A Coupled Multi-ALU Processing Node for a Highly Parallel Computer

by

Stephen William Keckler

Submitted to the
Department of Electrical Engineering and Computer Science
on May 7, 1992, in partial fulfillment of
the requirements for the Degree of Master of Science in
Electrical Engineering and Computer Science

Abstract

By 1995, improvements in semiconductor technology will allow as many as four high-performance floating-point ALUs and several megabits of memory to reside on a single chip. Multiple arithmetic units can be organized into a single processor to exploit instruction-level parallelism.

Processor Coupling is a mechanism for controlling multiple ALUs to exploit both instruction-level and inter-thread parallelism. Processor Coupling employs both compile time and run-time scheduling. The compiler statically schedules individual threads to discover available intra-thread instruction-level parallelism. The runtime scheduling mechanism interleaves threads, exploiting inter-thread parallelism to maintain high ALU utilization. ALUs are assigned to threads on a cycle by cycle basis, and several threads can be active concurrently.

This thesis describes the architecture of a Processor Coupled node and evaluates parameters that affect performance. Processor Coupling is compared with a multiple ALU statically scheduled machine model and a multiprocessor model in which threads are executed on different arithmetic units. The experiments address the effects of memory latencies, function unit latencies, and communication bandwidth between function units. The instruction scheduling compiler and the configurable simulator that were built to evaluate these different organizations are described. An implementation and feasibility study of a Processor Coupled node is presented as well.

On four benchmark programs Processor Coupling runs in 60% fewer cycles than a statically scheduled node. Processor Coupling is more tolerant of latencies executing 75% fewer cycles when memory delays are included, and 65% fewer cycles when floating point latencies are increased. On threaded code, the multiprocessor and Processor Coupled nodes achieve nearly the same performance. On sequential sections of code, the multiprocessor organization requires on average 2.9 times as many cycles as the Processor Coupled model.

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

Keywords: Instruction Level Parallelism, Compile Time Scheduling, Runtime Scheduling, Multithreading, Parallel Computers.

Acknowledgments

Many people have contributed to this work and to my MIT experience. I would like to thanks them all.

Thanks to Bill Dally, my mentor and thesis advisor. He provided the vision, encouragement, and technical expertise that made this work possible. I am in debt to him for bringing me to Cambridge and for sending me to Turkey and Australia.

Other members of the CVA group were of great help as well. Thanks to Rich Lethin for being an excellent officemate both in his helpful suggestions to this work and in his attempts at improving my social life (at least he did get me into sailing). Thanks to world traveler Stuart Fiske who arrived back in Cambridge just in time to ask important questions and to plow through drafts of the paper and this thesis. Thanks to Debby Wallach for her careful review of early drafts of the paper. Although she probably does not want to known to be a Latex and Postscript guru, her help was invaluable in getting the text tools to behave. Thanks to Mike Noakes and Ellen Spertus for their eleventh hour proof reading. Finally, I am grateful to the entire CVA group for their feedback and for providing both support and skepticism.

Thanks go to those outside the CVA group as well. Thanks to Ricardo Telichevesky for his help in formulating the **LUD** benchmark. Donald Yeung, in addition to being a good housemate, provided formal and informal feedback. Thanks also to Bill Hall for dragging me out of the lab to go sailing and skiing.

Most of all, thanks go to my family for all of their encouragement throughout the years. Thanks Mom for laughing at my jokes and for being there when I needed you. Thanks Dad for challenging me to be my best and for providing many words of wisdom. Without their support this journey would have seemed so much further.

The research described in this thesis was supported in part by a National Defense Science and Engineering Graduate Fellowship, by the Defense Advanced Research Projects Agency under contracts N00014-88K-0738, N00014-87K-0825, F19628-92-C-0045, and N00014-91-J-4038, and by a National Science Foundation Presidential Young Investigator Award, grant MIP-865-7531, with matching funds from General Electric Corporation, IBM Corporation, and AT&T.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Instruction-level Parallelism

By 1995, improvements in semiconductor technology will allow multiple high performance floating point units and several megabits of memory to reside on a single chip. One possible use of these multiple arithmetic units is to organize them in a single processor to exploit instruction-level parallelism. Controlling multiple function units on a single processor presents a challenge. Applications exhibit an uneven amount of instruction-level parallelism during their execution [JW89]. In some parts of a program, all of the function units will be used, while in others only some will be used since serial computations with little instruction-level parallelism dominate. The amount of available parallelism depends upon both the computation at hand and the accessibility of data. Long memory latencies can stifle opportunities to exploit instruction-level parallelism. Conditional branch operations also reduce the amount of available instruction-level parallelism, especially in processors which use only runtime mechanisms for scheduling.

The ideal multiple function unit processing node allows a task to use as many function units as it needs, but also allocates unused units to other tasks. In addition to being effective as a uniprocessor, such a node serves well in a parallel computing environment with many available tasks. Furthermore, exploiting instruction-level parallelism in conjunction with coarser grained algorithmic concurrency improves machine performance.

1.2 Technological Advances

Microprocessor performance has been increasing at an exponential rate since the mid-1970s. Much of the speedup has been due to improvements in integrated circuit fabrication technology. The effective silicon area in λ^2 available to circuit designers has been increasing due to smaller feature sizes and larger die areas.[1] Single-chip microprocessors announced in early 1992 have more than 3 million transistors [AN$^+$92]. In 1995 CMOS chips will be $17.5mm$ on a side and have transistors with 0.5μ gate lengths. In addition, they will have 3–4 layers of metal interconnect.

This expanse of chip area will provide opportunities for architects to create new, highly integrated designs that have not yet been possible. At these densities 8Mbits of SRAM can be built along with several integer and floating point units, including interconnect. Integrating these components on a single chip will allow experimentation with low latency interaction between function units. Such exploration will likely yield new methods of building high performance computers. Processor Coupling is intended to be one of those methods.

1.3 Processor Coupling

This thesis introduces Processor Coupling, a runtime scheduling mechanism in which multiple function units execute operations from multiple instruction streams and place results directly in each other's register files. Several threads may be active simultaneously, sharing use of the function unit pipelines. Instruction-level parallelism within a single thread is exploited using static scheduling techniques similar to those demonstrated in the Multiflow Trace system [CNO$^+$88]. At runtime, the hardware scheduling mechanism interleaves several threads, exploiting inter-thread parallelism to maintain high utilization of function units.

Figure 1.1 demonstrates how Processor Coupling dynamically interleaves instruction streams from multiple threads across multiple function units. The operations from threads **A**, **B**, and **C** are scheduled independently at compile time as shown in the top of the

[1] The parameter λ is, to first order, process independent and is equivalent to one half of the minimum feature size. For a $0.5\mu m$ process, λ is 0.25.

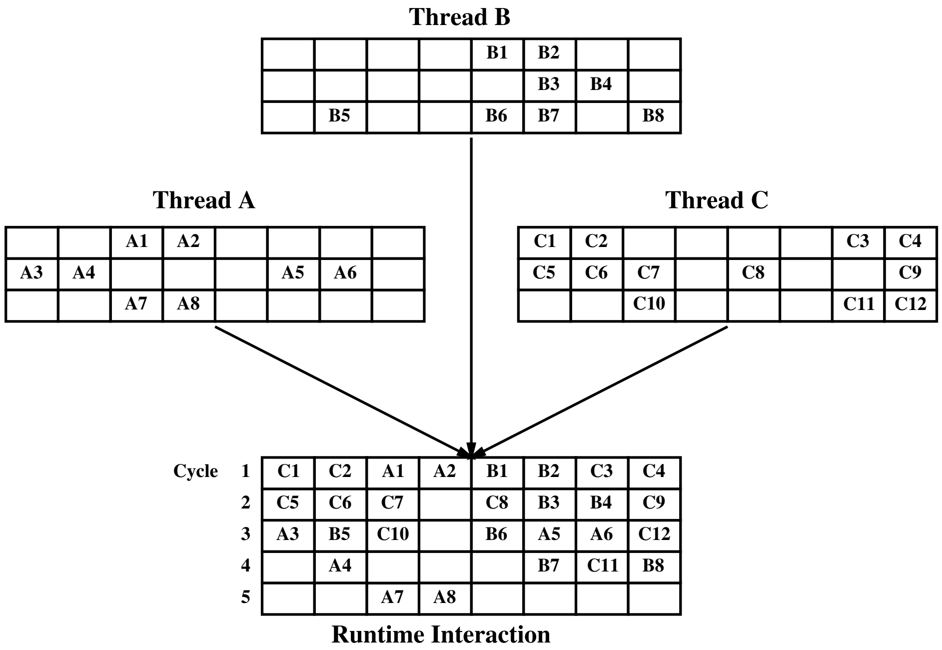

Figure 1.1: Interleaving of instruction streams. Threads A, B, and C are scheduled separately and their instruction streams are shown at the top of the diagram. Each column is a field for a particular function unit and each row holds those operations which are allowed to execute simultaneously. The bottom box shows a runtime interleaving of these threads in which some operations are delayed due to function unit conflicts.

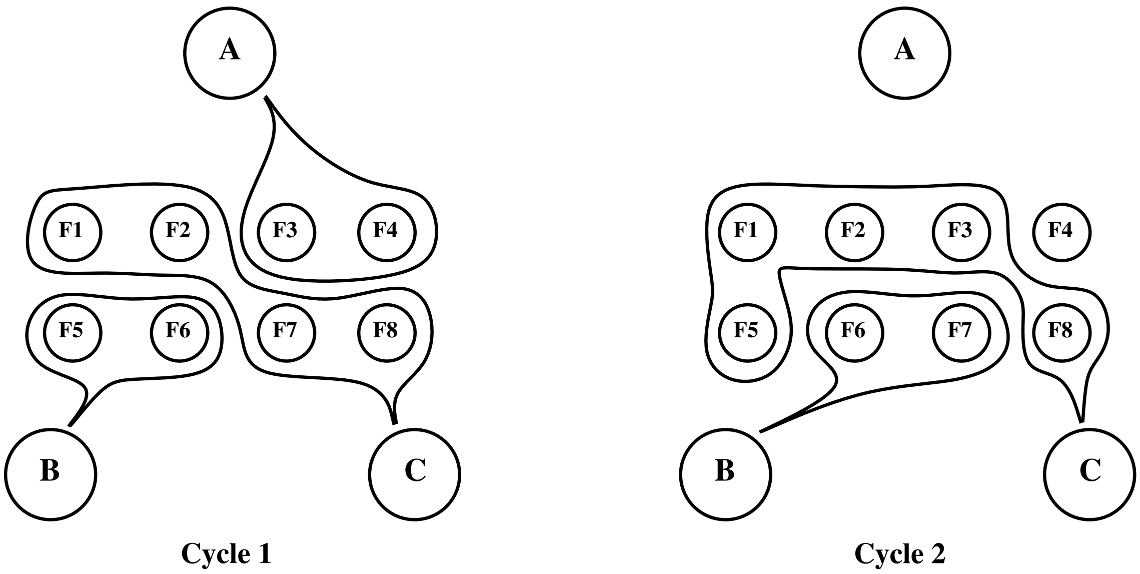

Figure 1.2: Two possible mappings of function units to threads. These mappings correspond to the first two cycles shown in Figure 1.1.

figure. Each column in a thread's instruction stream represents an operation field for a single function unit. Each row holds operations that may be executed simultaneously. The empty boxes indicate that there is insufficient instruction-level parallelism to keep all of the function units busy. During execution, arbitration for function units is performed on a cycle by cycle basis. When several threads are competing for a given function unit, one is granted use and the others must wait. For example, operations **A3** and **A4** are blocked during the second cycle because thread **C** is granted those units instead. Figure 1.2 illustrates the mapping of function units to threads, as a result of runtime arbitration, for the first two cycles shown in Figure 1.1.

Note that operations scheduled in a single long instruction word need not be executed simultaneously. Allowing the static schedule to slip provides for finer grain sharing of function units between threads. In Figure 1.1, operations **A3**-**A6** are scheduled on the same instruction word in thread **A**. Operations **A3**, **A5**, and **A6** are all issued during cycle 3, while **A4** is not issued until cycle 4. However, **A4** must be issued before **A7** and **A8**.

Figure 1.3 shows the allocation of function units to threads for a statically scheduled VLIW machine and for a multiprocessor. A VLIW machine generally has a single thread which is allowed to use all of the function units all of the time. Conventional threaded machines with multiple pipelines also statically allocate function units to threads. A thread is assigned to a particular function unit and may not use function units belonging to other threads.

A compiler can be used to extract the statically available instruction-level parallelism from a program fragment. However, compile time scheduling is limited by unpredictable memory latencies and by some dependencies, such as data dependent array references, which cannot be statically determined. Furthermore, although trace scheduling [Ell86] and software pipelining techniques [Lam88] can be used, branch boundaries tend to limit the number of operations that can be scheduled simultaneously. By interleaving multiple threads, the hardware runtime scheduling mechanisms of Processor Coupling address the limits of static scheduling due to dynamic program behavior.

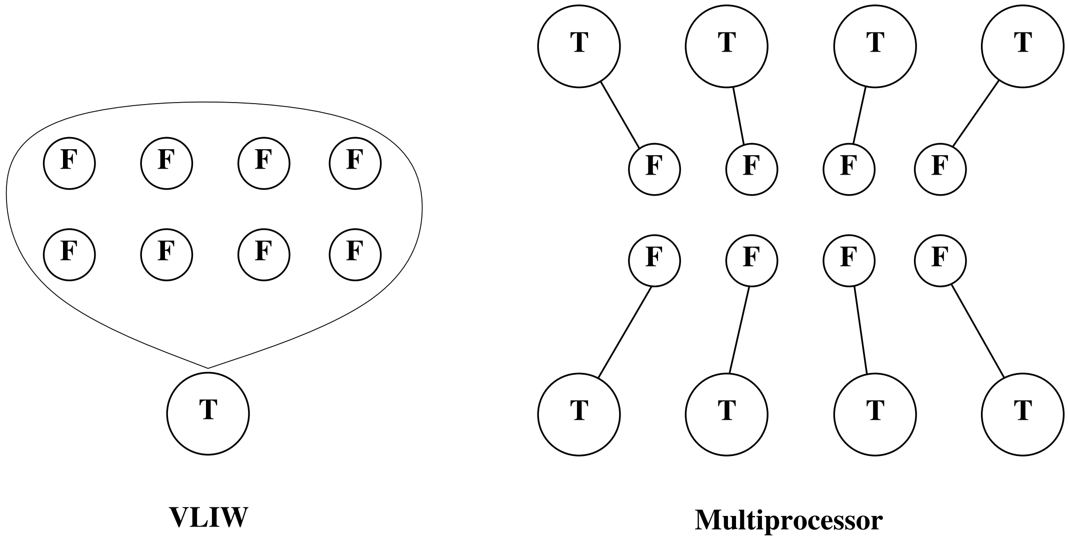

Figure 1.3: In a VLIW machine all of the function units are used by a single thread. The compiler uses as many units as it can and does not share them with other threads. In a purely multiprocessor organization, different threads run on different processing units and cannot couple multiple units together.

1.4 Processor Coupling Performance

The experimental environment built to evaluate Processor Coupling includes both a compiler that schedules operations and generates assembly code, and a simulator that runs the code and generates statistics. The compiler and the simulator can be configured to simulate different machine models including purely statically scheduled, multiprocessor, and Processor Coupled nodes. Four benchmarks are used to evaluate these different configurations.

The base node used in simulation has four integer ALUs, four floating point units, four memory units, and one branch unit. With single cycle memory latencies Processor Coupling executes in 60% fewer cycles than a statically scheduled node. For a probabilistic memory model with variable memory latencies, Processor Coupling executes in 75% fewer cycles. Processor Coupling is also more tolerant to function unit latencies. When the floating point unit latency is increased from 1 to 5 cycles, Processor Coupling only needs 10% more cycles, while the statically scheduled node requires 90% more cycles.

On threaded code the multiprocessor and Processor Coupled nodes achieve nearly the same performance. However, on sequential sections of code, the Processor Coupled node is able to use all of the function units while the multiprocessor node is restricted to using only those function units within a single cluster. In sequential sections the multiprocessor organization requires on average 2.9 times as many cycles as the Processor Coupled model. On the benchmark that has a sequential section, the multiprocessor node must execute 79% more cycles than the coupled node.

1.5 Thesis Overview

This thesis describes Processor Coupling architecture and its implementation issues, the compiler and simulator which comprise the experimental environment, and the evaluation of Processor Coupling performance.

Chapter 2 describes work in related fields that has contributed to the ideas in this thesis. Chapter 3 presents an architectural overview of a Processor Coupled node, including the partitioning of function units into clusters. Intra-thread synchronization, thread control, and the requirements of the memory system are described as well.

The experimental environment consists of a compiler and a simulator. Chapter 4 presents ISC, a prototype compiler for Processor Coupling. ISC partitions source level programs, schedules individual operations in wide instruction words, and produces simulator assembly code. PCS, the simulator described in Chapter 5, runs the code and generates statistics such as cycle counts and unit utilization data.

Chapter 6 describes the experimental process, including the benchmark suite, experimental assumptions, and performance evaluation. The performance of statically scheduled, multiprocessor, and Processor Coupled nodes is compared under conditions of variable memory latencies, different floating point unit latencies, and different interconnection strategies between function units. In addition, several methods of expressing parallel loops are evaluated for multiprocessor and Processor Coupled configurations. Different mechanisms for moving data between function units in a coupled node are explored as well.

Chapter 7 sketches an implementation of Processor Coupling, including a description of the pipeline and synchronization mechanisms. It also presents a plan for building a Processor Coupled node on a single integrated circuit with 1995 technology. Finally, Chapter 8 summarizes the conclusions of this thesis and proposes directions for further research.

