fault if memory synchronization fails
`stscnd`	store integer register and return result of memory synchronization test
`stsu`	store integer register and return result of memory synchronization test; ignore memory block status failure
`fsts`	store floating-point register and fault if memory synchronization fails
`fstscnd`	store floating-point register and return result of memory synchronization test
`fstsu`	store floating-point register and return result of memory synchronization test; ignore memory block status failure

A.3.3 Address Calculation

`lea`	load effective address
`leab`	load effective address from segment base

A.3.4 Special Memory Operations

`cbar`	cluster barrier
`srs`	store integer register and overwrite memory synchronization bit
`fsrs`	store floating-point register and overwrite memory synchronization bit
`flne`	flush cache line
`getcstat`	read cache line block status
`putcstat`	write cache line block status
`mbar`	memory barrier; block until all outstanding memory references complete

A.3.5 Thread Management Operations

`hfork`	invoke thread in another cluster
`hexit`	terminate current thread

A.3.6 Arithmetic Operations

`add`	integer signed add
`sub`	integer signed subtract

`mov`	move immediate or register
`and`	bitwise logical and
`or`	bitwise logical or
`xor`	bitwise logical exclusive-or
`not`	bitwise logical negation

A.4 Floating-point Operations

A.4.1 Floating-point Arithmetic Operations

`fadd`	floating-point add
`fsub`	floating-point subtract
`fmul`	floating-point multiply
`fdiv`	floating-point divide
`fmula`	floating-point fused multiply-add
`fsqrt`	floating-point square-root

A.4.2 Integer Arithmetic Operations

`imul`	integer multiply (low 64 bits of 128 bit product)
`hmul`	integer multiply (high 64 bits of 128 bit product)
`idiv`	integer divide
`idivu`	unsigned integer divide

A.4.3 Data Movement

`mov`	move immediate or register
`fempty`	invalidate vector of floating-point registers

A.4.4 Data Conversion

`itof`	integer to floating-point conversion
`ftoi`	floating-point to integer conversion

A.4.5 Comparison Operations

`flt`	floating-point less-than
`fle`	floating-point less-than or equal
`feq`	floating-point equal
`fne`	floating-point not-equal

A.4.6 Immediate Operations

`fimm`	create 16-bit immediate
`fshoru`	shift-then-or unsigned 16-bit immediate

A.4.7 Communication Operations

`fsnd0`	send priority 0 message (*user level*)
`fsnd0o`	send priority 0 message and preserve message ordering (*user level*)
`fsnd0p`	send priority 0 message using physical address
`fsnd0po`	send priority 0 message using physical address and preserve message ordering
`fsnd0pnt`	send priority 0 message using physical address with no message throttling
`fsnd0pnto`	send priority 0 message using physical address with no message throttling and preserve message ordering
`fsnd1pnt`	send priority 1 message using physical address with no message throttling
`fsnd1pnto`	send priority 1 message using physical address with no message throttling and preserve message ordering

Appendix B

Graphs of Application Results

Figures B.1– B.5 in Section B.1 show the cycle breakdowns for each of the applications of Chapter 6, using inner-loop parallelism. The application set includes multigrid (MG), fast-Fourier transform (FFT), electromagnetic simulation (EM3D), conjugate gradient (CG), and a simulation of the human cochlea (EAR). The applications are parallelized by examining the inner loops to find subroutines and expressions that can be executed concurrently. The **Cache** bars show the components of running time when using the on-chip cache to communicate between the master and the slave threads, while **Register** uses the MAP's register communication mechanisms. Each application is run on a variety of data sets, ranging from small to medium sizes. The running time is broken down into the cycles spent executing instructions (**Execute**), waiting for the instruction fetch unit and instruction cache (**IFU**), waiting for data from the memory system (**Memory**), and communicating between the clusters (**Comm**). Figure B.6 summarizes the execution time of the applications across all of the problem sizes.

Figures B.7– B.10 in Section B.2 shows the cycle breakdowns for each of the applications using outer-loop parallelism. Outer–loop parallelism comes from the outer loops of the applications, mainly by dividing the data set across the processors and assigning independent loop iterations to them. The different phases of the computation are separated by barriers, which are implemented either in memory (**Cache**) or using the cluster barrier instruction **CBAR**. The cycle breakdown is categorized like the inner–loop graphs except that the **Barrier** bars are used to indicate the sum of the barrier overhead and the time spent waiting at a barrier. Figure B.11 summarizes the total execution time for each of the outer-loop applications.

B.1 Inner–Loop Parallelism

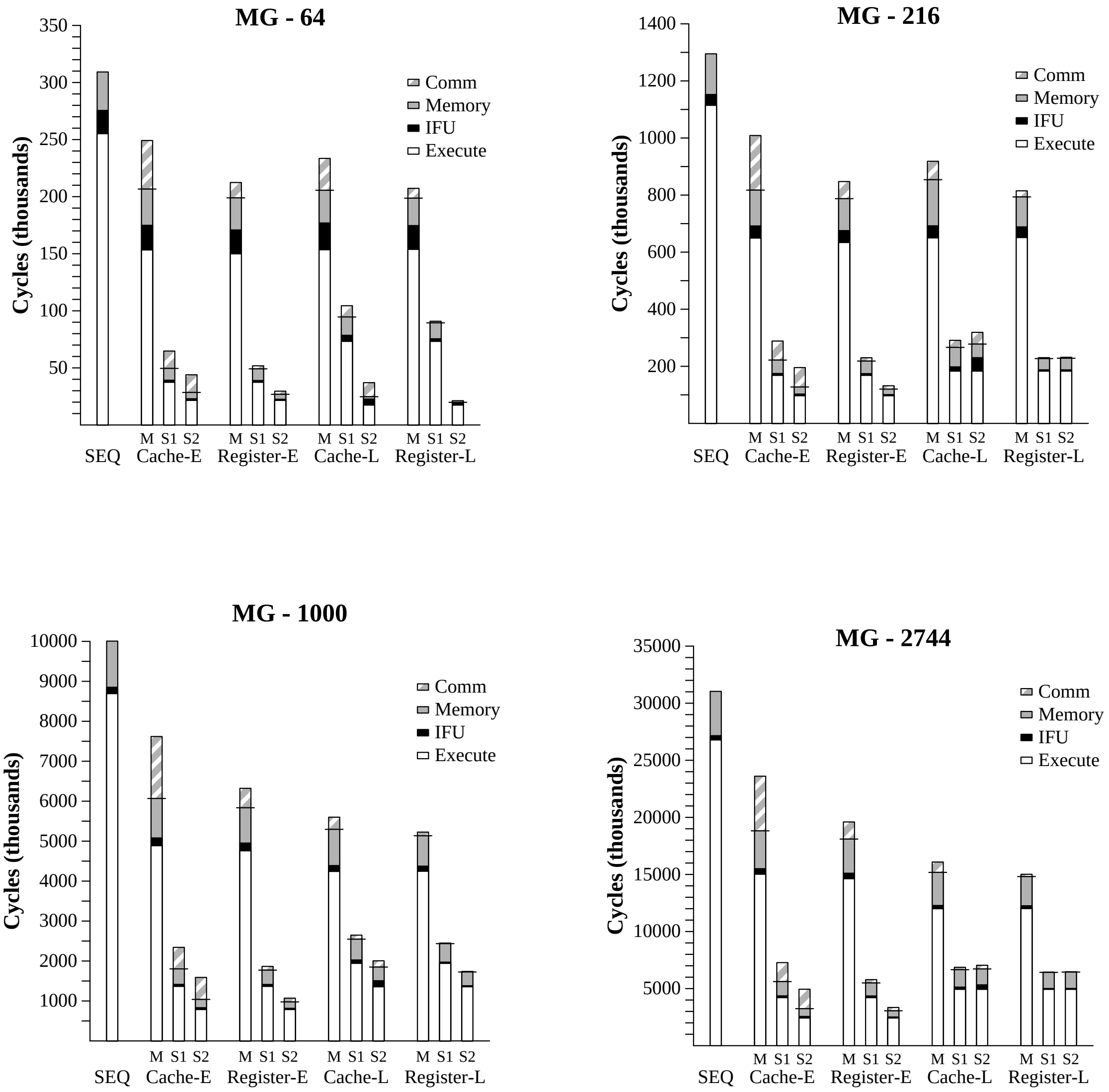


Figure B.1: MG cycle breakdown using inner-loop parallelism.