Chapter 2

Background

Processor Coupling is influenced by a variety of fine grained parallelism methods, including compile time scheduling techniques, superscalar designs, and multithreaded machines. Fisher and Rau [FR91] summarize many means of exploiting instruction-level parallelism. Processor Coupling integrates certain aspects of these techniques in order to achieve higher instruction throughput and better function unit utilization without extreme hardware costs.

2.1 Superscalar Processors

Strict superscalar processors execute multiple instructions simultaneously by relying upon runtime scheduling mechanisms to determine all data dependencies. Current superscalar designs assume little aid from a compiler and are considered to be object code compatible with their purely sequential counterparts. Much of the current work in superscalar technology has stemmed from the dependency analysis and reservation stations used in the IBM 360/91 floating point processor [Tom67]. Many different mechanisms to allow hardware to discover instruction-level parallelism have since been developed and are discussed by Johnson in [Joh91]. Instructions within a limited size window are examined for dependencies and are selected for issue. Register renaming hardware is used to resolve register resource conflicts created by the compiler that are not related to data dependencies. Out of order instruction issue complicates exception handling since it is difficult to determine where to restart the program. A history buffer can be used to unroll the already executed instructions to a point where computation can begin again. To achieve higher performance many superscalar designs rely upon speculative execution. When a branch instruction is reached, the hardware predicts which direction is taken and begins to execute instructions down that path. If the wrong direction was selected, the state of the machine must be rolled back before execution can continue.

The Metaflow architecture is a current design that incorporates many of these superscalar techniques [PSS+91]. This design uses a complex instruction shelving mechanism that analyzes dependencies, issues instructions out of order, predicts branches, commits results to the register file, and eliminates uncommitted registers to roll back mispredicted branch code. Special register files with several read and write ports are necessary. The designers estimate that four chips will be necessary to implement this design.

The analysis by Wall [Wal91] shows that for an idealized machine with an ambitious branch prediction scheme superscalar processors can only hope to achieve on average between 5 and 10 instructions per cycle. Considering the hardware complexity involved in runtime dependency checking and instruction scheduling, this is not very promising. The compile time techniques described in the next section have demonstrated that much more instruction-level parallelism can be exploited by the compiler without the aid of runtime scheduling hardware.

2.2 Compile Time Scheduling

Compile time scheduling has been used to enhance the amount of instruction-level parallelism available in scientific programs. Assuming perfect static branch prediction, Nicolau and Fisher [NF84] report finding average instruction-level parallelism of 90 on a variety of Fortran programs. Very Long Instruction Word (VLIW) processors such as the Multiflow Trace series [CHJ+90] employ only compile time scheduling to manage instruction-level parallelism and resource use. No hardware synchronization logic is required as all resource scheduling is done by the compiler. Compilation is performed using trace scheduling mechanisms based on those in the Bulldog compiler [Ell86]. This compiler represents a program as a directed acyclic graph (DAG) of basic blocks. The scheduler traverses the graph, choosing the most likely path using static branch prediction techniques. All of the instructions in this path are scheduled and allocated to multiple function units as if they were in the same basic block. The scheduler then chooses the most likely path through the remaining blocks in the DAG. This continues until all blocks have been scheduled. The compiler must generate compensation code for the off trace blocks in the event of a mispredicted branch. Trace scheduling uses loop unrolling to extract additional instruction-level parallelism.

Another compiler method for handling loop scheduling is software pipelining [Lam88]. In this technique, a loop is decomposed into resource independent blocks. The loop is unrolled enough times to overlap execution of different iterations of the original loop. If the loop is meant to terminate in the middle of the pipeline, additional code is required to undo the work of the extra iterations. This method complements trace scheduling well in that it achieves excellent scheduling of loop code without the large increase in code size seen with uncontrolled loop unrolling.

The weakness of the compiler scheduling the code without hardware interlocks involves uncertainty in the latency of operations. In a distributed memory machine, a memory access might require communication between remote nodes. Since the latencies cannot be known until runtime, the compiler is unable to schedule resources without hardware assistance.

2.3 Multithreaded Architectures

Using multiple threads to hide memory latencies and pipeline delays has been explored in several different studies and machines. Gupta and Weber explore the use of multiple hardware contexts in multiprocessors [GW89]. Their machine model switches threads whenever a long latency memory reference occurs, such as a cache miss or a write-hit to shared data. Their results show that multiple contexts are most effective when memory latencies are long and context switch time is small. The Alewife machine [ALKK90] uses this type of coarse grained multithreading to tolerate memory latencies in a shared memory environment.

MASA [HF88] as well as HEP [Smi81] use fine grain multithreading to issue an instruction from a different context on every cycle in order to mask pipeline latencies. An instruction from one thread cannot be issued until its previous one has completed. Although these approaches reduce the complexity of the processor by eliminating complicated scheduling mechanisms, single thread performance is degraded by the number of pipeline stages.

Processor Coupling with the capability, but not the requirement, of switching threads every cycle achieves single thread performance comparable to that of a VLIW.

A dataflow approach described by Arvind and Culler [AC86] attempts to exploit parallelism by decomposing programs into single instruction threads that are scheduled and executed as their data dependencies are satisfied. Because of the overhead associated with scheduling each operation, other research such as Ianucci [Ian88] and Culler [CSS+91] have suggested increasing the scheduling granularity by composing multiple operations into groups. This composite then becomes the element that is scheduled by the hardware. They do not, however, address instruction-level parallelism within a group.

2.4 Multiple ALUs and Multiple Threads

Daddis and Torng [DT91] simulate multiple instruction streams on superscalar processors. They interleave instruction fetching between two active threads and dispatch instructions for execution based upon dynamically checked data dependencies. Their speedup comes from masking instruction cache miss latencies and data dependencies within one thread. They report speedups of between 90 and 100 percent for two active threads.

The proposed XIMD [WS91] architecture employs compile time techniques to statically schedule instructions as well as threads. The XIMD compiler determines the function unit requirements of each thread. It then packs the threads into a schedule so that no two threads require the same function unit simultaneously. Threads synchronize by blocking on globally visible synchronization signals. Function units with blocked threads sit idle since threads do not share function unit pipelines. Processor Coupling removes these constraints by allowing runtime scheduling mechanisms to determine allocation of function units to threads. Furthermore, Processor Coupling synchronizes through memory so that a function unit can work on other threads when one thread is blocked.

2.5 The M-Machine

Processor Coupling is useful in machines ranging from workstations based upon a single multi-ALU node to massively parallel machines such as the MIT M-Machine, which is currently being designed. The M-Machine will consist of thousands of multi-ALU Processor Coupled nodes and will have many threads to be interleaved at each node. The machine will thus take advantage of a hierarchy of parallelism, ranging from coarse-grained algorithmic parallelism to extremely fine-grained instruction-level parallelism. However, as will be demonstrated in Chapter 6, Processor Coupling can be effective even on a single node. This thesis will consider only a single node instance of Processor Coupling.

Chapter 3

Architectural Overview

The architecture of a Processor Coupled node has two main components. First, the low interaction latency between function units on a single chip is exploited for instruction-level parallelism within a single thread. Multiple function units issue instructions from a common stream and place their results directly in each other's register files. Secondly, multiple threads run concurrently to increase function unit and memory bandwidth utilization. Resources such as register files and process state registers are duplicated to allow multiple threads to be active simultaneously.

The architectural description in this chapter provides an overview of how a Processor Coupled node functions and identifies features not usually found in commercial microprocessors, including mechanisms for synchronization through presence bits in registers and in memory. Since this thesis neither defines a complete microarchitecture nor presents any logic design, many details will not be addressed.

3.1 A Processor Coupled Node

3.1.1 Node Organization

A Processor Coupled node, as shown in Figure 3.1 consists of a collection of function units, register files, memory banks, and interconnection networks. A function unit may perform integer operations, floating point operations, branch operations, or memory accesses. Function units are grouped into *clusters*, sharing a register file among them. The register

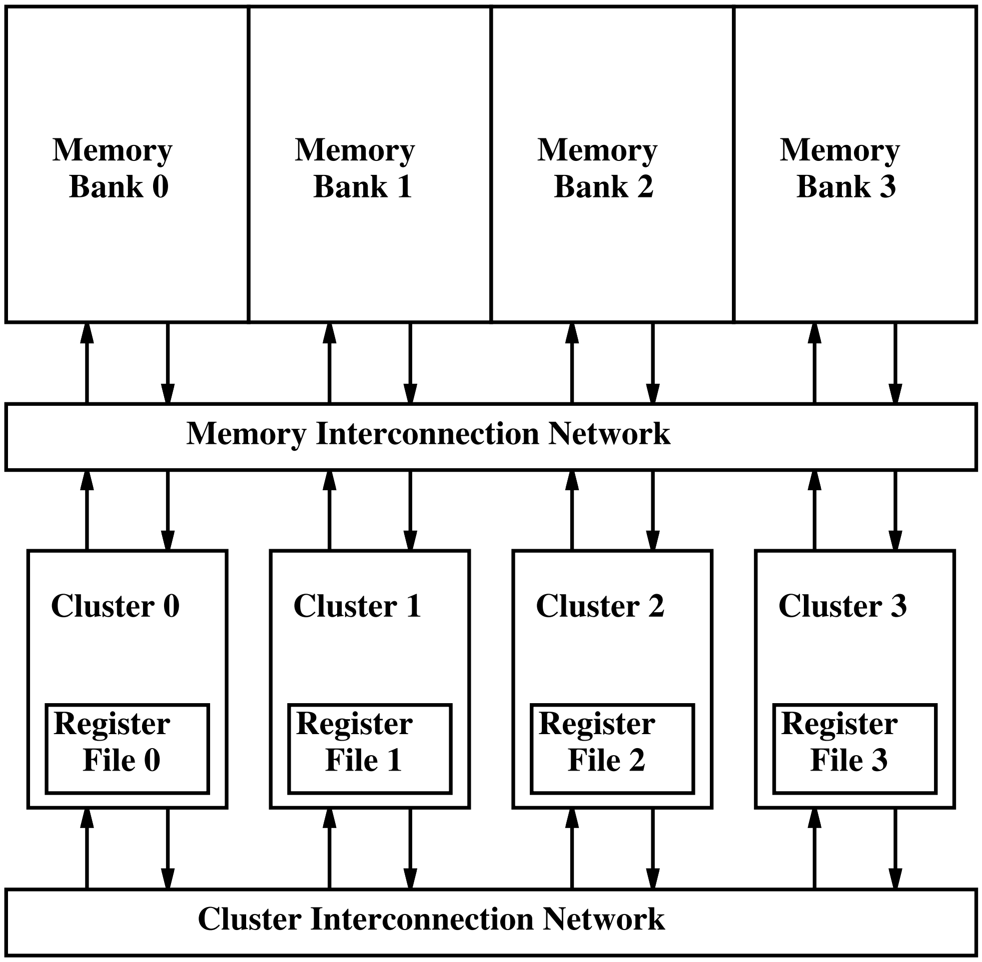

Figure 3.1: This sample machine consists of four clusters, each of which contains a register file and some number of function units. The clusters communicate with each other through the Cluster Interconnection Network and through memory.

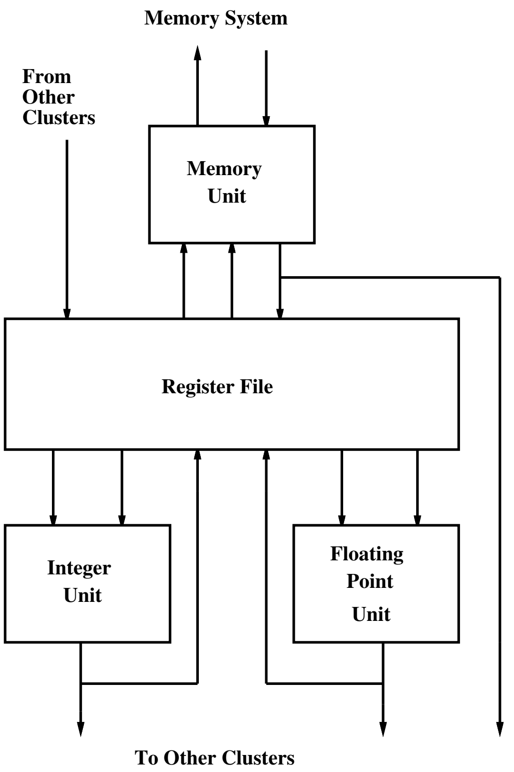

Figure 3.2: The prototypical cluster contains three function units: a memory unit, an integer arithmetic unit, and a floating point unit. Each share access to a common register file. The memory unit communicates with the memory system. Each unit can send its results to the local register file as well as to the register files of remote clusters.

---

file may be divided into integer and floating point partitions. A cluster can write to its own register file or to that of another cluster through the Cluster Interconnection Network. Clusters access memory banks through the Memory Interconnection Network.

The cluster shown in Figure 3.2 has an integer unit, a floating point unit, and a memory unit all sharing access to a common register file. The result of a function unit's computation is written to the local register file or is sent to another cluster's register file. The memory unit takes addresses and data from the register file, accesses the memory system, and returns the results to the appropriate cluster's register file. A memory request can deliver data to other clusters by using the Cluster Interconnection Network (CIN), or by telling the memory system to route the result directly through the Memory Interconnection Network (MIN).

3.1.2 Function Unit Components

A function unit may be an integer ALU, a floating point ALU, a branch calculation unit, or a memory access unit; a unit may be pipelined to arbitrary depth. As shown in Figure 3.3, a thread's instruction stream can be considered as a sparse matrix of *operations*. Each column in the matrix corresponds to a single function unit. To keep signal propagation delays as short as possible, control is distributed throughout the clusters. Each function unit contains an *operation cache* and an *operation buffer*. When summed over all function units the operation caches form the instruction cache. These components of an arithmetic function unit are shown in Figure 3.4.

The operation buffer holds a pending operation from each active thread. A cluster's register file is multi-ported to allow multiple read and write operations per cycle. Although execution of a thread's instructions does not take place in lock step, function units are loosely synchronized to prevent operations from issuing out of order. Chapter 7 discusses implementation details such as function unit pipelining in further detail.

3.1.3 Instruction Format

An instruction in a Processor Coupled node consists of a vector of operations, one for each function unit. Each operation specifies an opcode, one or more destination register fields, and operand fields. Operation encodings can be shared by different function units since each operation is generated by the compiler for a particular type of function unit. For example, `load` and `store` operations will not be generated for an integer arithmetic unit.

Function units transfer data by specifying destination registers in other clusters' register files. Often a value will be needed by a unit in a remote cluster. Encoding multiple destinations eliminates the operation required to move this data. Each destination register specifier is composed of two fields: one identifies the cluster register file and the other determines the particular register to be used. One destination specifier might be reserved for writes to the register file within the cluster.

Each thread's instruction stream may have holes due to unused function units. If the stream is encoded with explicit `nops` as placeholders, valuable space in the on-chip operation caches will be wasted. A potentially more efficient encoding has each operation specify the

C1	C2					C3	C4
C5	C6	C7		C8			C9
		C10				C11	C12

Figure 3.3: Three statically scheduled instructions from thread C of Figure 1.1 in Chapter 1. Each column of the instruction stream is a field for one of the eight function units and each row holds those operations which are allowed to execute simultaneously.

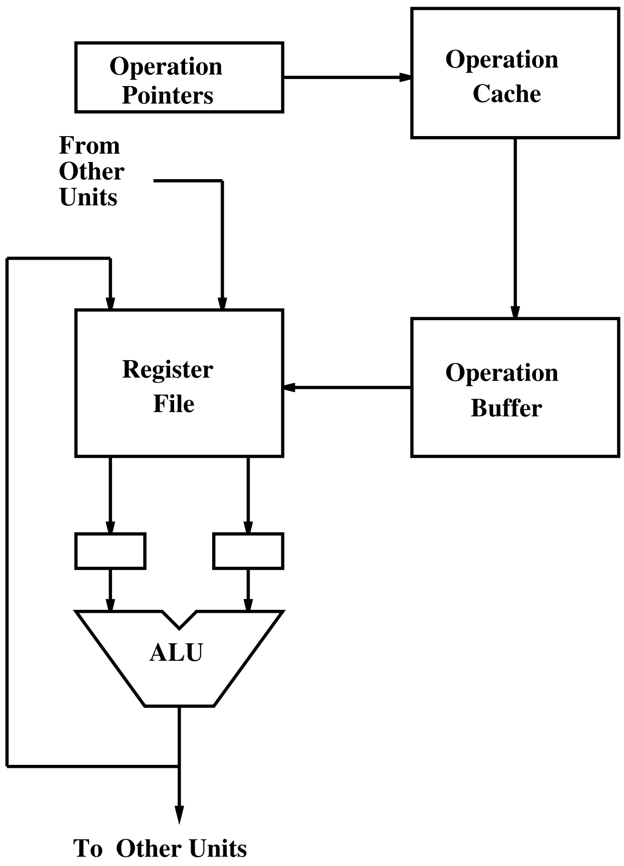

Figure 3.4: The components of a general function unit are shown in this sample arithmetic unit. Each thread has an operation pointer to access the operation cache. Fetched operations waiting for dependencies to be satisfied are held in the operation buffer. The register file is shown as a part of the function unit, but it is actually shared with other function units in the cluster.

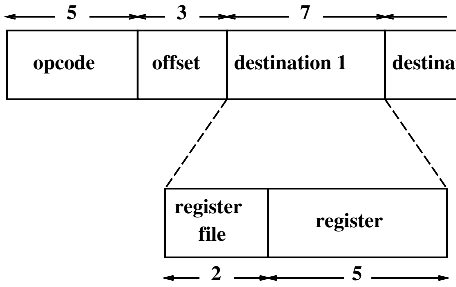

Figure 3.5: This operation devotes 5 bits to the opcode and 3 bits to the offset for the next executable operation. Destinations are specified using 2 bits to identify the cluster and 5 bits to identify the register. The two source registers are specified using 5 bits each.

---

number of following `nops` that the function unit will encounter before finding the next available operation. A short offset field in the operation encoding provides this capability. Alternatively, an instruction can be stored densely in memory and be expanded with explicit `nops` during operation cache refills.

Figure 3.5 shows how an operation can be encoded. This example displays a simple operation in which the operands are both in registers and the result can be sent to two different registers. Two bits select the destination cluster and 5 bits are used to specify each operand and destination register. The offset field of 3 bits allows the operation to indicate that as many as 7 implicit `nops` can be skipped before the next executable operation. The opcode needs only 5 bits. Thus, each operation is encoded in 32 bits and two operations can reside in a single 64 bit word. Standard techniques for encoding other instruction formats can be used as well.

3.2 Intra-thread Synchronization

Processor Coupling uses data presence bits in registers for low level synchronization within a thread. An operation will not be issued until all of its source registers are valid and all operations from the previous instruction have been issued. When an operation is issued, the valid bit for its destination register is cleared. The valid bit is set when the operation completes and writes data back to the register file. To move data between function units, an operation may specify destination registers in other clusters. Thus registers are used to indicate data dependencies between individual operations and to prevent operations from executing before their data requirements are satisfied. Because different function units may have different pipeline latencies, this discipline ensures in-order operation issue, but not necessarily in-order completion.

3.3 Multiple Threads

Hardware is provided to sequence and synchronize a small number of active threads. Each thread has its own instruction pointer and logical set of registers, but shares the function units and interconnection bandwidth. A thread's register set is distributed over all of the clusters that it uses. The combined register set in each cluster can be implemented as separate register files or as a collection of virtually mapped registers [ND91]. Communication between threads takes place through the memory on the node; synchronization between threads is on the presence or absence of data in a memory location.

Each function unit determines independently, through examination of dynamic data dependencies, the next operation to issue. That operation may be from any thread in the active set; threads may have different execution priorities. The function unit examines simultaneously the data requirements for each pending operation by inspecting the valid bits in the corresponding register files. The unit selects a ready operation, marks its destination registers invalid, and issues it to the execution pipeline stages.

A Processor Coupled system provides a set of thread management functions. If a thread in the active set idles, it may be swapped out for another thread waiting to execute. The process of spawning new threads and of terminating threads must occur with low latency as well. Thread management issues are beyond the scope of this thesis.

3.4 Memory System

The memory system is used for storage, synchronization, and communication between threads. Like the registers, each memory location has a valid bit. Different flavors of loads and stores are used to access memory locations. The capabilities of memory reference operations are similar to those in the Tera machine [ACC+90] and are summarized in Ta-

Reference	Precondition	Postcondition
load	unconditional	leave as is
	wait until full	leave full
	wait until full	set empty
store	unconditional	set full
	wait until full	leave full
	wait until empty	set full

Table 3.1: Loads and stores can complete if the location's valid bit satisfies the precondition. When a memory reference completes, it sets the valid bit to the specified postcondition.

ble 3.1. These mechanisms can be used to build producer-consumer relationships, atomic updates, semaphores, and other types of synchronization schemes.

On-chip memory is used as a cache and is interleaved into banks to allow concurrent access to multiple memory locations. Memory operations that must wait for synchronization are held in the memory system. When a subsequent reference changes a location's valid bit, waiting operations reactivate and complete. This split transaction protocol reduces memory traffic and allows memory units to issue other operations.

3.5 Summary

A multi-ALU node for Processor Coupling consists of several clusters of function units, an on-chip cache, and switches between different clusters and between clusters and memory banks. Each cluster contains a small number of function units, a register file, and the control logic and state, such as the operation cache and the operation buffer, needed to manage several active threads. The operation buffer is used to hold operations waiting to issue from different active threads.

The low interaction latency between function units allows different clusters to participate in the execution of the same thread and efficiently deposit data directly into each other's register files. Within a thread, the data dependencies of an operation are checked dynamically using the presence bits in the register files. Multiple threads are active simultaneously to increase utilization of function unit and memory resources. Different threads communicate through the memory of the node using fine grain synchronization bits on individual memory locations.

Chapter 4

Instruction Scheduling Compiler

The simulation environment for testing Processor Coupling consists of two parts: the Instruction Scheduling Compiler and the Processor Coupling Simulator. Figure 4.1 shows the flow through the compiler and simulator. A program and a configuration file are used by the compiler to generate assembly code and a simulator configuration file. After reading the assembly code file, the data file, and the configuration file, the simulator executes the compiled program and generates statistics about the runtime behavior. Optionally a trace file containing an exact history of the operations executed can be produced.

This chapter describes the Instruction Scheduling Compiler (ISC) that was developed to generate test programs for a Processor Coupled node. ISC serves several purposes. First, it provides a programming environment which allows benchmark programs to be developed quickly. Secondly, it allows the programmer to easily experiment with different partitioning and synchronization strategies. Finally, ISC provides the flexibility to generate code for machines with different hardware parameters such as number of function units, function unit latencies, and expected memory latencies. Chapter 5 describes the Processor Coupling Simulator.

Each benchmark program is written in a simple source language called PCTL (Processor Coupling Test Language) that has simplified C semantics with Lisp syntax. PCTL includes constructs for parallel execution of threads as well as explicit synchronization points on the granularity of a single variable. Along with the source program, ISC expects a machine configuration file which specifies the number and type of function units, each function

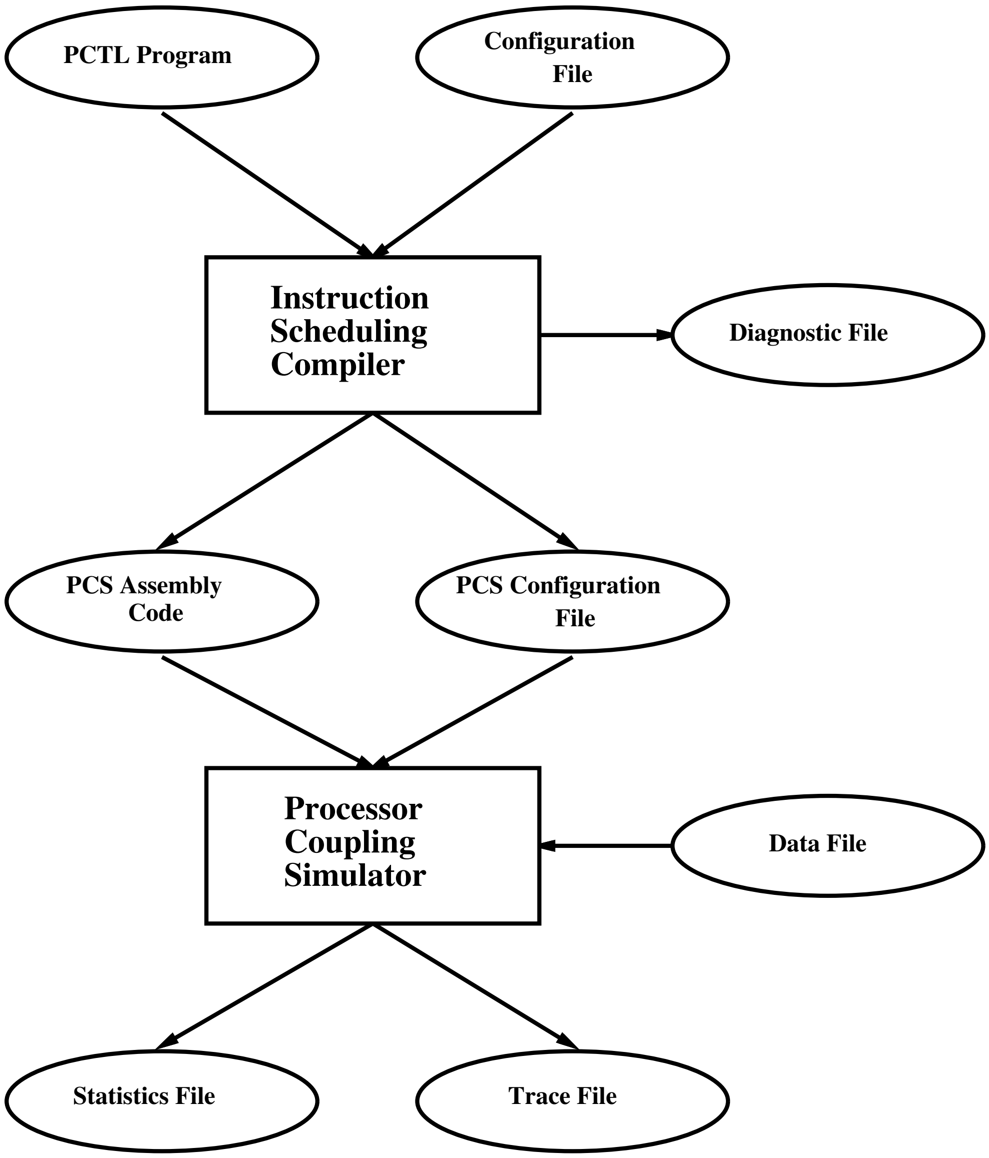

Figure 4.1: The flow through the simulation environment. PCTL programs are compiled into PCS assembly code by the Instruction Scheduling Compiler (ISC). PCS runs the program using the input data provided by the Data File. ISC determines the machine parameters from the Configuration File and produces a modified PCS Configuration File for the simulator.

unit's pipeline latency, and the grouping of function units into clusters. The compiler uses configuration information to statically schedule thread operations. ISC is implemented in Common Lisp [Ste90] and the source code can be found in [Kec92a].

Section 4.1 describes the source language PCTL, including its capabilities and its limits. The operation of the Instruction Scheduling Compiler is outlined in Section 4.2. Section 4.3 discusses the parameters of the programming environment that provide a variety of compilation strategies. Section 4.4 proposes enhancements that would improve operation and execution of ISC. Finally, Section 4.5 identifies the different simulator modes used in the experiments.

4.1 PCTL

The Processor Coupling Test Language (PCTL) is an imperative language much like C. A pseudo-Lisp syntax is used in PCTL to eliminate parsing issues and grammatical ambiguities, and to simplify the implementation of the compiler. PCTL provides a few data types: integers, floating point numbers, and one and two dimensional arrays of integers and floats. There is no support for complex data structures. The language structures described in Section 4.1.1 give explicit control of the hardware model to the programmer.

4.1.1 Language Constructs

Variable Declaration: Declarations are made using the `declare` statement. The declaration section of a program is of the form:

```
(declare ((var-name-1 var-type-1)
                   ⋮
          (var-name-N var-type-N)))
   E_body)
```

The variable declarations are valid for the entire scope of E_{body}. Variable types include `int`, `float`, and `array`. Arrays are declared using the following statements:

```
(array i var-type)
(array (i j) var-type)
```

The values i and j are integers indicating the dimension of the arrays while *var-type* determines whether the array elements are integers or floating point numbers. Procedures are specified in the declaration block using the form:

`(lambda (`*arg-list*`)` E_{body}`)`

where E_{body} can include local variable declarations and expressions.

Control: The following structures defined in PCTL determine the program's control flow.

`(begin` E_1..E_n`)` groups expressions E_1 through E_n to be executed sequentially. The execution order of operations is determined by data dependency analysis in the compiler.

`(begin-sync` E_1..E_n`)` is similar to `begin` except that the enclosed operations are not executed until all operations from previous blocks have been issued. The use of `begin-sync` in synchronization is discussed in Section 4.1.3.

`(parex` E_1..E_n`)` groups expressions E_1 through E_n to be executed in parallel. A new thread is created for each element in the `parex` body.

`(if (`E_{test}`)` E_{cons} E_{alt}`)` executes E_{cons} if $E_{test} \neq 0$; otherwise E_{alt} is evaluated.

`(for (`E_{init} E_{test} E_{inc}`)` E_{body}`)` is the PCTL version of a sequential `for` loop. E_{init}, E_{test}, and E_{inc} are the initialization, test, and increment expressions, respectively. E_{body} is executed once for each iteration of the loop.

`(forall (`E_{init} E_{test} E_{inc}`)` E_{body}`)` is similar to the `for` loop except that all iterations are executed simultaneously as different threads. The number of iterations must be known at compile time.

`(forall-iterate (`E_{init} E_{test} E_{inc}`)` E_{body}`)` is another parallel `for` loop, but the number of iterations can be determined at runtime. Unlike `forall` a new asynchronously running thread is created for E_{body} on each iteration of the loop.

`(while (`E_{test}`)` E_{body}`)` executes E_{body} sequentially for each iteration of the loop. The loop terminates when E_{test} evaluates to zero.

`(fork` E_{body}`)` creates a new thread to asynchronously execute E_{body}. The thread issuing the `fork` instruction continues to execute as well.

`(fork-if (`E_{test}`)` E_{body}`)` evaluates E_{test} and creates a new asynchronous thread for E_{body} if $E_{test} \neq 0$. This type of conditional execution reduces the number of branch operations which limit instruction-level parallelism.

`(call` *proc arg-list*`)` emulates a procedure call by a macro-expansion that is inlined in the code. Recursion is not permitted.

Synchronization: PCTL allows synchronization on the granularity of a single variable. The keywords `uncond`, `consume`, `produce`, and `leave` can be used when accessing a variable to alter its presence bit in memory. This will be further discussed in Section 4.1.3.

Assignment: Variables can be assigned values using the `:=` operator. Executing

`(:=` *var* E_{body}`)`

assigns the value of E_{body} to *var*. One dimensional arrays are referenced using

`(aref` *a-name index*`)`

where *a-name* is the array and *index* is an integer variable or constant. Two dimensional arrays are accessed as

`(aref` *a-name* `(`*row column*`))`

Arithmetic Operators: PCTL supports a standard set of arithmetic operators including: `+`, `-`, `*`, `/`, and `neg` (arithmetic negation). Integer or floating point operators are chosen automatically and type-casting is transparent to the programmer. The shift operators, `<<` (arithmetic shift left) and `>>` (arithmetic shift right), perform only on integers.

Relational and Logical Operators: Relational operators produce integral results from comparisons on integer or floating point operands. As in C, false is represented as the integer 0, and true is anything other than zero. The relational operators that PCTL supports are: `<` (less than), `<=` (less than or equal), `>` (greater than), `>=` (greater than or equal), `==` (equal), and `!=` (not equal). The logical operators—`and`, `or`, and `not`—operate only on integers.

```
(declare
 ((old-array (array 18 float))
  (new-array (array 18 float))
  (size int) (i int) (num-iter int)
  (iter int))
 (begin
  (:= size 16)
  (:= num-iter 4)
  (for ((:= iter 0) (< iter num-iter) (:= iter (+ iter 1)))
       (begin
        (for ((:= i 1) (<= i size) (:= i (+ i 1)))
             (begin
              (:= (aref new-array i)
                  (/ (+ (aref old-array (- i 1))
                        (aref old-array (+ i 1)))
                     2))))
        (for ((:= i 1) (<= i size) (:= i (+ i 1)))
             (begin
              (:= (aref old-array i)
                  (aref new-array i))))))))
```

Figure 4.2: This sample PCTL program performs Jacobi relaxation on a one dimensional array of 16 elements. The algorithm uses evaluate and update phases. The program is sequential and does not require synchronization.

Figure 4.2 shows a PCTL implementation of one dimensional Jacobi relaxation. The arrays `old-array` and `new-array` contain 18 elements: 16 for the vector and 2 for the end-point boundary conditions. The outer loop controls the number of iterations of the relaxation while the two inner loops execute the evaluate and update phases. This version is entirely sequential and requires neither partitioning nor synchronization.

4.1.2 Parallel Partitioning

The PCTL constructs to explicitly declare multithreaded parallelism were introduced in the previous section and are described here in more detail. Parallel tasks that can be split into threads and executed concurrently are created using the `fork` command. A new thread of control that can be run concurrently with the other threads already executing is created for each forked expression.

Other more complex constructs are built using `fork`. The `parex` operation creates a

Reference	Directive	Precondition	Postcondition
load	`uncond`	unconditional	leave as is
	`leave`	wait until full	leave full
	`consume`	wait until full	set empty
store	`uncond`	unconditional	set full
	`leave`	wait until full	leave full
	`produce`	wait until empty	set full

Table 4.1: These directives can be used when accessing a variable to alter a memory presence bit. The access will stall until the specified precondition is met, and when complete will set the presence bit to the postcondition.

thread for each of its subsidiary expressions. A `forall` loop can be used if the number of iterations is known at compile time. It has the same semantics as a `for` loop, but it creates one thread for each iteration of the loop, and executes all iterations concurrently. The `forall-iterate` command can be used as a parallel loop when the number of iterations is not known until runtime. The `forall-iterate` structure differs from a `fork` within a `for` loop because it attempts to perform load balancing. The body of the loop is compiled several times, each time with a different function unit access pattern. At runtime each thread selects a version of the compiled code so that threads spread their work over all of the function units.

4.1.3 Synchronization Cues

PCTL provides the capability to create explicit synchronization points. The programmer can specify synchronization on a single variable. Synchronization is performed using specialized read and write operations to alter the memory presence bits. Table 4.1 is an augmentation of Table 3.1 in Chapter 3 showing the corresponding directives for preconditions and postconditions set by memory operations. If no directive is specified, the access defaults to the `uncond` mode.