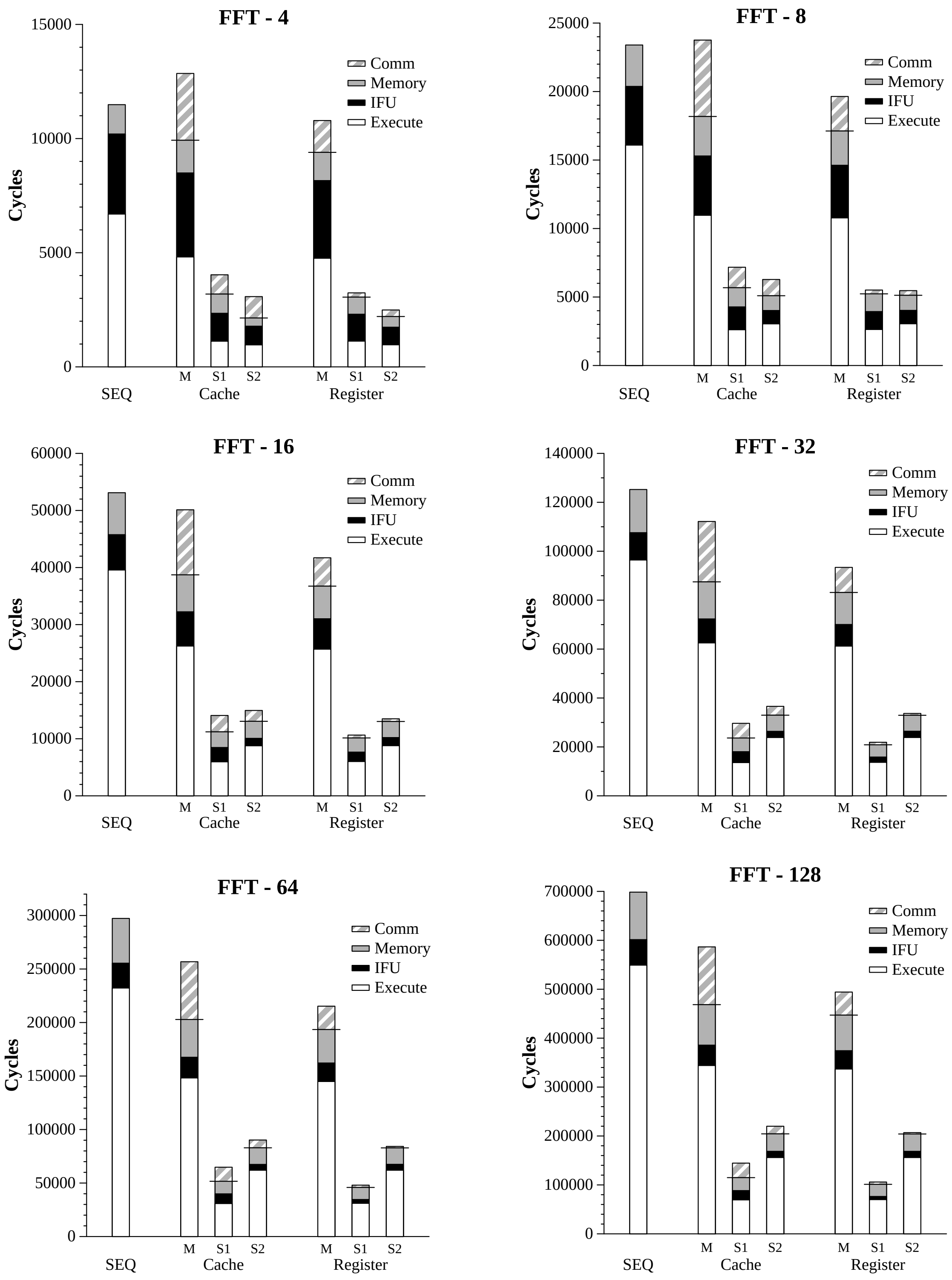


Figure B.2: FFT cycle breakdown using inner-loop parallelism.