When a thread attempts to access a variable with an empty presence bit, it will stall until the value is ready. Thus, the presence tags can be used to implement atomic updates to variables, semaphores, and other more complex synchronization schemes. A simple producer-consumer relationship between two threads uses the `produce` and `consume` direc-

```
(:= sync-var (consume sync-counter))
(if (< sync-var num-threads)
   (begin
     (:= (produce sync-counter) (+ sync-var 1))
     (:= sync-var (leave sync-start)))
  (:= (produce sync-start) sync-var))
```

Figure 4.3: An example of the barrier synchronization code to be executed by each thread requiring synchronization. The variable `sync-counter` is the counter for number of threads to synchronize. If `sync-counter` is less than `num-threads` (the number of synchronizing threads), then the thread increments `sync-counter` and tries to read `sync-start`. If all the threads have in fact reached the barrier, the thread writes to `sync-start` to wake up the stalled threads. The `produce` and `consume` tags ensure atomic updates to `sync-counter`. The `leave` and `produce` tags make `sync-start` a broadcast variable.

---

tives when referencing a shared variable. A barrier synchronization point can be created using a shared counter. When a thread reaches the barrier, it checks the value of the counter. If the counter has a value less than the number of synchronizing threads it is incremented and the thread suspends by attempting to reference an invalid synchronization variable. When the last thread reaches the barrier it finds that the counter has the proper value and writes to the shared synchronization variable to meet the precondition of the read accesses of all of the waiting threads, thus reactivating them. Figure 4.3 shows the barrier synchronization code that each thread must execute. In order to scale well, barrier synchronization should be implemented as a combining tree [YTL87].

The compiler assumes that all operations within a thread can be scheduled according to data dependencies. In order to ensure a predictable sequential ordering of synchronization operations, the programmer must use the `begin-sync` construct. The `begin-sync` statement guarantees that no subsequent operations will be executed until all previous operations have been issued. This can be used to indicate critical sections as well as synchronization dependencies between memory accesses.

Figure 4.4 shows a parallel implementation of the sequential one-dimensional Jacobi relaxation program of Figure 4.2. Like the sequential version, this program has evaluate and update phases. Work is partitioned into four threads, each of which is responsible for evaluating and updating a four element subarray. The threads are created using `forall` and

```
(declare
 ((old-array (array 18 float))
  (new-array (array 18 float))
  (size int) (i int) (num-iter int) (iter int) (temp1 int)
  (in-sync int) (sync-counter int) (sync-start int)
  (barrier (lambda (sync-var)
             (begin (:= sync-var (consume sync-counter))
                    (if (< sync-var 4)
                        (begin
                         (:= (produce sync-counter) (+ sync-var 1))
                         (:= (uncond sync-var) (leave sync-start)))
                      (:= (produce sync-start) sync-var)))))
  (evaluate (lambda (index)
              (declare ((in-sync int) (base int) (j int))
                       (begin (:= base (+ (* index 4) 1))
                              (for ((:= j base) (< j (+ base 4)) (:= j (+ j 1)))
                                   (begin
                                    (:= (aref new-array j)
                                        (/ (+ (aref old-array (- j 1))
                                              (aref old-array (+ j 1))) 2))))
                              (begin-sync (call barrier in-sync))))))
  (update (lambda (index)
            (declare ((in-sync int) (base int) (j int))
                     (begin
                      (:= base (+ (* index 4) 1))
                      (for ((:= j base) (<= j (+ base 4)) (:= j (+ j 1)))
                           (begin (:= (aref old-array j)
                                      (aref new-array j))))
                      (begin-sync (call barrier in-sync)))))))
 (begin (:= size 16)
        (:= num-iter 4)
        (:= (uncond sync-start) 0)
        (for ((:= iter 0) (< iter num-iter) (:= iter (+ iter 1)))
             (begin (:= (uncond sync-counter) 0)
                    (:= temp1 (consume sync-start))
                    (:= temp1 (+ temp1 1))
                    (forall ((:= i 0) (< i 4) (:= i (+ i 1)))
                            (begin (call evaluate i)))
                    (begin-sync (call barrier in-sync))
                    (:= (uncond sync-counter) 0)
                    (:= temp1 (consume sync-start))
                    (:= temp1 (+ temp1 1))
                    (forall ((:= i 0) (< i 4) (:= i (+ i 1)))
                            (begin (call update i)))
                    (begin-sync (call barrier in-sync)))))))
```

Figure 4.4: A parallel version of the sequential Jacobi relaxation code shown in Figure 4.2.

barrier synchronization is used after each evaluate and update phase. Three procedures—`barrier`, `evaluate`, and `update`—are declared globally, and each can have its own local variables.

4.2 Compilation

The Instruction Scheduling Compiler (ISC) discovers instruction-level parallelism in a PCTL program and schedules its operations to make maximum use of multiple function units. The input to ISC is a PCTL program and a configuration file. During the parse and partition phase, the compiler uses the cues specified in the program to partition it into multiple threads and generate intermediate code. Each thread's code then proceeds independently through the subsequent optimization, data dependency analysis, scheduling, and code generation phases. Upon completion of compilation, ISC produces assembly code for the Processor Coupling Simulator (PCS), a PCS configuration file, and a diagnostic file displaying the compiler flags that were selected. The PCS configuration file contains additional information indicating the number of registers and the amount memory required. The compilation flow of ISC is shown in Figure 4.5.

Figure 4.6 shows a simple matrix-vector product program, computing $A \times x = b$, that will be used to demonstrate the compilation steps discussed in subsequent sections. The matrix-vector product code is partitioned into one master thread and four subsidiary threads. The master thread forks the subsidiary threads at runtime; each subsidiary thread is responsible for computing the dot-product of one row of `a-mat` with `x-vec` to produce one element of `b-vec`. No synchronization is required since the computation of each element of `b-vec` is independent.

4.2.1 Parsing

The parser first transforms the PCTL source program by desugaring complex expressions into the core kernel language of the compiler and breaking complicated arithmetic expressions into balanced trees. Then it partitions the result according to the program specified directives and creates an annotated parse tree for each thread. A global symbol table as well as a symbol table for each thread is instantiated with declared variables. Global vari-

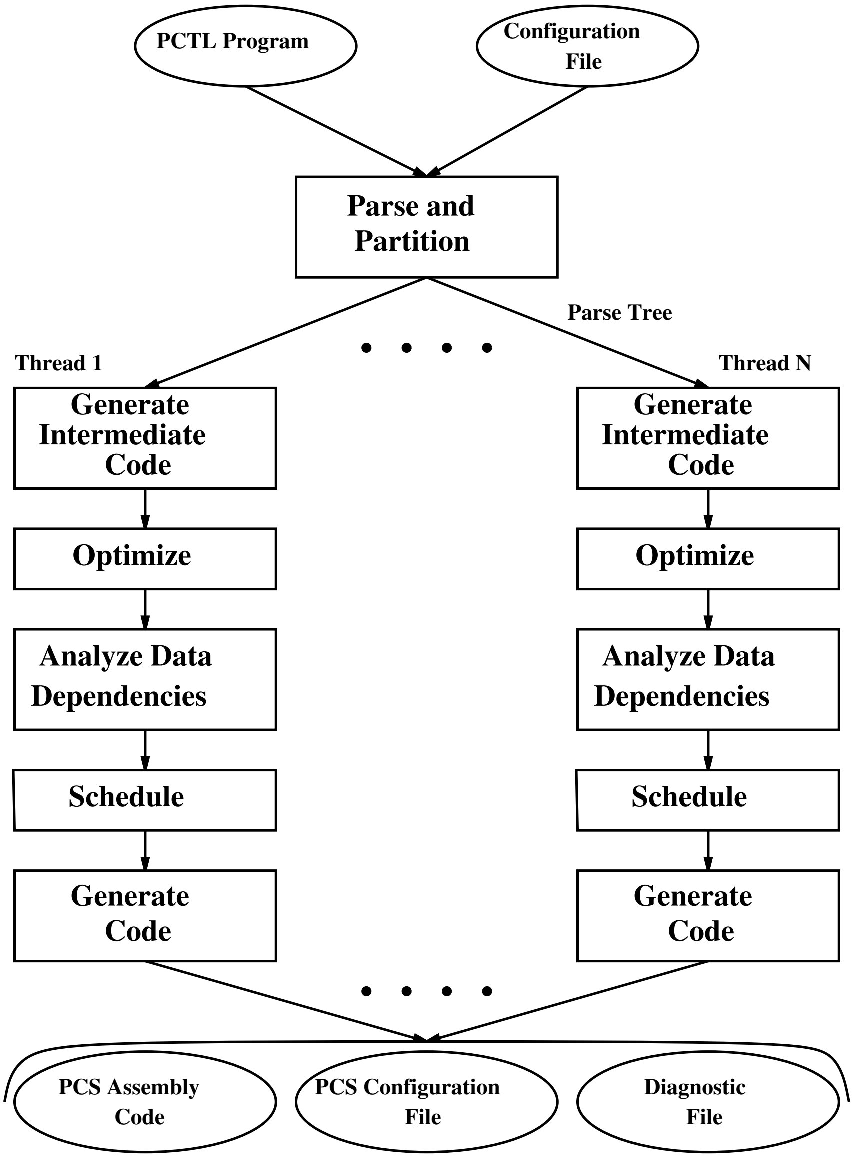

Figure 4.5: Compiler flow of control. The program and configuration files are used by the parser and partitioner which generate intermediate code. Each thread has its own intermediate code which proceeds through the optimization, data dependency analysis, scheduling, and code generation stages. The output is a new configuration file, the assembly code, and a diagnostic file.

```
(declare
 ((a-mat (array (4 4) float))
  (x-vec (array 4 float))
  (b-vec (array 4 float))
  (i int)
  (dot-product
   (lambda (a-mat x-vec b-vec index)
     (declare
      ((j int) (temp float))
      (begin
       (:= temp 0)
       (for ((:= j 0) (< j 4) (:= j (+ j  1)))
            (begin (:= temp
                       (+ temp
                          (* (aref a-mat (index j))
                             (aref x-vec j))))))
       (:= (aref b-vec index) temp))))))
 (begin
  (forall ((:= i 0) (< i 4) (:= i (+ i 1)))
          (begin
           (call dot-product a-mat x-vec b-vec i))))))
```

Figure 4.6: A parallel implementation of a 4×4 matrix-vector product, computing $A \times x = b$. The program is partitioned into four threads, each of which computes a dot-product of a row of `a-mat` and `x-vec`.

ables are assigned absolute addresses while variables local to a thread are given offsets to be combined with a context pointer at runtime. Procedures are inlined and new names for local variables are created to avoid false aliasing.

Rudimentary optimizations, including constant propagation and removal of unreachable blocks are performed. Before completing, the parser makes a data structure for each thread to hold its parse tree, symbol table, and slots for intermediate code generated in later stages. Each thread proceeds independently through the subsequent phases of the compiler.

4.2.2 Generating Intermediate Code

The intermediate code generator transforms a parse tree into a sequence of basic blocks. Basic block breakpoints are located by analyzing the control flow structures such as loops and conditionals. A basic block has one entry point, one exit point, and up to two pointers to blocks that may follow it in the control flow graph. Each basic block data structure

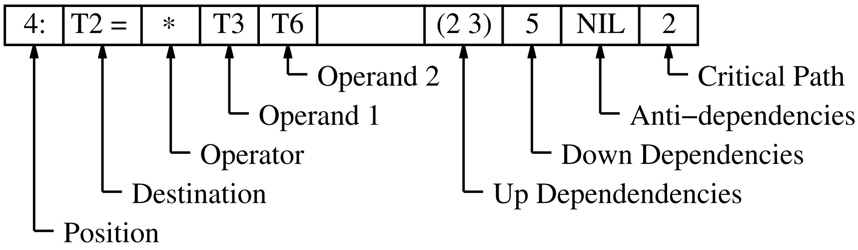

Figure 4.7: Three address code statment computing $T3 \times T6 = T2$.

contains:

- An identification number.
- A list of the variables it uses.
- A list of the variables it defines.
- A data structure for intermediate code.
- Pointers to subsequent blocks in the flow control graph.

The simple three address code used as the intermediate form in ISC is generated directly from the operations specified in the parse tree. Figure 4.7 shows a sample of a three address code statement. The position field indicates the location of the statement in the code vector of its basic block. The operation is specified by the destination, operator, and operand fields. Up dependency and anti-dependency fields indicate the previous operations on which the statement depends. The down dependency field shows the subsequent operations that depend upon the result. Finally, the critical path field indicates how many operations lie between the statement and the bottom of the basic block. The intermediate code generator creates the code vector consisting of these three address statements, but the dependency and critical path fields are determined in the data dependency analysis phase. The intermediate code for matrix-vector product is shown later in Section 4.2.4.

4.2.3 Optimization

The optimizations performed on the three address code include constant propagation and common subexpression elimination [ASU88]. Constant propagation beyond that performed in the parser might be available due to binding of variables to constants during the mapping from parse trees to three address code. Common subexpressions are recognized by comparing the operator and operands of different statements in the same basic block. Optimizations between basic blocks are not performed.

In a multiple arithmetic unit processor, eliminating common subexpression may require more data to be transferred between function units. For example the result from an address calculation might be needed by several memory operations on different clusters. If operands are required in several function units, recomputing a value may be more efficient that transferring it. The analysis of the efficiency of common subexpression elimination is not examined in this thesis and not included in ISC.

4.2.4 Data Dependency Analysis

In order to schedule independent operations into long instruction words, the dependencies between operations within a basic block must be determined. Data dependency analysis is performed on a block by block basis by traversing a block's code vector in reverse order. Each statement's operands are compared with the destinations of previous operations to determine the direct dependencies. Anti-dependencies are discovered by comparing a statement's destination with the operands and destinations of previous operations.

The data dependency graph is traversed to find the critical path. Operations with no dependencies are labeled with a zero. Statements that generate values used by zero-labeled operations are labeled one. This continues until all statements have been labeled. A statement's critical path label indicates the maximum number of operations that lie between it and the end of the basic block along the data dependency graph. These identifiers are used by the scheduler for efficient operation scheduling along the critical path of the code.

Live variables are kept in registers between basic blocks, but operations are not moved beyond basic block boundaries. Dependencies between blocks are maintained only within a directed acyclic subgraph (DAG) [AHU83] of the thread's flow control graph. Loops cause all live variables to be saved to memory.

Figure 4.8 shows the intermediate three address code for one of the subsidiary threads in the matrix-vector product program of Figure 4.6 after data dependency analysis. Basic block 0 initializes the values before entering the loop body of basic block 1. The critical path in basic block 1 starts with the address calculation for `a-mat` and terminates with the branch statement. The element computed in the thread is stored in `b-vec` in basic block 2.

4.2.5 Scheduling

The scheduler maps the intermediate code from the data dependency analysis phase to the physical machine. As in previous phases, work is performed on a block by block basis, and the scheduler does not move operations across basic block boundaries. The instruction schedule is a matrix whose width is equal to the number of function units, and whose length is determined by the number of dependent operations in the basic block.

The scheduler fills slots in the matrix by selecting and placing operations from the intermediate code. Operations are selected according to their location in the critical path. This method provides the most desirable locations in the schedule to critical operations and guarantees that when an operation is selected its predecessors have already been scheduled.

Upon entry to a basic block, the scheduler first generates and places `load` operations to load the variables not already in registers. Then for each operation in the intermediate code, the scheduler performs the following analysis to determine where to place it in the matrix. First the scheduler locates the operands and uses the function unit latencies specified in the configuration file to determine the first row of the matrix in which both operands will be available. Then the scheduler examines that row to find all unscheduled function units that can execute the selected operation. If it finds several free function units, it tries to select the one that minimizes data movement. For example if both operands are located in a particular cluster and that cluster has a free function unit, then the scheduler will place the operation there. If the operands reside in different clusters or in a cluster with already filled function units, then the operands can be transferred to a cluster with a free function unit.

Data can be moved between function units in two ways. In the first method, a value

```
basic block 0
 up = NIL
 uses: NIL
 0 : |i0| = 0                  (NIL NIL)   NIL       NIL       0
 1 : |temp5| = 0               (NIL NIL)   NIL       NIL       0
 2 : |j4| = 0                  (NIL NIL)   NIL       NIL       0
 3 : branch FALSE  1           (NIL NIL)   NIL       NIL       0
 definitions: (|i0| |j4| |temp5|)
 next = bb1   branch = bb2

basic block 1
 up = (bb1 bb0)
 uses: (|i0| |j4| |temp5| A-MAT X-VEC)
 0 : T5  = *   4   |i0|        (NIL NIL)   (1)       NIL       5
 1 : T4  = +   T5  |j4|        (0 NIL)     (2)       NIL       4
 2 : T3  = aref A-MAT T4       (NIL 1)     (4)       NIL       3
 3 : T6  = aref X-VEC |j4|     (NIL NIL)   (4)       NIL       3
 4 : T2  = *   T3  T6          (2 3)       (5)       NIL       2
 5 : T1  = +   |temp5| T2      (NIL 4)     (6)       NIL       1
 6 : |temp5| = T1              (5 NIL)     NIL       (5)       0
 7 : T7  = +   |j4| 1          (NIL NIL)   (8)       NIL       3
 8 : |j4| = T7                 (7 NIL)     (9)       (1 3 7)   2
 9 : T0  = <   |j4| 4          (8 NIL)     (10)      NIL       1
 10: branch TRUE  T0           (9 NIL)     NIL       NIL       0
 definitions: (|j4| |temp5|)
 next = bb2   branch = bb1

basic block 2
 up = (bb1 bb0)
 uses: (|i0| |temp5|)
 0 : aref B-VEC |i0| = |temp5| (NIL NIL)   NIL       NIL       0
 definitions: (B-VEC)
 next = bb3   branch = NIL

basic block 3
 up = (2)
 uses: NIL
 0 : exit                      (NIL NIL)   NIL       NIL       0
 definitions: NIL
 next = NIL  branch = NIL
```

Figure 4.8: The intermediate code for one of the subsidiary threads of the matrix-vector program after data dependency analysis.

generated in one function unit is sent directly to the register file of another cluster. In the second, an explicit move operation transfers data from one cluster to another. Transferring the data automatically is the fastest since data can be sent to other clusters without executing any additional operations.