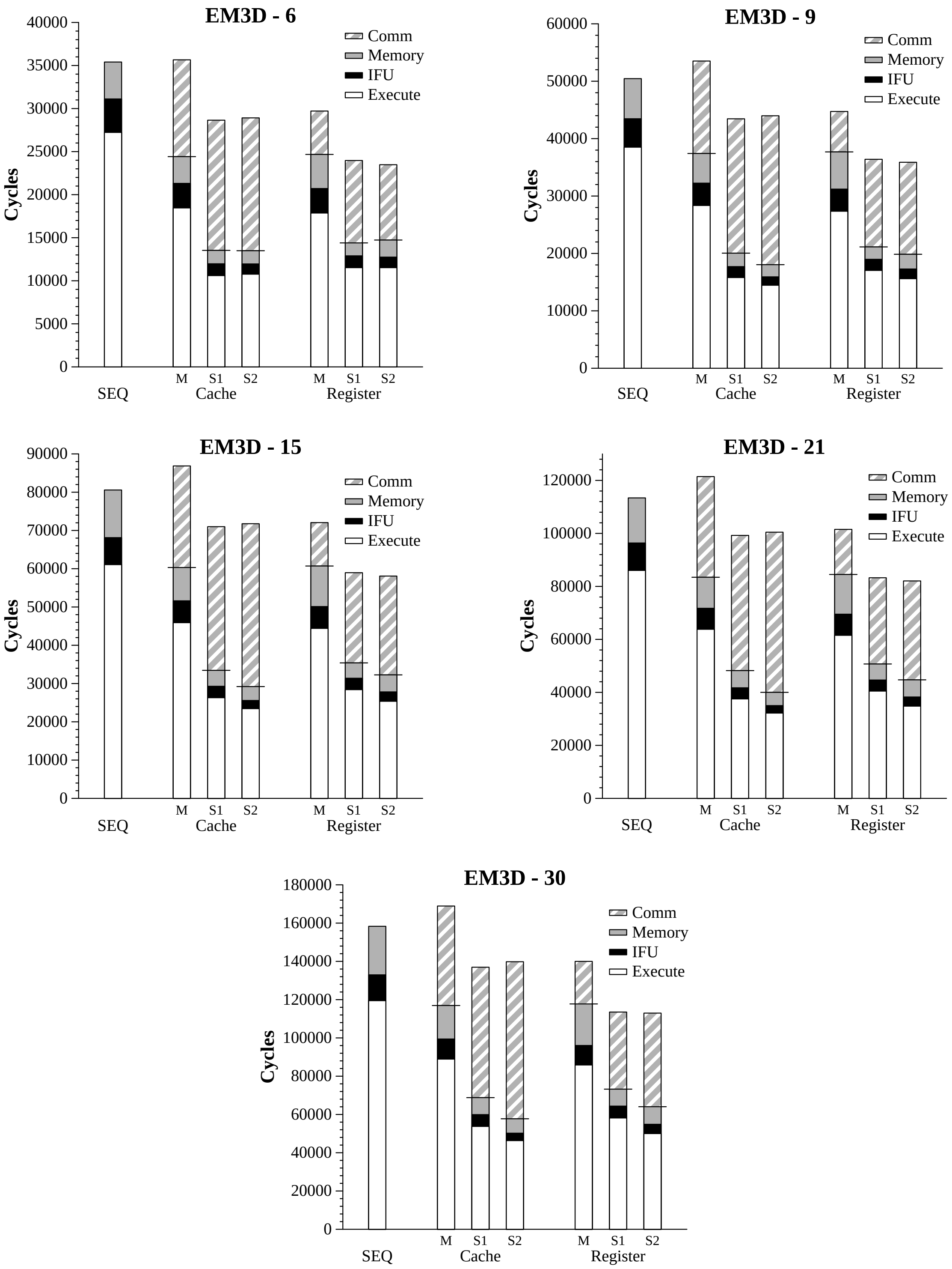


Figure B.3: EM3D cycle breakdown using inner-loop parallelism.

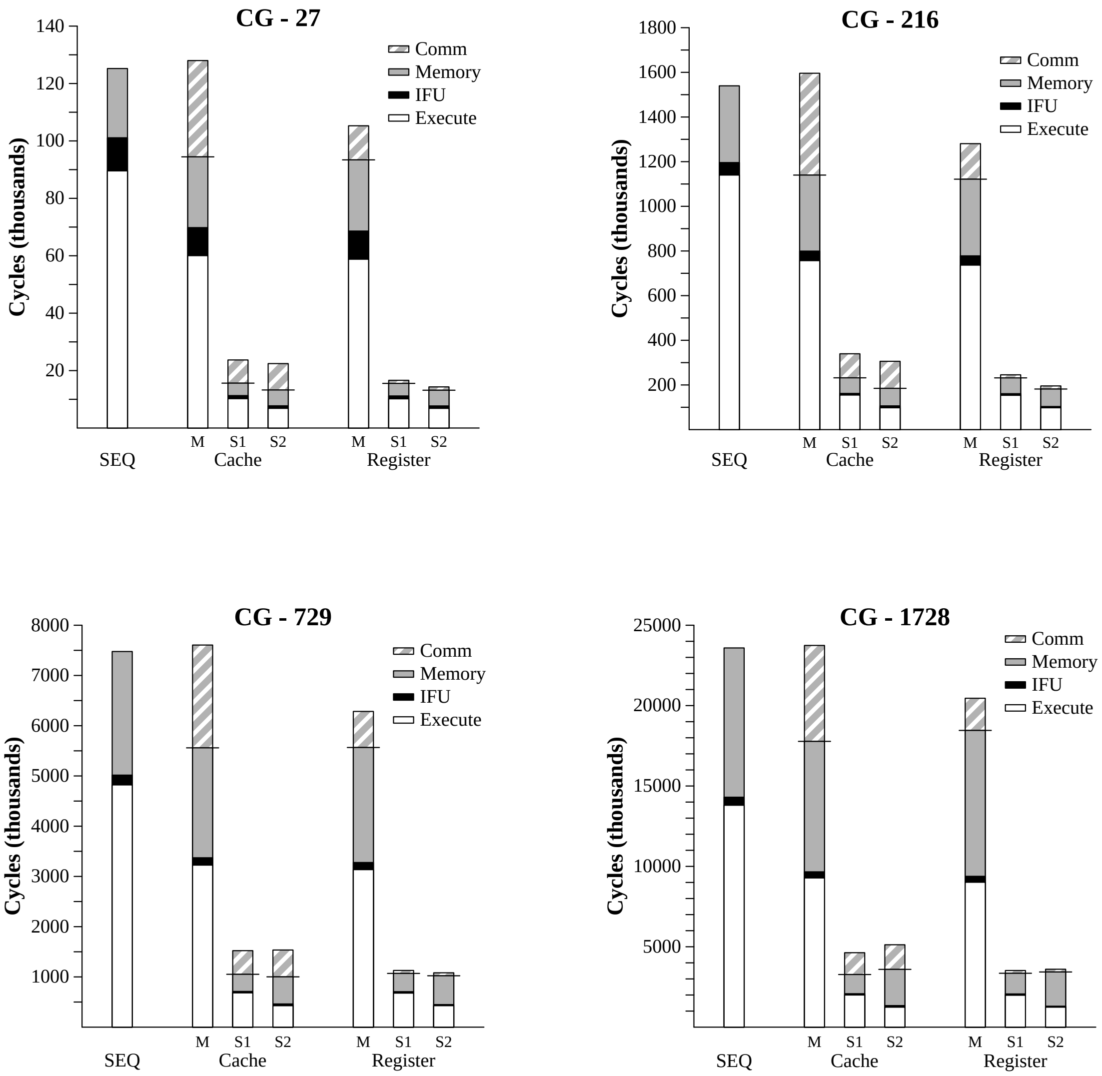


Figure B.4: CG cycle breakdown using inner-loop parallelism.