The scheduler tries to minimize the movement of data between units while still using as many of the function units as possible. It uses its knowledge of the cost of moving data to decide where to place an operation. If an explicit move is required to use a function unit on the selected cycle, the following cycle is examined for a better slot. If no such slot is available, then move code will be generated and the operation will be placed in the first available location.

If multiple function units are available, the scheduler will select one according to a fixed preference assigned to the thread. Each thread has a list of all function units sorted by its compiler assigned preference. This priority is used to achieve spatial locality in the access patterns of the function units. Thus the operations from a thread are not randomly scattered over the function units, but instead are kept locally on those function units with the highest priority. The compiler prioritizes the function units differently for each thread in order to prevent all threads from having the same function unit access pattern. This serves as a simple form of load balancing.

When an appropriate location for an operation has been found, the code generator is called to generate the assembly code for the selected operation. The scheduler does not backtrack so that once an operation has been placed, it will not be moved to another location. Contention for register ports and write back buses is not considered by the compiler; it assumes that all the resources will be available when necessary.

4.2.6 Code Generation

Code generation takes place concurrently with scheduling. The scheduler calls the code generator when it has selected a code statement and its schedule location. The code generator chooses the correct opcode and determines the operation format, the number of registered or immediate operands, the type of synchronization required, and the registers to be used. After the code from all threads has been scheduled, a final pass of the code generator counts

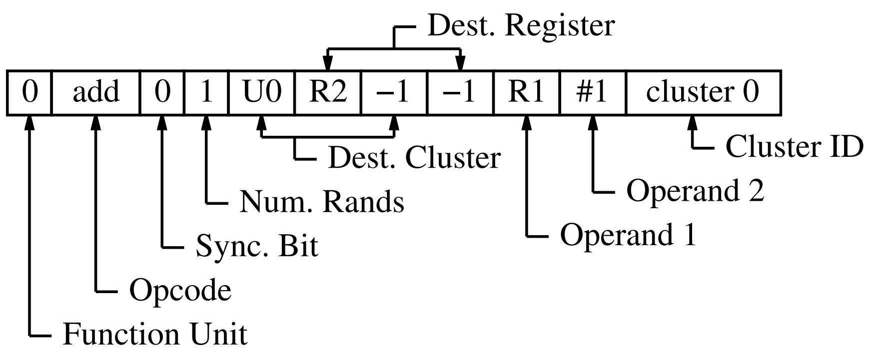

Figure 4.9: The fields of a PCS assembly operation.

instructions in order to determine the offsets for `branch` and `fork` operations.

The code generator does not perform register allocation but instead assumes that an infinite number of registers are available. Simulation results show that the realistic machine configurations all have a peak of fewer than 60 live registers per cluster for each of the selected benchmarks. Averaging over the benchmarks, a cluster uses a peak of 27 registers. However, ideal mode simulations in which loops are unrolled extensively by hand require as many as 490 registers.

4.2.7 Output

The compiler produces Processor Coupling Simulator (PCS) assembly code, a diagnostic file, and a modified configuration file with information concerning register and memory requirements. An instruction for PCS is a vector of operations, one for each function unit in the simulated machine. If the compiler cannot schedule an operation on a particular function unit, an explicit `nop` is assigned there. The PCS assembly program is represented by a dense matrix of operations.

Figure 4.9 shows a sample assembly operation, which adds *1* to the contents of *R1* and stores the value in *R2* of cluster 0. The function unit specifier determines the function unit where the operation is executed. The function unit numbering scheme is common to the compiler and the simulator. The synchronization bit is used to specify pre and post-conditions for load and store operations, and the *num-rands* field tells the simulator which operands are in registers. In this encoding, up to two destination registers may be specified. The *-1* in the second destination field indicates that only one target register is used. The cluster ID field shows the cluster where the operation will execute and is for user readability only. Load and store operations add the contents of the operand 1 and operand 2 fields to determine the memory address used. Store operations specify the data to be written using the first destination register specifier. Branch operations test the value in the operand 1 slot and branch to the offset specified in the first destination register.

Figure 4.10 shows the user readable assembly code for the master thread and one subsidiary thread of the matrix-vector program, generated directly from the intermediate code of Figure 4.8. Different instructions within a basic block are separated by spaces and the explicit `nops` are not displayed. Each operation's encoding is similar to the example in Figure 4.9. The master thread (Thread 0) forks the four subsidiary threads in basic block 0 and then exits in basic block 1. Thread 1 begins by storing the loop initialization variables at offsets to the context pointer *R0*. Basic block 1 executes the loop body once and branches back to the top of the block if more iterations are necessary. Basic block 2 stores the result of the dot product into the proper array slot of `b-vec`. The arrays `a-mat`, `x-vec`, and `b-vec` are at absolute addresses of `#1000`, `#1016`, and `#1020` respectively.

4.3 Flexibility

The Instruction Scheduling Compiler is capable of generating code for several machine configurations. The programmer can specify how compilation is performed using the configuration file and compiler switches. In addition, the programmer can explicitly partition the program and create synchronization structures using the directives described in Section 4.1.

4.3.1 Configuration Information

The configuration file provides information about the target machine to the compiler. The scheduler uses this information to map the intermediate code to the hardware model, which the code generator needs to format the assembly code appropriately. The configuration file is broken up into three sections. The header specifies the output file names and indicates the number of function units, the number of clusters, the number of register files, and the

```
Thread 0:
bb0:
12 forka 0 0 -1  thread4 -1 -1   0      0             cluster 4

12 forka 0 0 -1  thread3 -1 -1   0      0             cluster 4

12 forka 0 0 -1  thread2 -1 -1   0      0             cluster 4

12 forka 0 0 -1  thread1 -1 -1   0      0             cluster 4

bb1:
12 exit  0 0 -1   -1   -1   -1   0      0             cluster 4

Thread 1:
bb0:
1  stor  0 1 -1   #0   -1   -1   R0     #1            cluster 0
4  stor  0 1 -1   #0   -1   -1   R0     #0            cluster 1
10 stor  0 1 -1   #0   -1   -1   R0     #2            cluster 3
12 bf    0 0 -1   bb2  -1   -1   #1     0             cluster 4

bb1:
1  load  0 1 U1   R3   U0   R1   R0     #1            cluster 0
4  load  0 1 U1   R1   -1   -1   R0     #0            cluster 1
10 load  0 1 U3   R1   -1   -1   R0     #2            cluster 3

0  add   0 1 U0   R2   -1   -1   R1     #1            cluster 0
1  load  0 1 U1   R6   -1   -1   R1     #1016         cluster 0
3  mul   0 1 U1   R2   -1   -1   R1     #4            cluster 1

0  lt    0 1 U4   R1   -1   -1   R2     #4            cluster 0
1  stor  0 4 -1   R2   -1   -1   R0     #1            cluster 0
3  add   0 2 U1   R4   -1   -1   R2     R3            cluster 1

4  load  0 1 U1   R5   -1   -1   R4     #1000         cluster 1

5  fmul  0 2 U3   R2   -1   -1   R5     R6            cluster 1

11 fadd  0 2 U3   R3   -1   -1   R1     R2            cluster 3

10 stor  0 4 -1   R3   -1   -1   R0     #2            cluster 3
12 bt    0 1 -1   bb1  -1   -1   R1     0             cluster 4

bb2:
1  load  0 1 U0   R1   -1   -1   R0     #2            cluster 0
4  load  0 1 U0   R2   -1   -1   R0     #0            cluster 1

1  stor  0 4 -1   R1   -1   -1   R2     #1020         cluster 0

bb3:
12 exit  0 0 -1   -1   -1   -1   0      0             cluster 4
```

Figure 4.10: Assembly code for matrix-vector product.

number of destination register slots in an instruction.

The middle section enumerates the components of each cluster, including register files, and function unit types. A cluster descriptor starts with the keyword `cluster` and is followed by register file `regfile` keyword and function unit descriptors. The allowed function units are integer units (`int`), floating point units (`float`), branch units (`branch`), memory units (`mem`), and move units (`mov`). Each function unit entry includes an integer indicating the number of execution pipeline stages in the unit.

The final section has entries indicating the starting memory address that can be used by the compiler and the number of memory banks. The on-chip memory of a Processor Coupled node is interleaved into banks so that multiple memory operations can access memory concurrently. However, the current simulation environment ignores bank conflicts and assumes that an operation will not be blocked from accessing any memory bank. The sample configuration file of Figure 4.11 has five clusters. Four of the clusters are arithmetic and contain an integer unit, a memory unit, and a floating point unit. The fifth is a branch cluster and contains only a branch unit. Each cluster has one register file.

4.3.2 Compiler Switches

The user controlled compiler switches allow the programmer to specify how data is transferred between clusters and how the scheduler places operations on function units. The `*use-moves*` switch determines whether or not the compiler will use `move` operations to transfer data from one cluster's register file to that of another. If `*use-moves*` is set, each cluster must have a dedicated *move* unit specified in the configuration file. When data must be transferred between different clusters' register files, a `move` operation is generated and scheduled on the appropriate move unit. If `*use-moves*` is not selected then data must be transferred by an integer or floating point unit. This option provides the capability to evaluate the usefulness of explicit move units versus incorporating that function into the arithmetic units.

The `*mode*` switch determines where a thread's operations can be scheduled. If `*mode*` is set to `single`, all of the operations for a thread will be scheduled on the same cluster. Thus a program without multithreading would be restricted to using only one cluster. If

```
mat-vec         /* Prefix to be used for the output files */
13              /* Number of function units */
5               /* Number of clusters */
5               /* Number of register files */
2               /* Number of destination registers to be
                   specified in an instruction */

cluster         /* Cluster 1 */
regfile
int 1
mem 1
float 1

cluster         /* Cluster 2 */
regfile
int 1
mem 1
float 1

cluster         /* Cluster 3 */
regfile
int 1
mem 1
float 1

cluster         /* Cluster 4 */
regfile
int 1
mem 1
float 1

cluster         /* Cluster 5 */
regfile
branch 1

1000            /* Starting memory address */
2               /* Number of memory banks */
1               /* Cache hit latency */
0               /* Cache miss rate */
1               /* Cache minimum miss penalty */
1               /* Cache maximum miss penalty */
```

Figure 4.11: A sample compiler configuration file. The machine specified by this configuration file has five clusters, each with its own register file. Two destination registers may be targeted in each instruction. The arithmetic clusters have one integer unit, one floating point unit, and one memory unit while the last cluster has only a branch unit. There are no pipeline latencies as each unit operates in a single cycle. The cache parameters are passed on to the simulator.

`*mode*` is set to `unrestricted`, a thread's operations may be spread around the clusters by the scheduler, allowing the thread to use as many or as few of the function units as it needs. The `*mode*` switch allows the user to isolate threads to compare a multiprocessor model to the Processor Coupling model which uses the low interaction latency between clusters.

4.4 Future Work

ISC is a good first cut at an instruction scheduling compiler and is useful in testing the viability of Processor Coupling. However, further additions are required to make it a proper programming system. Enhancements to improve performance and expressibility include:

- Better Synchronization Analysis: Little is done in ISC to optimize synchronization points to allow for maximum instruction-level parallelism.

- Abstract Data Types: A new revision of the source language would need a method of composing simple data types into more complex data structures. Building arbitrary arrays of arrays would be sufficient.

- Efficient Scheduling: The current scheduler operates slowly and is good at analyzing operations only within a single thread. Efficient scheduling heuristics and analysis of operations between threads to reduce resource contention are open research topics.

- Dependency Analysis: The dependency analysis performed in ISC assumes that all references to a given array are related. The capability to disambiguate or explicitly synchronize array references is necessary to increase the amount of available instruction-level parallelism.

- Advanced Compiler Techniques: Using trace scheduling would provide a larger number of operations to schedule concurrently. Software pipelining would improve loop performance without the code explosion of loop unrolling.

- Register Allocation: A register allocator which operates on a fixed number of registers and generates spill code if necessary is needed for a real system.

- Subroutines: A proper subroutine calling convention allowing reentrant code and recursion would provide better programming facilities to the user.

In order to make ISC a tool for a massively parallel computer, other enhancements must be added. These include the ability to partition programs across multiple nodes, to communicate with remote nodes, and to manage global names.

4.5 Simulation Modes

Since an input program specifies the amount of threading and the compiler can adjust how operations are assigned to clusters, five different modes of operation are possible. Each mode corresponds to a different type of machine and is described below:

1. Sequential (SEQ): The program is written using only a single thread, and the compiler schedules the operations on only one cluster. This is similar to a statically scheduled machine with an integer unit, a floating point unit, a memory unit, and a branch unit.

2. Statically Scheduled (STS): As in Sequential mode, only a single thread is used, but there is no restriction on the clusters used. This approximates a VLIW machine without trace scheduling.

3. Ideal: The program is single threaded, has its loops unrolled as much as possible, and is completely statically scheduled. This mode is not available for those benchmarks with data dependent control structures, as they cannot be statically scheduled.

4. Thread Per Element (TPE): Multiple threads are specified, but each thread is restricted to run on only one cluster. Static load balancing is performed to schedule different threads on different clusters. A thread may not migrate to other clusters, but the benchmark programs are written to divide work evenly among the clusters.

5. Coupled: Multiple threads are allowed and function unit use is not restricted.

Threading is specified using the partitioning constructs in PCTL and the `*mode*` switch determines where a thread's operations will be scheduled. These modes will be used in Chapter 6 as the experimental machine models.

4.6 Summary

The Instruction Scheduling Compiler (ISC) translates PCTL (Processor Coupling Test Language) programs into Processor Coupling Simulator (PCS) assembly code. The input to ISC is a PCTL program and a hardware configuration file; the output is assembly code and a PCS configuration file. PCTL is a simple language that includes primitive data structures and allows the programmer to specify synchronization and program partitioning explicitly.

ISC optimizes the program using common subexpression elimination and constant folding, and uses data dependency analysis to discover instruction-level parallelism. The scheduler packs independent operations into wide instruction words, attempting to place operations to maximize instruction-level parallelism while minimizing data movement between function units.

The hardware configuration file tells the compiler about the target machine including the number and type of function units, the partitioning of function units into clusters, and the number of destination registers that can be specified in an operation. The programmer can set the `*use-moves*` switch to tell the compiler how to transfer data between function units. The `*mode*` switch determines how the scheduler assigns operations to clusters. Operations from a thread can be restricted to a single cluster, or they can be spread across the clusters to exploit instruction-level parallelism.

ISC does not perform trace scheduling or software pipelining and does not schedule code across basic block boundaries. Loops must be unrolled by hand and procedures are implemented as macro-expansions. Although a few modern compilers do perform trace scheduling and software pipelining, ISC provides a good lower bound on the quality of generated code. Using more sophisticated scheduling techniques should benefit Processor Coupling at least as much as other machine organizations.

The integrated experimental environment, including PCTL, ISC, and PCS (the Processor Coupling Simulator described in Chapter 5) is able to simulate a variety of machines. At one extreme is a simple superscalar processor with one integer unit, one floating point unit, one memory unit, and one branch unit. At the other extreme is the coupled machine using both compile time and runtime scheduling techniques to manage many function units. A statically scheduled machine and a multithreaded multiprocessor can be simulated as well.

Chapter 5

Processor Coupling Simulation

This chapter describes the Processor Coupling Simulator (PCS), a flexible functional level simulator that runs assembly code generated by the Instruction Scheduling Compiler (ISC). ISC and PCS are linked through shared configuration information; together they can model machines with different configurations of function units and clusters. Simulation is at a functional level rather than at a register transfer level, but PCS is accurate in counting the number of cycles and operations executed. Figure 5.1 shows a diagram of the simulator's interfaces to the outside world. The inputs to PCS are a program data file, an assembly code file, and a hardware configuration file. PCS executes the program and generates statistics including cycle counts, dynamic operation counts, number of cache hits and misses, function unit utilization, and an instruction-level parallelism histogram. PCS is implemented in C++ [Str87] and the source code can be found in [Kec92b].

Section 5.1 describes the operation of PCS in further detail, including the simulator inputs and outputs, as well as the models used for the function units, the communication network, and the memory system. The PCS configuration file, the statistics file, and the trace file are shown for the matrix-vector product program of Chapter 4. The assumptions made by the simulator about the Processor Coupled machine are stated in Section 5.2. Finally, Section 5.3 summarizes the simulator's capabilities.

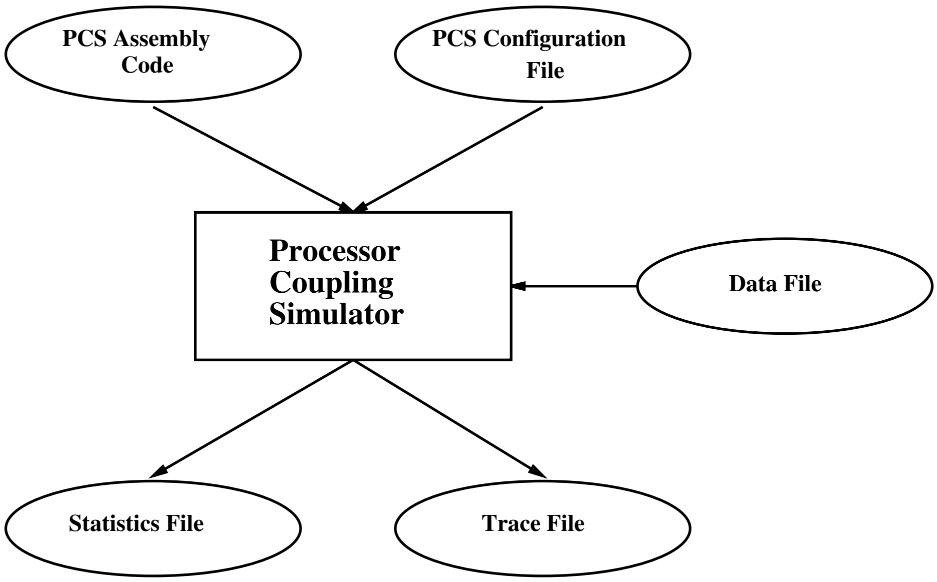

Figure 5.1: Simulator inputs and outputs.