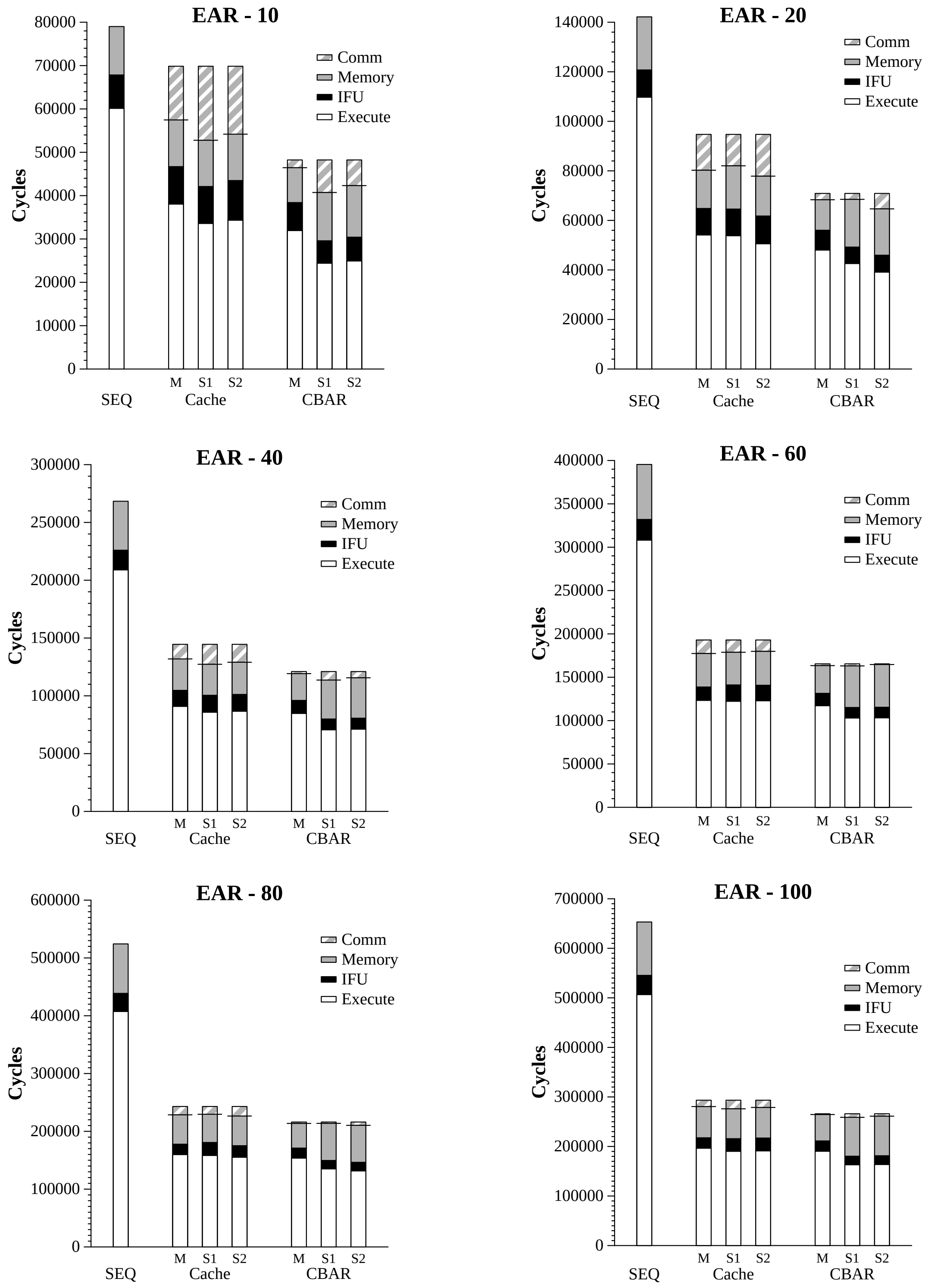


Figure B.5: EAR cycle breakdown using inner-loop parallelism.

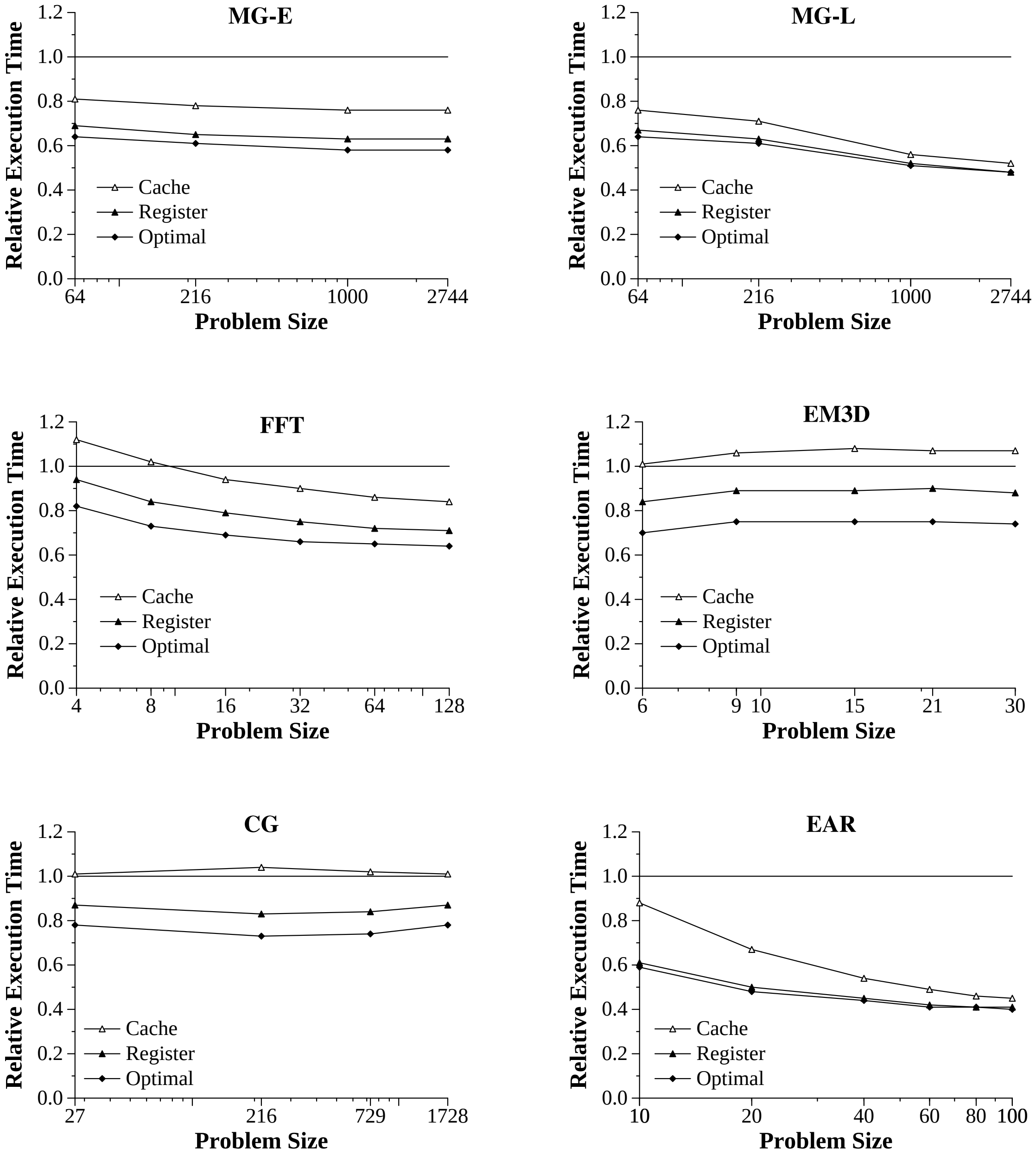


Figure B.6: Summary of inner–loop execution times.

B.2 Outer–Loop Parallelism

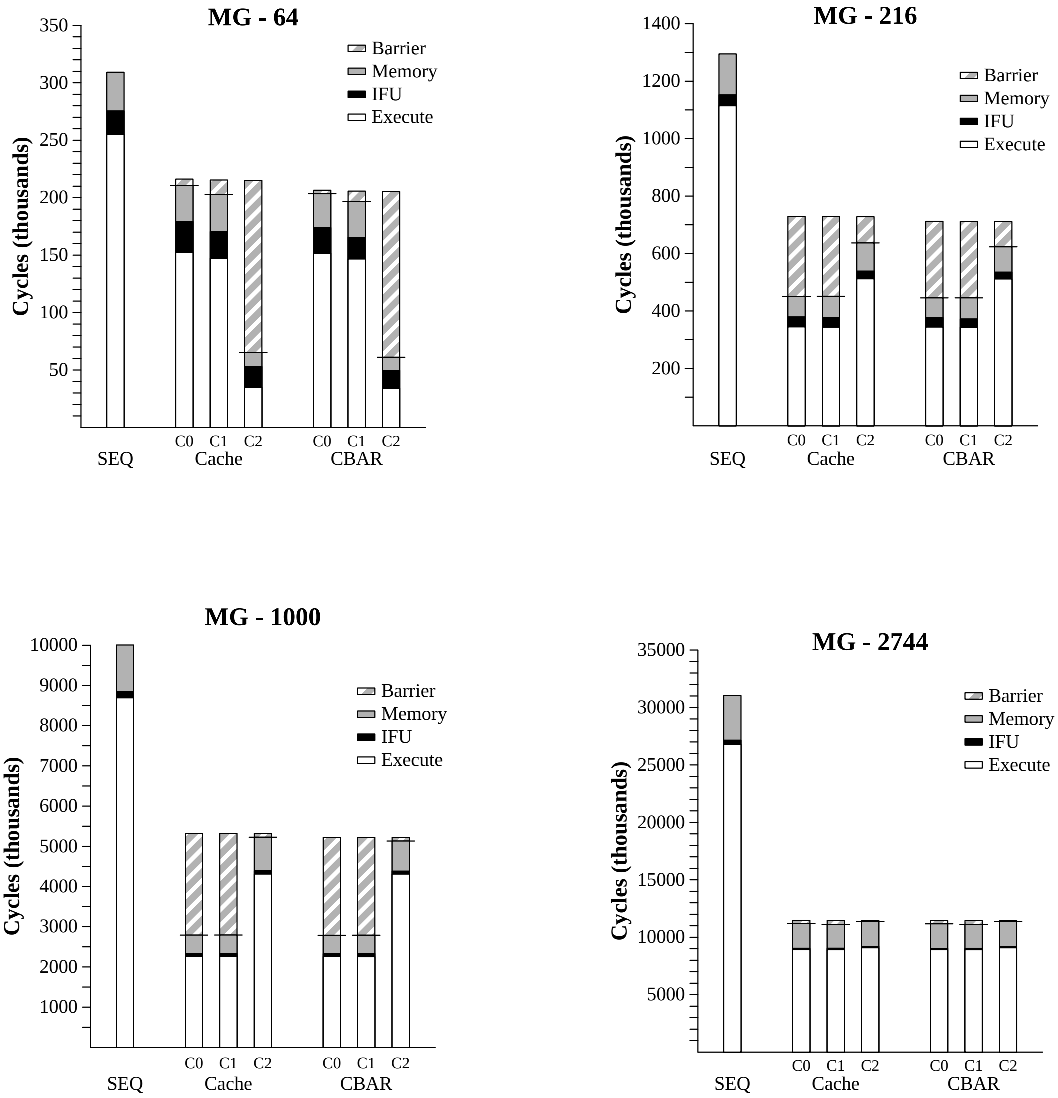


Figure B.7: MG cycle breakdown using outer-loop parallelism.

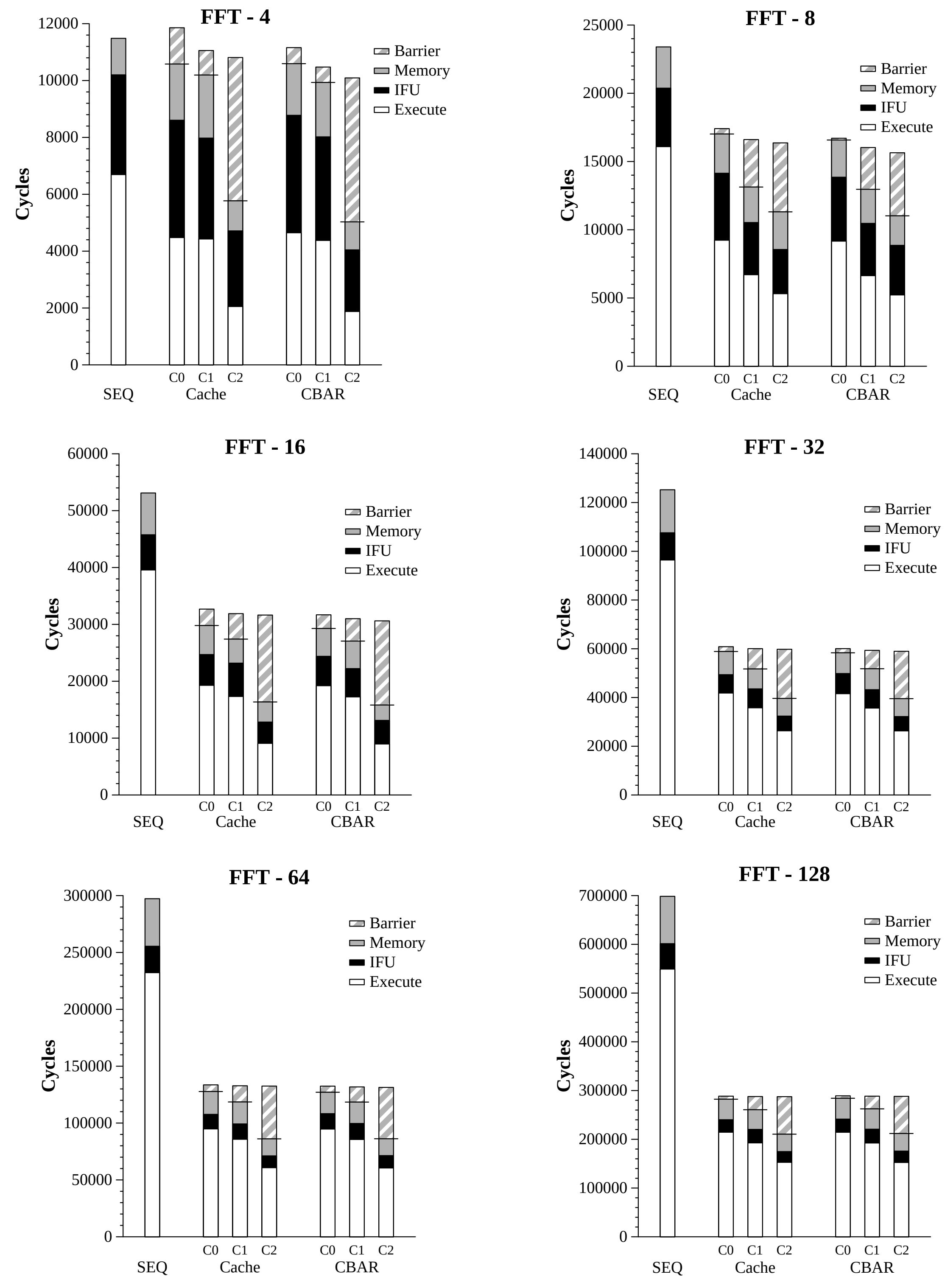


Figure B.8: FFT cycle breakdown using outer-loop parallelism.