5.1 Processor Coupling Simulator

5.1.1 Simulator Structure

PCS is implemented as a collection of modules; each function unit is a C++ object derived from a common function unit type. The memory system and the Cluster Interconnection Network (CIN) are individual modules as well. More accurate models with the same module interface can easily be substituted. Figure 5.2 shows the simulator's model of a cluster with an integer unit, a floating point unit, and a memory unit. The registers can be accessed by all three function units simultaneously. The memory system is shared by all clusters and its interface consists of one request port and one reply port per memory unit. The CIN interface has request and acknowledge signals; it performs arbitration internally.

A monitoring module contains counters for instructions, cycles, memory accesses, cache hits and misses, and CIN conflicts. The function unit, memory, and interconnection modules access these counters through special interface functions. For example, when an operation is issued, its type and the function unit that issued it are logged by the monitor. If execution tracing is on, the operation and its operand values are printed to the trace file. When the

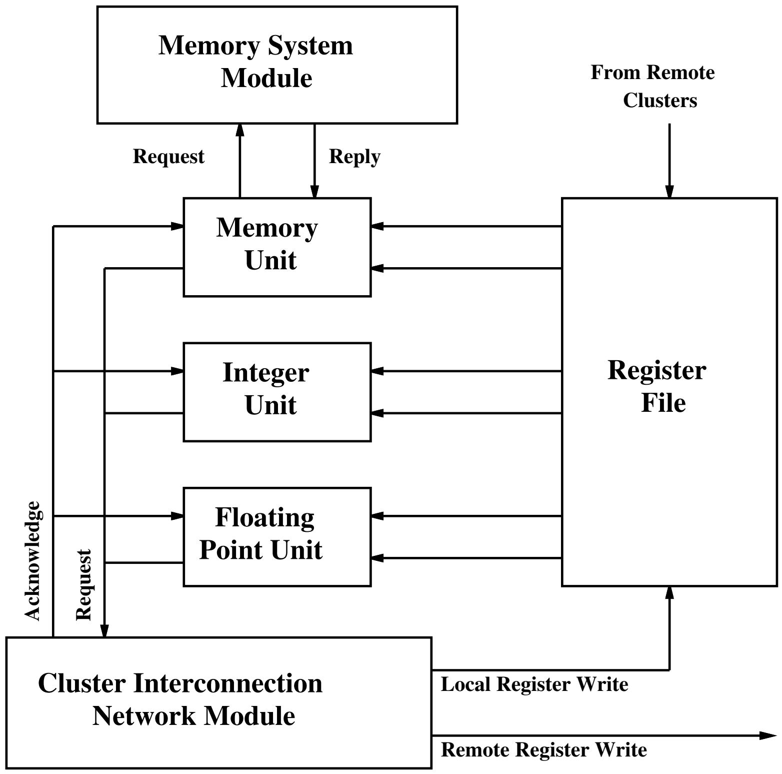

Figure 5.2: Simulator cluster model.

simulator completes, the monitor tabulates the statistics and prints them to the output file.

5.1.2 Inputs

The inputs to PCS are the assembly code file, the data file, and the configuration file. The data file provides the simulator with input values, since neither input nor output is supported by PCTL. The simulator first initializes memory with the values specified in the file before executing any code. When simulation completes, the final contents of memory are printed to the output file.

The PCS configuration file is generated by augmenting the ISC configuration file. ISC contributes an additional field to each register file entry displaying the number of registers needed. The compiler also provides the amount of node memory required for simulation, the maximum amount of local memory needed by a thread, and the name of the assembly code file. The memory size, number of registers, and maximum number of threads are used

```
100 13          /* Number of threads, function units */
5 5             /* Number of clusters, register files */
2               /* Number of destination registers to be
                   specified in an instruction */
mat-vec.asm     /* Assembly code file */
59              /* Number of assembly code instructions */
4               /* Number of local variables per thread */

cluster         /* Cluster 1 */
regfile  7
int 1
mem 1
float 1
cluster         /* Cluster 2 */
regfile  7
int 1
mem 1
float 1
cluster         /* Cluster 3 */
regfile  7
int 1
mem 1
float 1
cluster         /* Cluster 4 */
regfile  7
int 1
mem 1
float 1
cluster         /* Cluster 5 */
regfile   2
branch 1

input           /* Cluster communication specifier */
100             /* Number of global buses */
3               /* Total number of register ports */
1               /* Number of locally reserved register ports */

25              /* Number of necessary memory locations */
1000            /* Starting memory address */
2               /* Number of banks */
1 0             /* Cache hit latency, miss rate */
1 1             /* Minimum and maximum miss penalties */
mat-vec.trace   /* Trace output file */
mat-vec.dat     /* Data input file */
```

Figure 5.3: A sample PCS configuration file for a machine with five clusters. The physical parameters are specified first, followed by cluster descriptions, communication parameters, and memory system parameters.

by the simulator to allocate internal memory efficiently.

The user must designate the CIN and memory system parameters. These provide a means to investigate the effects of restricted communication strategies and variable memory latencies on the performance of different machine models and can be changed without recompiling the program. These configuration options are discussed further in Sections 5.1.5 and 5.1.6. Finally the user must provide names for the data input file and instruction trace output file.

Figure 5.3 shows the PCS configuration file produced by compiling the matrix-vector product program in Chapter 4. The machine model has five clusters and five register files. Two destination registers may be targeted in each instruction. The first four clusters have an integer unit, a floating point unit, and a memory unit, while the last cluster has only a branch unit. Each unit's execution pipeline is one stage deep and takes a single cycle. Register files in clusters 1–4 have 7 registers each, but cluster 5 only has 2 registers. Each register file has 3 register write ports with one reserved for local register writes. The 100 buses specified ensures that communication is limited only by the availability of register file write ports. A memory operation that hits in the cache has a one cycle latency. The miss rate is 0 percent so the minimum and maximum miss penalties are irrelevant. The assembly code is located in the file named `mat-vec.asm` while the trace output file and the input data file are `mat-vec.trace` and `mat-vec.dat` respectively.

5.1.3 Simulation Sequencing

A PCS simulator cycle is divided into the three pipeline phases shown in Figure 5.4. The first phase fetches an operation from the operation cache. The second performs the operation in the function unit's execution pipeline. The third phase writes the result back to the appropriate register file. The top level simulation loop executes a phase by sequencing the function units in order. All function units complete a phase before the next phase is begun. Thus, PCS simulates the fetch-execute-write cycle as if it occurs synchronously and concurrently on all function units.

The operation fetch and register write phases each take a single cycle. Since the number of execution pipeline stages of different function units may vary, operations issued on

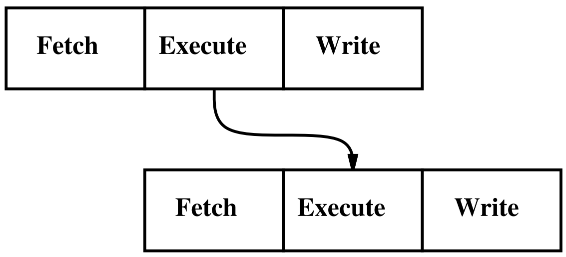

Figure 5.4: The simulator function unit pipeline has three stages. The Fetch and Write stages are one cycle each. The Execute stage shown is one cycle but it can be specified by the user. Results are bypassed from one Execute stage to the next so that one thread can issue an operation on every cycle.

---

the same cycle may complete on different cycles. For any given thread, all operations in an instruction are fetched simultaneously and only after all operations from the previous instruction have issued. There are no branch delay slots and a branch operation stalls the branching thread until the target address is available. Other threads may continue to execute.

5.1.4 Function Unit Models

As shown in Figure 5.5, the simulator's representation of a function unit contains an ALU pipeline, a result register, an operation cache, and the functions to select and sequence operations from active threads. The thread control functions access all of the active thread data structures to determine which operation to issue to the pipeline. ALU results are stored in the Result Register while waiting to be sent to the interconnection module.

During the fetch stage, a function unit fetches operations from each thread that has successfully issued all operations from its previous instruction. Operations are retrieved from the operation cache and placed in the operation buffer.

During the execute phase a function unit examines each active thread's operation buffer. If an operation that has its register requirements satisfied is found, it is issued to the execution pipeline. If no valid operation if found, a `nop` is sent instead. The pipeline then advances one stage. Threads are prioritized by time of creation so that the first thread always has the best chance of issuing an operation.

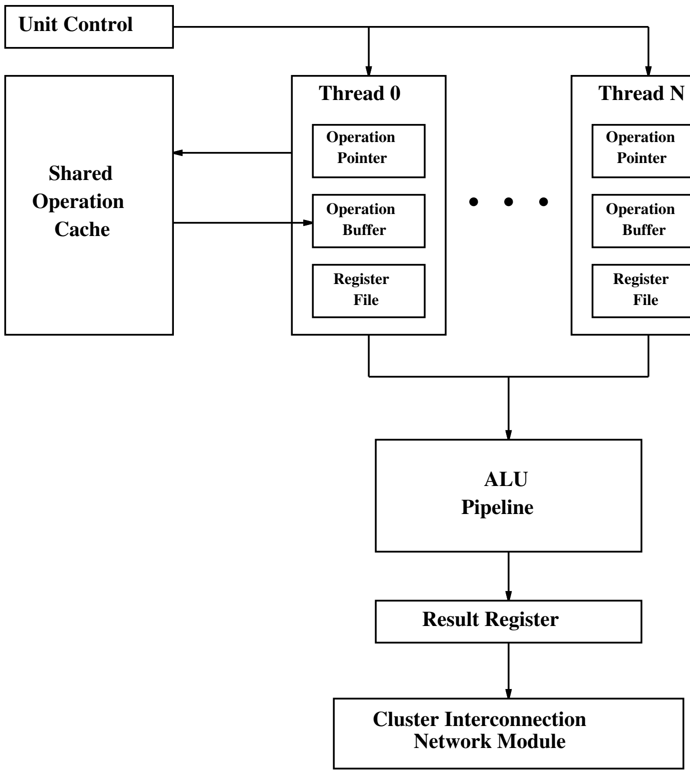

Figure 5.5: Simulator function unit model. Each function unit has an operation pointer and an operation buffer for each thread. The register file for a thread is shared with other function units in the cluster. The operation cache is shared by all active threads.

In the register write phase, the function unit attempts to send its result to the appropriate register file. If the register write fails, then the function unit stalls until the resources to perform the write can be obtained on a later cycle. Register writes that succeed can also bypass the register file so that operations requiring the data can issue immediately.

The compiler guarantees that each function unit will execute only appropriate operations. Function unit types are `int`, `float`, `mem`, `mov`, and `branch`. Each of these units is described below.

- Arithmetic Units: The integer and floating point units (`int` and `float`) execute arithmetic and logical operations. In addition, the `imov` (integer unit move) and `fmov` (floating point unit move) operations allow an operand to pass through the ALU unchanged so that it can be transferred to another cluster.

- Memory Unit: The memory unit (`mem`) issues requests to the memory system and sends the results to the Cluster Interconnection Network. Only simple addressing modes which can be expressed by adding two operands are allowed. Operands may be immediate constants or they may reside in registers.

- Move Unit: The move unit (`mov`) is responsible for transferring data between different clusters' register files. The compiler can be directed to use `imov` and `fmov` operations in the arithmetic units instead, rendering the move unit unnecessary.

- Branch Unit: The branch unit (`branch`) executes flow of control operations such as conditional and unconditional branches as well as thread control operations such as `fork` and `exit`. Branch operations cause the other function units to stall until the target address is available. Optimizations such as branch prediction or delay slots are not used.

5.1.5 Cluster Communication

The function units request register file ports and bus access from the Cluster Interconnection Network (CIN) module which arbitrates when conflicts occur. Different configurations can be selected to explore the effects of restricting interconnection bandwidth and the number of register ports between clusters. After arbitration, the CIN module sends acknowledgments to those units which are granted use of the buses and ports; it then routes the data to the appropriate register files. In some network configurations all of the function units may write results simultaneously, while more restrictive schemes require blocked units to stall until the ports or buses become available.

The communication specifier in the PCS configuration file consists of four fields: interconnection type, number of global buses, number of total register ports, and number of local register ports. The number of remote transfers that may take place simultaneously is limited by the number of buses. The number of register ports specifies the total number of write ports for a register file. The number of local register ports indicates how many of the register file write ports are reserved for use within the cluster. Figure 5.6 shows a configuration with two local buses, four register ports, and two locally reserved register

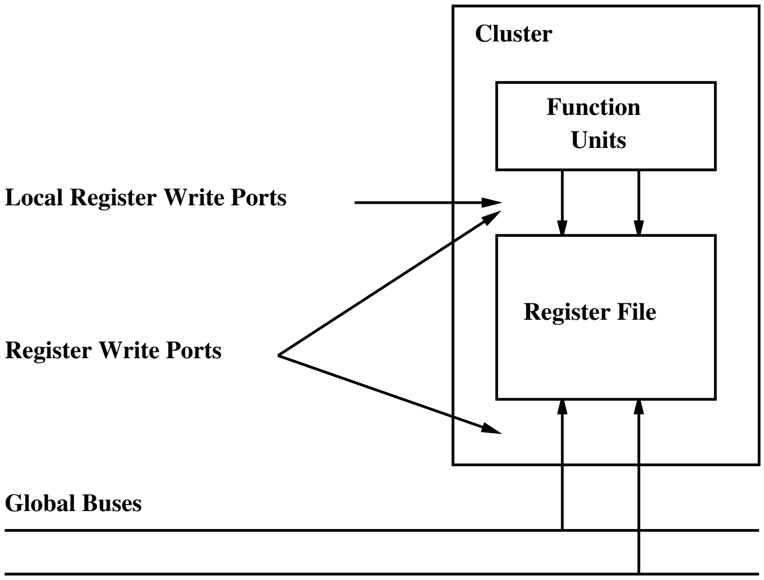

Figure 5.6: Interconnection network buses and ports. Writes to the local register file can use any of the register file ports. Remote writes may not use the local register write ports.

ports.

The three main interconnection types each have their own arbitration functions. The type `full` specifies that all function units are fully connected and there are no shared resources. The type `input` indicates that register writes between function units must compete for register file ports but not for global buses. The type `sharedbus` is similar to `input` except that writes must compete for access to shared buses and register file ports. Function units have identification numbers generated from the configuration file and request resources from the CIN in that order.

5.1.6 Memory System Model

The memory system module interface consists of a request and reply port for each memory unit. On a memory read, a memory unit sends an address to the request port. Some number of cycles later the result is returned on the reply port of the memory unit in the destination cluster. The destination cluster may be local or remote. On a memory write, a memory unit sends both address and data to the memory system via the request port;

no reply is necessary. The memory system module models memory latencies according to the specifications in the configuration file and automatically queues synchronizing requests if their preconditions are not met.

The configuration file specifies the following memory system parameters: starting memory address, number of banks, cache hit latency, cache miss rate, and cache minimum and maximum miss penalties. The compiler generates the number of memory locations required by the program, the address where memory starts, and the number of memory banks. The user can control the latency of memory operations by varying the other four cache parameters. Ideal cache behavior (no misses) can be selected by making the miss rate zero. If misses are allowed, a random number of accesses, specified by the miss rate, will incur a miss penalty. The miss penalties are uniformly randomly distributed between the minimum and maximum values fixed in the configuration file.

5.1.7 Threads

Each thread has its own instruction pointer and register set but shares the function units, memory system, and cluster interconnection module. When a thread is forked, the simulator creates a Context Pointer (CP) and a register file in each cluster. An Operation Pointer (OP) and an Operation Buffer are created for each function unit within a cluster. Figure 5.7 shows the state associated with a thread. The CP resides in register 0 in all of the thread's register files and points to a context consisting of newly allocated local memory for the thread. The entire program is loaded into the operation caches before execution, and the OP indexes into the operation cache to fetch the the thread's operations. The operation buffer holds operations waiting for register synchronization before being issued. When a thread completes, all of its state and local memory is reclaimed by the simulator.

When the simulator begins executing the program, the first thread is automatically started. Subsequent threads are created when the branch unit executes a `fork` operation. The new thread is allocated state registers and context memory, and is added at the end of the thread priority queue. The global thread priority queue maintains a list of active threads in the order in which they were created.

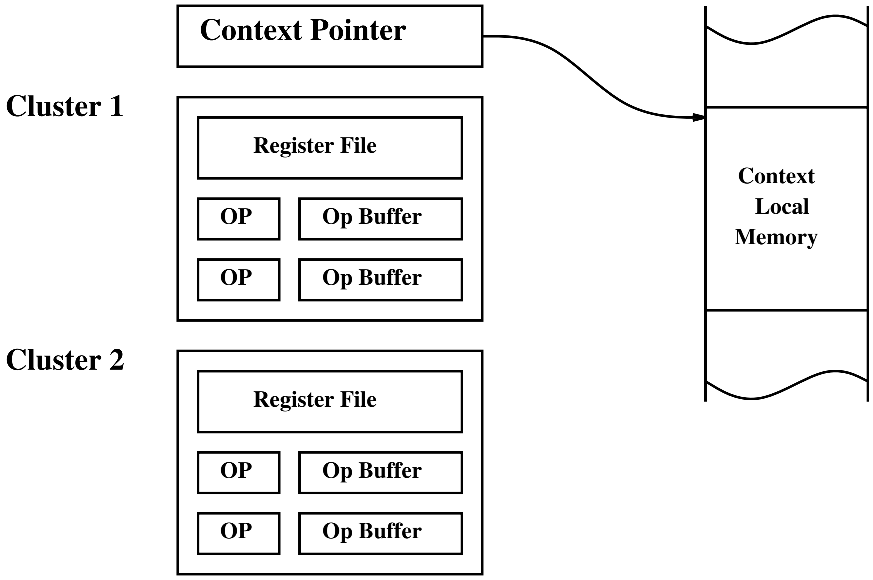

Figure 5.7: The state associated with a thread. This thread is for a machine with two clusters, each with two function units. The thread has a register file in each cluster and an Operation Pointer (OP) and Operation Buffer in each function unit. The Context Pointer (CP) points to the thread's local memory.