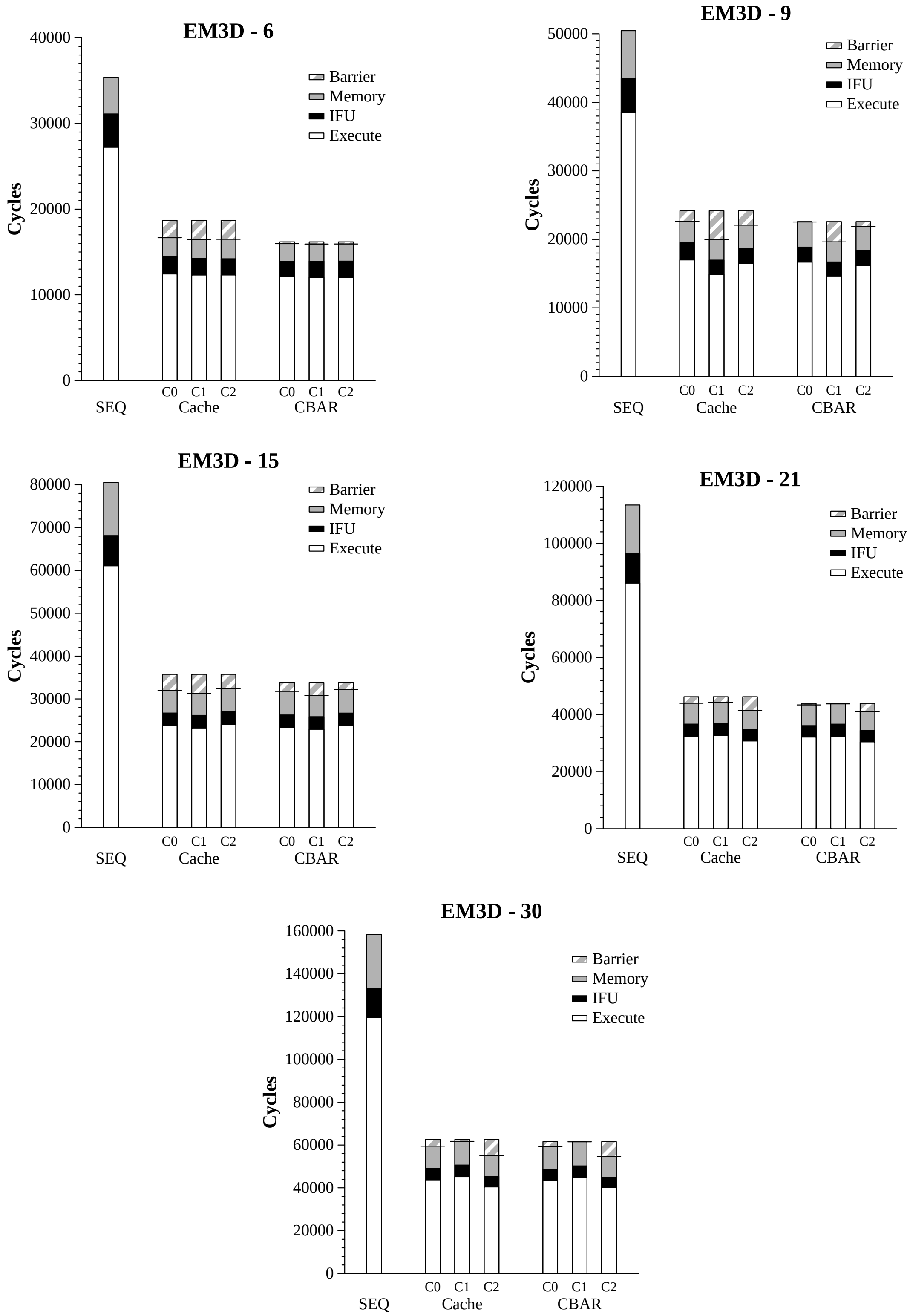


Figure B.9: EM3D cycle breakdown using outer-loop parallelism.

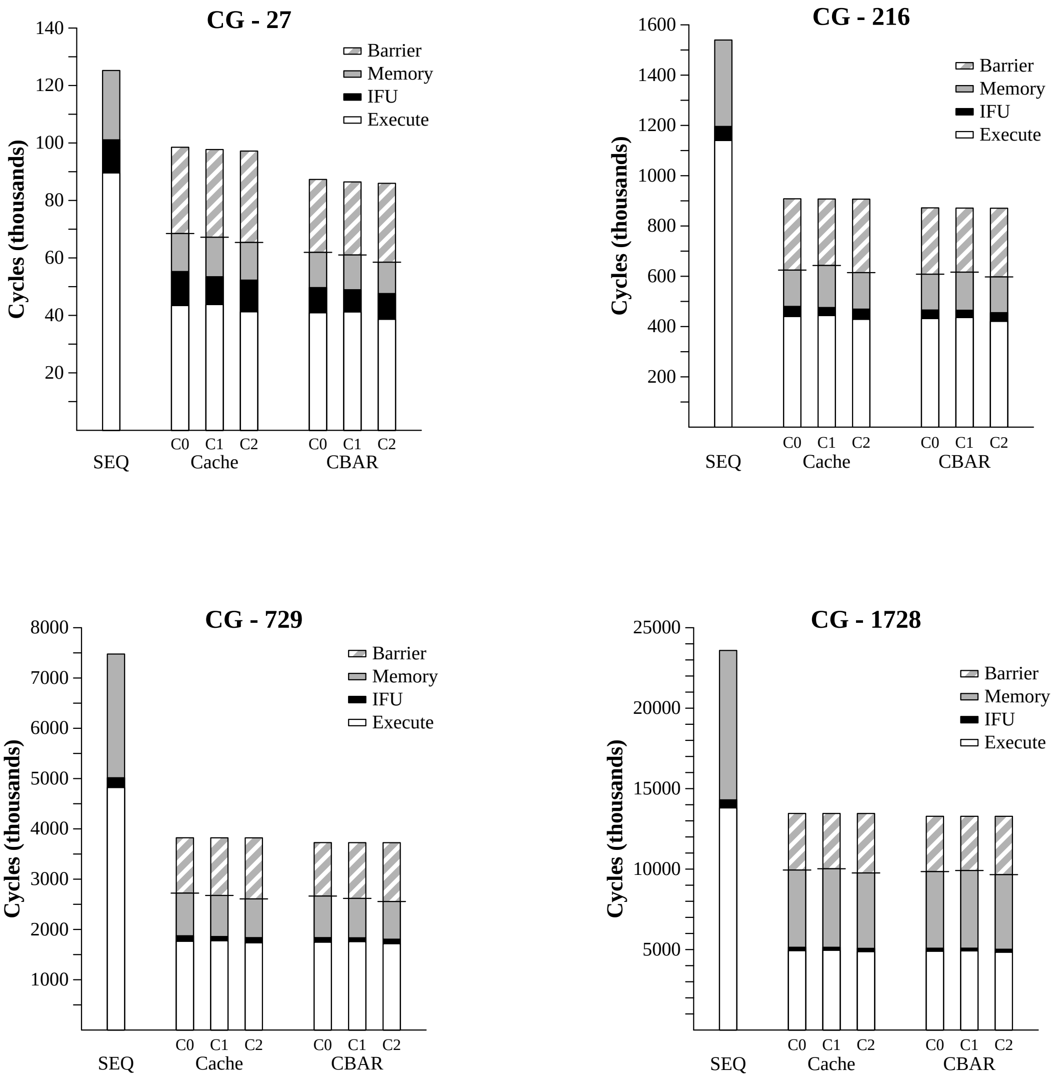


Figure B.10: CG cycle breakdown using outer-loop parallelism.

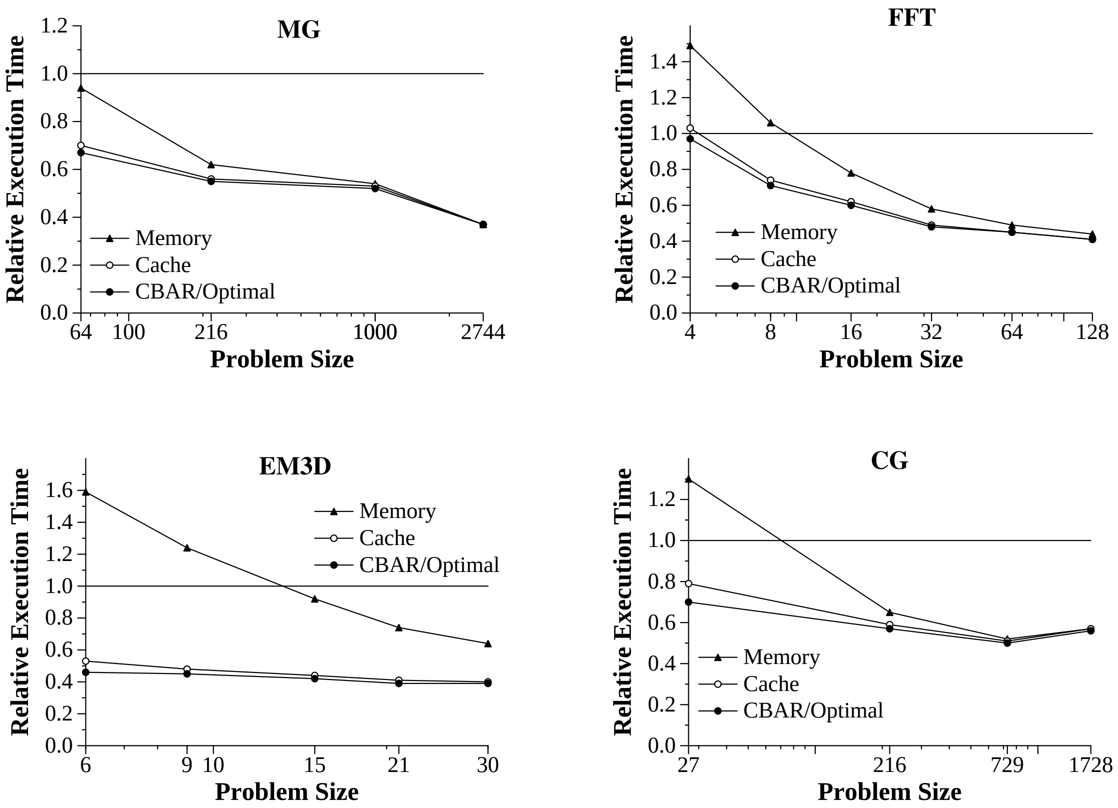


Figure B.11: Summary of outer–loop execution times.