5.1.8 Output

Simulation terminates when all threads have completed or if no further progress can be made. The program may deadlock if it has a synchronization error. By checking forward progress, the simulator detects these occurrences and notifies the user. Upon termination, the simulator produces an output file and optionally a trace file.

Output File

The output file is composed of three parts. The first enumerates the complete machine configuration that was simulated, including function unit, memory system, and communication network parameters. The second lists the final memory contents so the user can determine if the program produced the correct result.

The last part gives the statistics tabulated during execution. Figure 5.8 shows sample statistics generated from running the matrix-vector product program. The parallelism his-

```
Parallelism of  0 is    0 cycles and  0.00%
Parallelism of  1 is    6 cycles and 10.53%
Parallelism of  2 is    4 cycles and  7.02%
Parallelism of  3 is    2 cycles and  3.51%
Parallelism of  4 is   14 cycles and 24.56%
Parallelism of  5 is   11 cycles and 19.30%
Parallelism of  6 is   13 cycles and 22.81%
Parallelism of  7 is    4 cycles and  7.02%
Parallelism of  8 is    3 cycles and  5.26%
Parallelism of  9 is    0 cycles and  0.00%
Parallelism of 10 is    0 cycles and  0.00%
Parallelism of 11 is    0 cycles and  0.00%
Parallelism of 12 is    0 cycles and  0.00%
Parallelism of 13 is    0 cycles and  0.00%

Unit   Type       Ops Executed     Utilization      Bus Conflicts
  0    int             16             28.07%              0
  1    mem             34             59.65%              0
  2    float            8             14.04%              0
  3    int             16             28.07%              0
  4    mem             34             59.65%              0
  5    float            8             14.04%              0
  6    int             16             28.07%              0
  7    mem             34             59.65%              0
  8    float            8             14.04%              0
  9    int             16             28.07%              0
 10    mem             34             59.65%              0
 11    float            8             14.04%              0
 12    branch          29             50.88%              0

  Unit type          # Operations    % Utilization    Ops Per Cycle
   integer                 64           0.2807           1.12
   float                   32           0.1404           0.56
   moves unit moves         0
   integer unit moves       0
   FPU moves                0
   loads                   88           0.5965           2.39
   stores                  48
   branches                29           0.5088           0.51
Number of memory references    = 136
  Total number of cache hits   = 136
  Total number of cache misses =   0
Average memory access latency  = 1.0
Number of cycles executed      =  57
Number of operations executed  = 261
Average parallelism            = 4.58
Utilization for entire machine = 35.22%
```

Figure 5.8: The simulator output file for matrix-vector product containing only the statistics generated during execution. A complete output file includes configuration information and the contents of memory.

togram shows the number of cycles in which a given number of function units are used. In matrix-vector product, 14 cycles have parallelism of 4, while only 3 cycles have parallelism of 8. The function unit operation counts and utilization percentages are shown next. The *bus conflict* field enumerates the number of times units were stalled due to contention for register file ports or buses. Function unit operation counts and utilization are categorized by type and given in the subsequent table. The number of memory references, cache hits, and cache misses shows the effect of the memory system model. Average memory access latency is given in cycles. Finally, the total cycle count, operation count, average parallelism, and overall utilization are tallied.

Trace file

The trace file contains a history of the operations executed during simulation. Figure 5.9 shows the first 8 cycles of the trace file for the matrix-vector product program. The *unit* and *thread* fields of each entry show the function unit and the thread that executed the operation. The *address* field displays the operation's absolute address in the assembly code file. Runtime values of registers instead of register numbers are found in the *operand* fields. The remaining fields correspond to the assembly operations described in Section 4.2.7 of Chapter 4.

5.2 Simulation Assumptions

Since simulation takes place at a functional, rather than a register transfer level, certain simplifying assumptions were made. Many were mentioned in the description of the simulator structure, but those that warrant further discussion are thread behavior, the memory system, and the Cluster Interconnection Network.

5.2.1 Thread Behavior

Although threads can be created and destroyed dynamically, the simulator places no limit on the number of active threads. For the benchmarks used in this experimental evaluation, at most 20 threads are active, but examination of the trace files shows that usually fewer

```
Unit  Thread  Address  Opcode AccCd #Rands DestID  Dest  DestID  Dest   Op1  Op2

Clock cycle 1:
12      0      1000    forka    0      0    U-1    #38    U-1    R-1    #0   #38

Clock cycle 2:
 1      1      1038    stor     0      1    U-1    R0     U-1    R-1    #0   #1027
 4      1      1038    stor     0      1    U-1    R0     U-1    R-1    #0   #1026
 7      1      1038    stor     0      1    U-1    R3     U-1    R-1    #3   #1025
12      0      1001    forka    0      0    U-1    #26    U-1    R-1    #0   #26

Clock cycle 3:
 4      2      1027    stor     0      1    U-1    R0     U-1    R-1    #0   #1031
 7      2      1027    stor     0      1    U-1    R0     U-1    R-1    #0   #1030
10      2      1027    stor     0      1    U-1    R2     U-1    R-1    #2   #1029
12      0      1002    forka    0      0    U-1    #14    U-1    R-1    #0   #14

Clock cycle 4:
 1      3      1016    stor     0      1    U-1    R0     U-1    R-1    #0   #1035
 7      3      1016    stor     0      1    U-1    R1     U-1    R-1    #1   #1033
10      3      1016    stor     0      1    U-1    R0     U-1    R-1    #0   #1034
12      0      1003    forka    0      0    U-1    #2     U-1    R-1    #0   #2

Clock cycle 5:
 1      4      1005    stor     0      1    U-1    R0     U-1    R-1    #0   #1038
 4      4      1005    stor     0      1    U-1    R0     U-1    R-1    #0   #1037
10      4      1005    stor     0      1    U-1    R0     U-1    R-1    #0   #1039
12      0      1004    exit     0      0    U-1    R-1    U-1    R-1    #0   #-1

Clock cycle 6:
12      1      1038    bf       0      0    U-1    #8     U-1    R-1    #1   #8

Clock cycle 7:
 1      1      1039    load     0      1    U0     R1     U-1    R-1    #-1  #1025
 4      1      1039    load     0      1    U0     R3     U1     R1     #-1  #1026
 7      1      1039    load     0      1    U2     R1     U-1    R-1    #-1  #1027
12      2      1027    bf       0      0    U-1    #8     U-1    R-1    #1   #8

Clock cycle 8:
 0      1      1040    mul      0      1    U0     R2     U-1    R-1    #3   #4
 3      1      1040    add      0      1    U1     R2     U-1    R-1    #0   #1
 4      1      1040    load     0      1    U0     R6     U-1    R-1    #-1  #1016
 7      2      1028    load     0      1    U3     R3     U2     R1     #-1  #1030
10      2      1028    load     0      1    U3     R1     U-1    R-1    #-1  #1029
12      3      1016    bf       0      0    U-1    #8     U-1    R-1    #1   #8
```

Figure 5.9: The first 8 cycles of the simulator trace file for matrix-vector product.

than 6 threads are issuing operations during any one region of the code. Since all threads are in the working set, no thread management, such as swapping an idle active thread for an inactive thread, is necessary. The simulator creates thread state, allocates contexts, and places the context pointer in register 0 instantaneously with no overhead. Likewise, destroying a thread has no cost.

In a real implementation, the number of active threads and the cost of thread management affects the performance of parallel programs. However, in this study the benchmarks use a small number of threads, and thread management is ignored. Thread management for a Processor Coupled node that may have multiple live threads is an open area of research.

5.2.2 Memory System

The PCS cache is divided into an infinite number of banks; memory accesses cannot be blocked by bank conflicts. The cache is modeled statistically by specifying the hit latency, the miss rate, and a variable miss penalty in the configuration file. On each memory access, a random number is generated to determine if the access misses in the cache. If a miss occurs, the additional latency is randomly chosen between the minimum and maximum miss penalties specified in the configuration file. A miss on a `load` delays the return of the data to a memory unit by the cost of the miss. A miss on a `store` delays the storing of the data. Memory references are non-blocking so that a thread may proceed after issuing a memory operation. Locality of reference is not modeled by the memory system module. This memory model approximates the latencies that might be seen in a massively parallel distributed memory machine and is sufficient for showing the effects of dynamic latencies on Processor Coupling performance.

A memory cell has three fields: a presence bit, a data word, and a queue. The memory system provides fine grain synchronization by using the queue to hold outstanding memory requests. The thread identifier and the destination specifier for synchronization operations waiting for a previous access to complete are automatically placed in the location's queue. The simulator performs enqueueing and dequeueing with no additional overhead. An implementation would use a polling scheme or a software trap mechanism to create synchronization structures.

When simulation begins, the entire program is loaded into the operation caches. Instruction cache misses are not included in the simulation. Additionally, the instruction streams are stored in the operation caches in unpacked form, meaning that a vacancy in a thread's instruction is held by a `nop`. In an implementation, storing `nops` wastes space in the operation caches; an encoding might be used to eliminate them.

5.2.3 Cluster Interconnection Network

The CIN module manages the shared register file ports and intercluster buses specified in the configuration file. Arbitration proceeds sequentially with the requests serviced in the function unit order declared in the configuration file. If an operation cannot be granted all of the necessary resources, the requesting function unit must stall; writes from an operation with two destination registers will not be split over multiple cycles. The CIN controller does not perform perfect arbitration. Granting resources in a different order might result in better utilization.

Register file write ports are divided into local and global categories. If an operation writes back to the register file in its own cluster, the arbiter first tries to schedule the access using a local register file port. If none are left, the arbiter will try to use a global register file port. Writes to register files in other clusters must compete for remote register file write ports. In `sharedbus` mode, multiple register file ports can share the same global bus. Arbitration is idealized and performed without overhead.

Since multiple threads are running simultaneously in an unpredictable fashion, the compiler cannot statically schedule the threads to eliminate register port and bus conflicts. Even single thread programs cannot be perfectly scheduled to eliminate conflicts because of the uncertainty of memory latencies. Thus the CIN parameters are not used by the compiler.

5.3 Summary

The Processor Coupling Simulator (PCS) is able to test the viability of Processor Coupling by providing a functional level simulation environment. The configuration of function units, clusters, and register files can be varied. Function unit latencies, cache miss rates and penalties, and the cluster interconnection configuration can be set by the user as well.

The simulator is dependent upon the Instruction Scheduling Compiler to produce the PCS configuration file and assembly code. After compilation, the user can define the memory system and the CIN parameters. On completion, the simulator writes the final contents of memory and the execution statistics to the output file. Statistics include dynamic operation counts, cycle counts, function unit utilization, cache performance, and the number of resource conflicts in the CIN.

The simulator assumes ideal behavior of several of its components. There is no limit on the number of active threads, and management functions such as creating and disposing of threads have no cost. The memory system is modeled statistically and it synchronizes references and queues outstanding requests automatically. Arbitration for shared buses and register ports is performed by the cluster interconnection module with no overhead cost.

Chapter 6

Experiments

This chapter presents the experimental performance results of Processor Coupling on the benchmark suite. The simulation environment described in Chapters 4 and 5 is used to generate performance results. First, the running time of the different machine models are compared. Further experiments explore the effects of interference between threads, variable memory latencies, floating point unit latencies, and restricted connectivity between function units. The number and mix of integer and floating point units are varied in another class of experiments. Finally, the results of three different methods of expressing parallel loops and of changing the strategy for moving data between function units are described.

Section 6.1 presents four benchmarks used in the evaluation of Processor Coupling. Sections 6.2 through 6.10 describe the experiments and present results comparing the performance of a Processor Coupled node to that of the statically scheduled and multiprocessor models.

6.1 Benchmarks

Table 6.1 summarizes the four simple benchmarks selected to test the effectiveness of Processor Coupling. Single-threaded and multithreaded versions of each benchmark are implemented. The code for each of the benchmarks can be found in Appendix A. Each program solves a small and well contained problem, and together the benchmarks can be used as building blocks for larger numerical applications. For example, the compute intensive por-

Benchmark	Description
Matrix	9×9 matrix multiply
FFT	32 point complex valued fast Fourier transform
Model	VLSI device model evaluation
LUD	Direct method sparse matrix lower-upper decomposition

Table 6.1: The benchmarks used to evaluate Processor Coupling.

tions of a circuit simulator such as SPICE include a model evaluator and sparse matrix solver [SV88].

6.1.1 Matrix Multiply

The **Matrix** benchmark multiplies two 9×9 matrices of floating point numbers. The matrices are represented as two dimensional arrays; the compiler generates the appropriate index calculations to access the elements.

The sequential version iterates over the outer two loops and has its inner loop completely unrolled. The threaded version executes all of the iterations of the outer loop in parallel as nine independent threads. As in the sequential version, the inner loop is unrolled completely. The Ideal version of **Matrix** has all of the loops unrolled and the entire computation is statically scheduled by the compiler.

6.1.2 Fast Fourier Transform

The **FFT** benchmark performs a 32 point decimation-in-time fast Fourier transform of complex numbers [OS75]. The algorithm consists of two phases. The first phase traverses the input vector, ordering the input values by bit-reversing their indices. The second phase has a logarithmic number of stages, each of which performs 16 butterfly computations to compute 2-point FFTs. The input vector is represented as a linear array in which each complex value vector element occupies two successive array locations.

The single-threaded and multithreaded versions of the **FFT** benchmark both use the same sequential bit-reversal code. In the second phase of the program, the sequential version has two loops. The outer loop iterates through the 5 stages while the inner loop performs all of the butterfly computations within a stage.

The multithreaded version also iterates sequentially through the stages. A new thread is created to compute each 2-point FFT so that all butterfly computations within a stage are performed concurrently. Barrier synchronization is used between iterations of the outer loop to prevent the iterations from overlapping. Without the barrier, a stale array element might be read by one iteration before the correct value is produced by the previous iteration.

In the Ideal version of **FFT**, the bit-reversal loop is unrolled completely. The five stages of the second phase are still executed sequentially, but the inner loop is unrolled and all 16 of the butterfly computations are scheduled simultaneously.

6.1.3 Model Evaluation

Model is a model evaluator for a VLSI circuit simulator in which the change in current for each node in the network is computed based upon previous node voltages. A model evaluator is required in the Newton-Raphson solution of circuit equations for DC and transient analysis [SV88].

A circuit may consist of resistors, capacitors, NMOS transistors, and PMOS transistors. Resistors and capacitors are evaluated using the simple models found in [VS83]. Resistor and capacitor models each have two terminals and a resistance or a capacitance value. The MOS transistors have gate, drain, and source terminals, a channel width, and a channel length. Transistors can operate in cutoff, triode, or saturation regions, each subject to simple device equations [GM84].

A full model evaluator will also include routines to generate a Jacobian for the variable parameters of the circuit elements, but **Model** only calculates the "right-hand side" of the circuit equation. The input circuit is represented as an array of device parameters; node voltages and incremental currents are represented by one-dimensional arrays. Each model is evaluated by applying the appropriate device equation to the voltages at its terminals. The device's contribution to the current at each node is then added to the proper entry in the global current array.

The single threaded version performs each evaluation sequentially. Since control flow depends upon the circuit configuration, loops are not unrolled. The multithreaded version evaluates all models concurrently. The PCTL `forall-iterate` construct is used to create

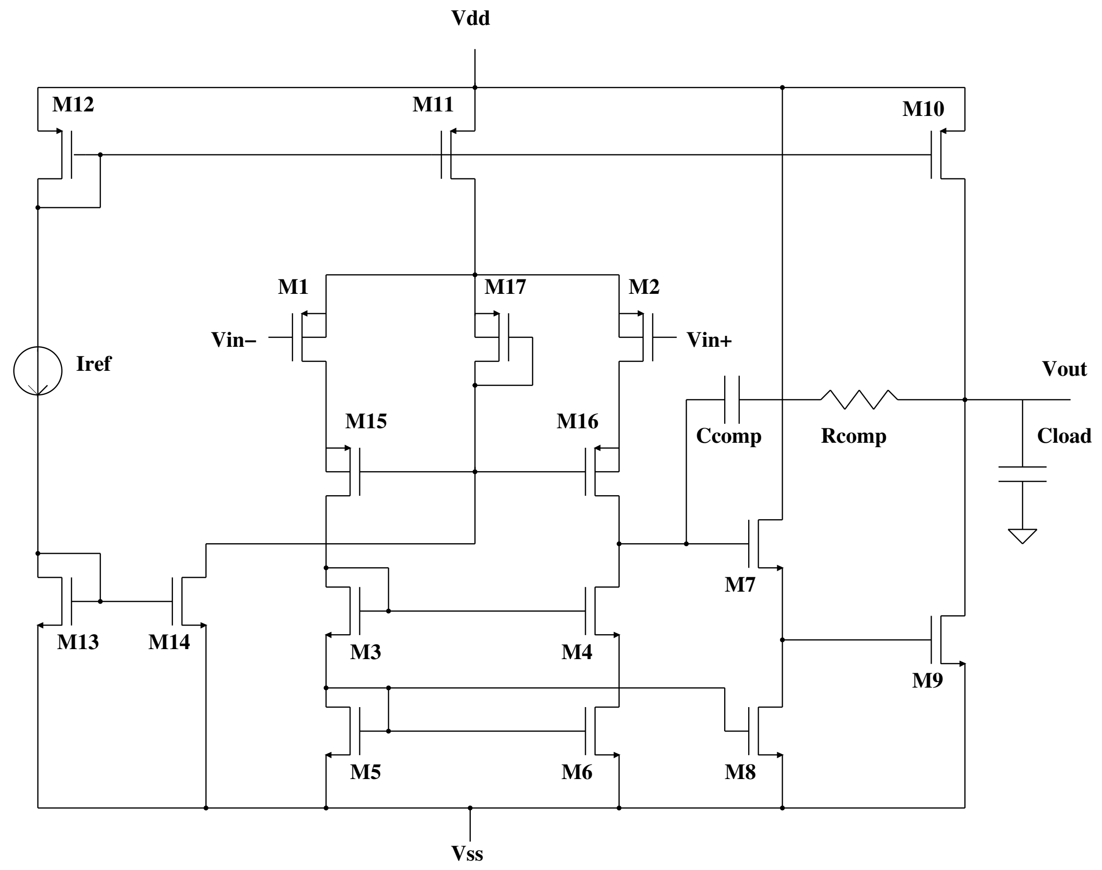

Figure 6.1: Operational Amplifier circuit used in the model evaluation benchmark.

a new thread for each device. A lock on each value in the current array guarantees atomic update. No global or barrier synchronization is necessary.