Bibliography

[ABC+95] Anant Agarwal, Ricardo Bianchini, David Chaiken, Kirk L. Johnson, David Kranz, John Kubiatowicz, Beng-Hong Lim, Kenneth Mackenzie, and Donald Yeung. The MIT alewife machine: Architecture and performance. In *Proceedings of the 22nd Annual International Symposium on Computer Architecture*, pages 2–13, 1995.

[ACC+90] Robert Alverson, David Callahan, Daniel Cummings, Brian Koblenz, Allan Porterfield, and Burton Smith. The Tera computer system. In *Proceedings of the International Conference on Supercomputing*, pages 1–6, June 1990.

[AKK+93] Anant Agarwal, John Kubiatowicz, David Kranz, Beng-Hong Lim, Donald Yeung, Godfrey D'Souza, and Mike Parkin. Sparcle: an evolutionary processor design for large-scale multiprocessors. *IEEE Micro*, 13(3):48–61, June 1993.

[ALKK90] Anant Agarwal, Beng-Hong Lim, David Kranz, and John Kubiatowicz. APRIL: A processor architecture for multiprocessing. In *Proceedings of the 17th Annual International Symposium on Computer Architecture*, pages 104–114, 1990.

[BCC+90] S. Borkar, R. Cohn, G. Cox, T. Gross, H.T. Kung, M. Lam, M. Levine, B. Moore, W. Moore, C. Peterson, J. Susman, J. Sutton, J. Urbanski, and J. Webb. Supporting systolic and memory communication in iWarp. In *Proceedings of the 17th International Symposium on Computer Architecture*, pages 70–81, May 1990.

[Boh96] Mark T. Bohr. Interconnect scaling – the real limiter to high performance ULSI. *Solid State Technology*, 39(9):105–111, September 1996.

[Car98] Nicholas P. Carter. *Hardware Support for Software Shared Memory on the M-Machine.* PhD thesis, Massachusetts Institute of Technology, Department of Electrical Engineering and Computer Science, expected 1998.

[CDG+93] David E. Culler, Andrea Dusseau, Seth Copen Goldstein, Arvind Krishnamurthy, Steven Lumetta, Thorsten von Eiken, and Katherine Yelick. Parallel programming in Split-C. In *Supercomputing*, pages 262–273, November 1993.

[Cha98] Andrew Chang. VLSI datapath choices: Cell-based versus full-custom. Master's thesis, Massachusetts Institute of Technology, Department of Electrical Engineering and Computer Science, February 1998.

[CKD94] Nicholas P. Carter, Stephen W. Keckler, and William J. Dally. Hardware support for fast capability-based addressing. In *Proceedings of the Sixth International Conference* *on Architectural Support for Programming Languages and Operating Systems (ASPLOS VI)*, pages 319–327, October 1994.

[CLB+96] Frederic T. Chong, Beng-Hong Lim, Ricardo Bianchini, John Kubiatowicz, and Anant Agarwal. Application performance on the MIT alewife machine. *IEEE Computer*, 29(12):57–64, December 1996.

[CLMY96] David E. Culler, Lok Tin Liu, Richard P. Martin, and Chad O. Yoshikawa. Assessing fast network interfaces. *IEEE Micro*, 16(1):35–43, February 1996.

[CNO+88] Robert P. Colwell, Robert P. Nix, John J. O'Donnell, David B. Papworth, and Paul K. Rodman. A VLIW architecture for a trace scheduling compiler. *IEEE Transactions on Computers*, 37(8):967–979, August 1988.

[CPWG97] Alan Charlesworth, Andy Phelps, Ricki Williams, and Gary Gilbert. Gigaplane-XB: Extending the ultra enterprise family. In *Proceedings of Hot Interconnects V*, pages 97–112, August 1997.

[CSY90] Ding-Kai Chen, Hong-Men Su, and Pen-Chung Yew. The impact of synchronization and granularity on parallel systems. In *Proceedings of the 17th International Symposium on Computer Architecture*, pages 239–248, May 1990.

[Dil97] Thomas J. Dillon. The VelociTI architecture of the TMS320C6xxx. In *Proceedings of the International Conference on Signal Processing Applications and Technology*, September 1997.

[DKC+94] William J. Dally, Stephen W. Keckler, Nick Carter, Andrew Chang, Marco Fillo, and Whay S. Lee. The MAP instruction set reference manual v1.0. Concurrent VLSI Architecture Memo 59, Massachusetts Institute of Technology, Artificial Intelligence Laboratory, January 1994.

[DLD93] Larry R. Dennison, Whay S. Lee, and William J. Dally. High-performance bidirectional signalling in VLSI systems. In *Proceedings of the Symposium on Research on Integrated Systems*, pages 300–319. MIT Press, March 1993.

[EL90] Miloš D. Ercegovac and Tomas Lang. Radix-4 square root without initial PLA. *IEEE Transactions on Computers*, 39(8):1016–1024, August 1990.

[Fan87] Jan Fandrianto. Algorithm for high speed shared radix 4 division and radix 4 square root. In *Proceedings of the 8th Symposium on Computer Arithmetic*, pages 73–79, May 1987.

[FD95] Stuart Fiske and William J. Dally. Thread prioritization: A thread scheduling mechanism for multiple-context parallel processors. In *Proceedings of the First IEEE Symposium on High-Performance Computer Architecture*, pages 210–221, Raleigh, NC, January 1995.

[FKD+95] Marco Fillo, Stephen W. Keckler, William J. Dally, Nicholas P. Carter, Andrew Chang, Yevgeny Gurevich, and Whay S. Lee. The M-Machine Multicomputer. In *Proceedings of the 28th International Symposium on Microarchitecture*, pages 146–156, Ann Arbor, MI, December 1995.

[FS96] Manoj Franklin and Gurinar S. Sohi. ARB: a hardware mechanism for dynamic reordering of memory references. *IEEE Transactions on Computers*, 45(5):552–571, May 1996.

[GAB+97] David Greenhill, Eric Anderson, James Bauman, Andrew Charnas, Rakesh Cheerla, Hao Chen, Manjunath Doreswamy, Phillip Ferolito, Srinivasa Gopaladhine, Kenneth Ho, Wenjay Hsu, Poonacha Kongetira, Ronald Melanson, Vinita Reddy, Raoul Salem, Harikaran Sathianathan, Shailesh Shah, Ken Shin, Chakra Srivatsa, and Robert Weisenbach. A 330mhz 4-way superscalar microprocessor. In *Proceedings of the IEEE International Solid-State Circuits Conference*, pages 166–167, February 1997.

[GKT91] Gina Goff, Ken Kennedy, and Chau-Wen Tseng. Practical dependence testing. In *Proceedings of ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 15–29, June 1991.

[Gur94] Yevgeny Gurevich. An assembler and linker system for the M-machine software project. Bachelor's Thesis, Massachusetts Institute of Technology, Department of Electrical Engineering and Computer Science, April 1994.

[Gur95] Yevgeny Gurevich. The M-Machine operating system. Master's thesis, Massachusetts Institute of Technology, Department of Electrical Engineering and Computer Science, September 1995.

[Gwe95] Linley Gwennap. Processor performance climbs steadily. *Microprocessor Report*, January 1995.

[Gwe96] Linley Gwennap. Digital 21264 sets new standard. *Microprocessor Report*, October 1996.

[Har96] Daniel K. Hartman. M-Machine floating-point multiplier datapath. Master's thesis, Massachusetts Institute of Technology, Department of Electrical Engineering and Computer Science, May 1996.

[IEE85] IEEE std. 754-1985, standard for binary floating-point arithmetic, 1985.

[Jou90] Norman P. Jouppi. Improving direct-mapped cache performance by the addition of a small fully-associative cache and prefetch buffers. In *Proceedings of the 17th International Symposium on Computer Architecture*, pages 364–373, 1990.

[KD92] Stephen W. Keckler and William J. Dally. Processor coupling: Integrating compile time and runtime scheduling for parallelism. In *Proceedings of the 19th International Symposium on Computer Architecture*, pages 202–213, May 1992.

[KHM89] David A. Kranz, Robert H. Halstead, and Eric Mohr. Mul-T: A high-performance Lisp. In *Sigplan '89 Symposium on Programming Language Design and Implementation*, pages 1–10, June 1989.

[LA94] Beng-Hong Lim and Anant Agarwal. Reactive synchronization algorithms for multiprocessors. In *Sixth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-VI)*, pages 25–35, October 1994.

[LAD+96] C.E. Leiserson, Z.S. Abuhamdeh, D.C. Douglas, C.R. Feynman, M.N. Ganmukhi, J.V. Hill, W.D. Hillis, B.C. Kuszmaul, M.A. St Pierre, D.S. Wells, M.C. Wong-Chan, Shaw-Wen Yang, and R. Zak. The network architecture of the connection machine CM-5. *Journal of Parallel and Distributed Computing*, 33(2):145–158, March 1996.

[Lam88] Monica Lam. Software pipelining: An effective scheduling technique for VLIW machines. In *SIGPLAN '88 Conference on Programming Language Design and Implementation*, pages 318–328. ACM, ACM, June 1988.

[LDK+98] Whay Sing Lee, William J. Dally, Stephen W. Keckler, Nicholas P. Carter, and Andrew Chang. Efficient, protected message interface in the MIT M-Machine. To appear in the IEEE Computer Special Issue on Design Challenges for High-Performance Network Interfaces, November 1998.

[LFK+93] P. G. Lowney, S. G. Freudenberger, T. J. Karzes, W. D. Lichtenstein, R. P. Nix, J. S. O'Donnell, and J. C. Ruttenberg. The multiflow trace scheduling compiler. *The Journal of Supercomputing*, 7(1-2):51–142, May 1993.

[LL97] James Laudon and Daniel Lenoski. The SGI Origin: a ccNUMA highly scalable server. In *Proceedings of the 24th International Symposium on Computer Architecture*, pages 241–251, June 1997.

[MLC+92] Scott A. Mahlke, David C. Lin, William Y. Chen, Richard E. Hank, and Roger A. Bringman. Effective compiler support for predicated execution using the hyperblock. In *Proceedings of the 25th International Symposium on Microarchitecture*, pages 45–54. ACM, December 1992.

[Moo95] Gordon E. Moore. Lithography and the future of Moore's law. In *SPIE Vol. 2440 Proceedings of the International Society for Optical Engineering*, pages 2–17, February 1995.

[MVCA97] Richard P. Martin, Amin M. Vahdat, David E. Culler, and Thomas E. Anderson. Effects of communication latency, overhead, and bandwidth in a cluster architecture. In *Proceedings of the 24th International Symposium on Computer Architecture*, pages 85–97, June 1997.

[NHO96] Basem A. Nayfeh, Lance Hammond, and Kunle Olukotun. Evaluation of design alternatives for a multiprocessor microprocessor. In *Proceedings of the 23rd International Symposium on Computer Architecture*, pages 67–77, May 1996.

[NPA92] Rishiyur S. Nikhil, Gregory M. Papadopoulos, and Arvind. *T: A multithreaded massively parallel architecture. In *Proceedings of the 19th International Symposium on Computer Architecture*, pages 156–167, May 1992.

[NWD93] Michael D. Noakes, Deborah A. Wallach, and William J. Dally. The J-Machine multicomputer: An architectural evaluation. In *Proceedings of the 20th International Symposium on Computer Architecture*, pages 224–235, San Diego, California, May 1993.

[PC90] Gregory M. Papadopoulos and David E. Culler. Monsoon: an explicit token-store architecture. In *The 17th Annual International Symposium on Computer Architecture*, pages 82–91. IEEE, 1990.

[PSW91] C. Peterson, J. Sutton, and P. Wiley. iWarp: a 100-MOPS, LIW microprocessor for multicomputers. *IEEE Micro*, 11(3):26–29, 81–87, June 1991.

[RR87] Kay A. Robbins and Steven Robbins. *The Cray X-MP/Model 24*. Springer-Verlag, 1987.

[SBV95] Gurindar S. Sohi, Scott E. Breach, and T.N. Vijaykumar. Multiscalar processors. In *Proceedings of the 22nd International Symposium On Computer Architecture*, pages 414–425, May 1995.

[Sem97] The national technology roadmap for semiconductors. Semiconductor Industry Association, 1997.

[SLH90] Michael D. Smith, Monica S. Lam, and Mark A. Horowitz. Boosting beyond static scheduling in a superscalar processor. In *Proceedings of the 17th International Symposium on Computer Architecture*, pages 344–354, June 1990.

[SPE92] Spec benchmark release v1.1, 1992.

[TEL95] Dean M. Tullsen, Susan J. Eggers, and Henry M. Levy. Simultaneous multithreading: Maximizing on-chip parallelism. In *Proceedings of the 22nd International Symposium On Computer Architecture*, pages 392–403, May 1995.

[TM91] Donald E. Thomas and Philip Moorby. *The Verilog Hardware Description Language.* Kluwer Academic Publishers, 1991.

[Tom67] R.M. Tomasulo. An efficient algorithm for exploiting multiple arithmetic units. *IBM Journal*, 11:25–33, January 1967.

[TW92] Than Tran and Chuan-lin Wu. Limitation of superscalar microprocessor performance. In *Proceedings of the 25th International Symposium on Microarchitecture*, pages 33–36, December 1992.

[Wal91] David W. Wall. Limits of instruction-level parallelism. In *Proceedings of the Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 176–188. ACM, 1991.

[WGH+97] Wolf-Dietrich Weber, Stephen Gold, Pat Helland, Takeshi Shimizu, Thomas Wicki, and Winfried Wilcke. The mercury interconnect architecture: A cost-effective infrastructure for high-perfomance servers. In *Proceedings of the 24th International Symposium on Computer Architecture*, pages 98–107, June 1997.

[YA93] Donald Yeung and Anant Agarwal. Experience with fine-grain synchronization in MIMD machines for preconditioned conjugate gradient. In *Proceedings of the 4th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 187–197, May 1993.