The input circuit is a high-gain, high bandwidth, two-stage CMOS operational amplifier. The circuit contains twenty elements including 9 NMOS transistors, 8 PMOS transistors, 2 capacitors and one resistor. Of the 17 transistors, 5 are in the triode region, 9 are saturated, and 3 are cutoff. The operational amplifier circuit diagram is shown in figure 6.1.

6.1.4 Sparse Matrix LU Decomposition

The **LUD** benchmark solves a sparse system of linear equations using the lower-upper decomposition technique. An off line program performs Markowitz reordering and generates fill-ins for the input matrix. The PCTL program uses an overlapped-scattered array to store the matrix [SV89]. Data structures for the diagonal elements of the array as well as indices for elements below and to the right of the diagonal are stored as dense arrays.

The **LUD** benchmark performs the decomposition using a source-row driven algorithm. The outer loop iterates over all rows of the matrix, while the inner loop normalizes and updates dependent target rows. On each iteration of the outer loop, a row is selected as the source. Each row with a non-zero element under the diagonal position of the source row is normalized and updated. After selecting a source row, all target row updates are independent. The outer loop executes sequentially, but fine grain synchronization on individual matrix elements could be used to overlap iterations.

The sequential version of **LUD** executes all loops sequentially. Since the control flow depends upon the input matrix, loops are not unrolled; there is no Ideal version of **LUD**. The multithreaded version performs all the target row normalizations and updates concurrently. After the outer loop selects a source row, a thread to update each target row is created. The `forall-iterate` statement is used to create a dynamically determined number of threads, and to balance the work across the function units. Barrier synchronization sequentializes iterations of the outer loop.

The input data used in the experiments is a 64×64 adjacency matrix of an 8×8 mesh. This input is a sparse banded diagonal matrix with 24% non-zero entries after reordering and generating fill-ins.

6.2 Baseline Comparisons

6.2.1 Machine Hardware Resources

The baseline machine consists of four arithmetic clusters and a branch cluster. Each arithmetic cluster contains an integer unit, a floating point unit, a memory unit, and a shared register file, while a branch cluster contains only a branch unit and a register file. The branch cluster may be used by any thread. Although Processor Coupling does not preclude multiple branch units, one branch cluster is sufficient for these simulations. Each function unit has a single cycle execution pipeline stage.

In the baseline machine, an operation can specify at most two destination registers. A

Machine Model	Description
SEQ	Single thread, restricted to one cluster
STS	Single thread, no cluster restrictions
Ideal	Single thread, loops unrolled and statically scheduled
TPE	Multiple threads, each thread restricted to one cluster
Coupled	Multiple threads, no cluster restrictions

Table 6.2: Machine models use in experiments.

function unit can write a result back to any cluster's register file, each of which has enough buses and ports to prevent resource conflicts. Memory units perform the operations required for address calculations. Memory references take a single cycle and have no bank conflicts. The machine models used throughout this chapter were described in detail in Chapter 4 and are summarized in Table 6.2.

6.2.2 Results

Assuming that the compiler produces the best possible schedule, the number of cycles executed by the Ideal version of a benchmark is a lower bound for a particular machine configuration. Sequential mode operation provides an upper bound since only the parallelism within a single cluster can be exploited. Table 6.3 shows the cycle counts for each machine model. Floating point utilization is calculated as the average number of floating point operations executed each cycle. Utilizations for the rest of the units are determined similarly. Figure 6.2 displays the cycle counts graphically.

In the single threaded models, STS mode requires on average 1.7 times fewer cycles than SEQ, since STS allows use of all the function units. Since the Ideal mode's program is fully unrolled and statically scheduled, loop overhead operations and redundant address calculations are eliminated. The fewer operations executed by the Ideal machine allows it to complete in an average of 7 times fewer cycles than SEQ.

In threaded mode, the cycle count for Coupled and TPE are nearly equivalent for the **Matrix**, **LUD**, and **Model** benchmarks, which have been stripped of nearly all sequential execution sections, and are easily partitionable. TPE is as fast as Coupled since the load is balanced across the its clusters. Processor Coupling will have an advantage in less intrinsically load balanced computations, as long as threads are not allowed to migrate between

Benchmark	Mode	#Cycles	Compared to Coupled	Utilization FPU	IU	Memory	Branch
Matrix	SEQ	1991	3.13	0.69	0.90	0.91	0.05
Matrix	STS	1181	1.85	1.17	1.52	1.53	0.09
Matrix	TPE	628	0.99	2.19	2.84	2.87	0.17
Matrix	Coupled	637	1.00	2.16	2.80	2.83	0.17
Matrix	Ideal	349	0.55	3.95	0.28	0.70	0.00
FFT	SEQ	3376	3.07	0.24	0.61	0.55	0.05
FFT	STS	1791	1.63	0.45	1.25	1.04	0.09
FFT	TPE	1976	1.79	0.40	1.05	0.96	0.20
FFT	Coupled	1101	1.00	0.73	2.03	1.72	0.36
FFT	Ideal	401	0.36	2.00	2.55	2.68	0.02
Model	SEQ	992	2.70	0.21	0.10	0.82	0.14
Model	STS	770	2.09	0.28	0.13	1.04	0.18
Model	TPE	394	1.07	0.54	0.65	1.77	0.60
Model	Coupled	368	1.00	0.58	0.70	1.82	0.64
LUD	SEQ	57974	2.69	0.14	0.45	0.98	0.08
LUD	STS	33125	1.54	0.24	0.78	1.72	0.14
LUD	TPE	22626	1.05	0.35	1.35	2.71	0.35
LUD	Coupled	21542	1.00	0.37	1.42	2.85	0.37

Table 6.3: Cycle count comparison of different types of machines. Floating point unit (FPU) utilization is given as the average number of floating point operations executed each cycle. Integer, memory, and branch unit utilizations are calculated similarly.

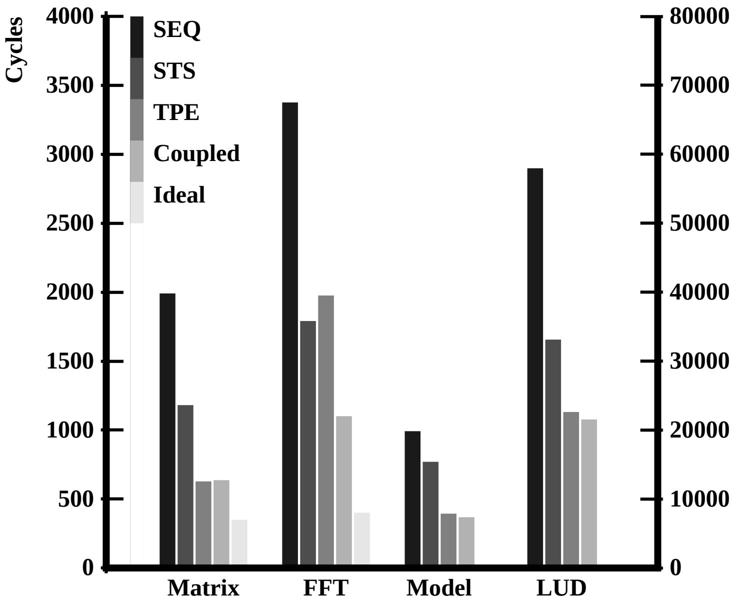

Figure 6.2: Baseline cycle counts for the five simulation modes. **Matrix**, **FFT**, and **Model**, are referenced to the left axis, while the scale for **LUD** is on the right. Ideal is only implemented for **Matrix** and **FFT**.

clusters.

One advantage of Coupled over TPE is found in sequential code execution. For example, **FFT** has a large sequential section that cannot be partitioned. Since each TPE thread can execute on only one cluster, performance in the sequential section is no better than SEQ. In fact, because of the dominance of the sequential section, TPE does not even perform as well as STS on the entire benchmark. Coupled, on the other hand, performs as well as STS on sequential code. The available instruction-level parallelism can be exploited by Coupled, but not by TPE. Since asymptotic parallel speedup of a program is limited by the amount of sequential code, single thread performance is important.

One of Processor Coupling's advantages over a statically scheduled scheme is the increased unit utilization allowed by dynamic scheduling. This capability allows Coupled mode to execute an average of 1.8 times fewer cycles than the statically scheduled STS mode. TPE, with its multiple threads, also executes faster than STS for all benchmarks but **FFT**. This reduced cycle count in TPE and Coupled is due to the fine grained interleaving of threads. The multiple threads of Coupled mode result in much higher function unit utilization, and therefore a lower cycle count, than STS.

6.3 Utilization

Figure 6.3 shows for each benchmark the utilization of the floating point units, integer units, memory units, and branch units. In all benchmarks, unit utilization increases as the simulation mode approaches Ideal. For **Matrix** the utilization of FPUs, IUs, and memory units is similar in each mode except Ideal. In Ideal mode, the FPU utilization is 3.9, which indicates that the compiler has filled nearly every floating point operation slot. Note that since the compiler has eliminated loop overhead operations and common subexpressions in array index calculations, very few integer and branch operations are required. Furthermore, a significant fraction of the memory operations have been replaced by register operations.

Aside from Ideal mode, the unit utilization in **FFT** is similar. The multiple active threads available in TPE and Coupled modes drive memory unit utilization up. This rise results from the many read and write operations performed by the butterfly calculations of the inner loop. Memory utilization in the Ideal mode is still high because the compiler

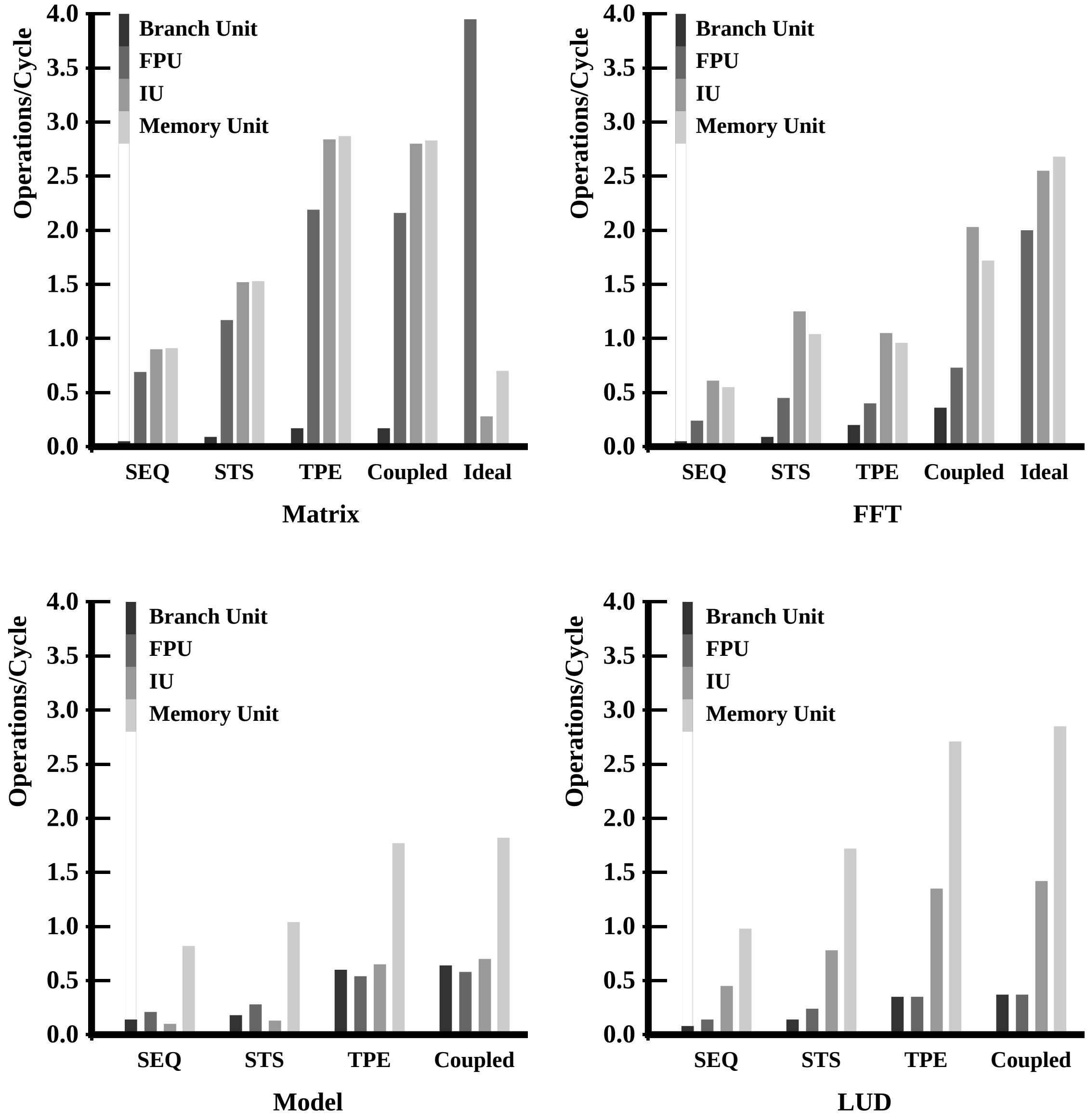

Figure 6.3: Function unit utilization.

Mode	Thread	Compile Time Schedule	Runtime Cycle Count	Devices Evaluated
STS	1	25	25.0	20
Coupled	1	23	28.0	8
Coupled	2	23	38.7	6
Coupled	3	23	77.3	3
Coupled	4	23	80.7	3

Table 6.4: Average cycle counts for each iteration of the inner loop of the **Model** benchmark for STS and Coupled, with threads assigned different priorities.

was unable to replace memory references with register references. The execution of the **Model** and **LUD** benchmarks is dominated by memory reference operations. Thus, even in Coupled and TPE mode, the integer and floating point utilizations are still quite small.

6.4 Interference

For multithreaded versions of the benchmarks, the statically scheduled threads interfere with one another, causing the runtime cycle count to be longer than the compile time schedule would suggest. To demonstrate this dilation, a slightly modified version of **Model** is used in Coupled mode. Four threads are created when program execution begins. Each thread accesses a common priority queue of devices to be evaluated, chooses a device, updates the queue, and then evaluates the device. This loop is repeated by all threads until the queue is empty. A new input circuit consisting of 20 identically operating NMOS transistors is used. This allows the code evaluating the other device models to be removed so that every operation specified in the new source program is executed. This provides the capability to compare runtime cycle count with the number of instructions generated by the compiler. With the modified benchmark, the effect of a thread's priority on its runtime schedule can be seen more clearly. The Coupled benchmark is compared to a similarly altered version of an STS mode program.

Table 6.4 shows the compile time schedule length and the average runtime cycle count to evaluate one model for each of the four threads in Coupled mode. The higher priority threads execute in fewer cycles. In STS mode, there is only one thread, and it runs in the same number of cycles as the static schedule predicts.

In Coupled mode even the highest priority thread requires more cycles than the schedule predicts. This is due to contention between threads for the shared queue. Taking a weighted average across the four threads, Coupled mode requires 46.5 cycles per device evaluation. Although STS requires only 25 cycles per evaluation, the multiple threads of Coupled allows evaluations to overlap such that the aggregate running time is shorter (274 cycles versus 505 for STS). On single threaded code, Coupled and STS perform equally well; on threaded code, Coupled mode will execute in fewer cycles.

6.5 Variable Memory Latency

Long memory latencies due to synchronization or remote references degrade performance of any machine. Statically indeterminate memory latencies are particularly damaging to single threaded modes since delays stall the entire program's execution. Multithreaded machines hide long memory latencies by executing other threads.

A five to ten percent miss rate is assumed for an on-chip cache, depending on its size and organization. If a cache miss occurs, the memory reference must go off chip. When the requested data is in local memory, the reference might complete in 20 cycles. References to physically remote nodes can take 100 or more cycles. The three models of memory system performance used in simulations are:

- **Min**: single cycle latency for all memory references.
- **Mem1**: single cycle hit latency, 5% miss rate, and a miss penalty randomly distributed between 20 and 100 cycles.
- **Mem2**: similar to **Mem1** with a 10% miss rate.

Figure 6.4 shows how the cycle counts for different machine models are affected by long memory latencies. Since the compiler is able to use 490 registers in Ideal mode for **Matrix**, very few memory references are needed. Thus long latencies hardly affect the Ideal machine's cycle count. In **FFT**, however, the cycle count for Ideal mode increases dramatically because many loads and stores are required, and each delayed memory reference halts computation. Cycle count for STS mode rises in all benchmarks with increasing memory latency for

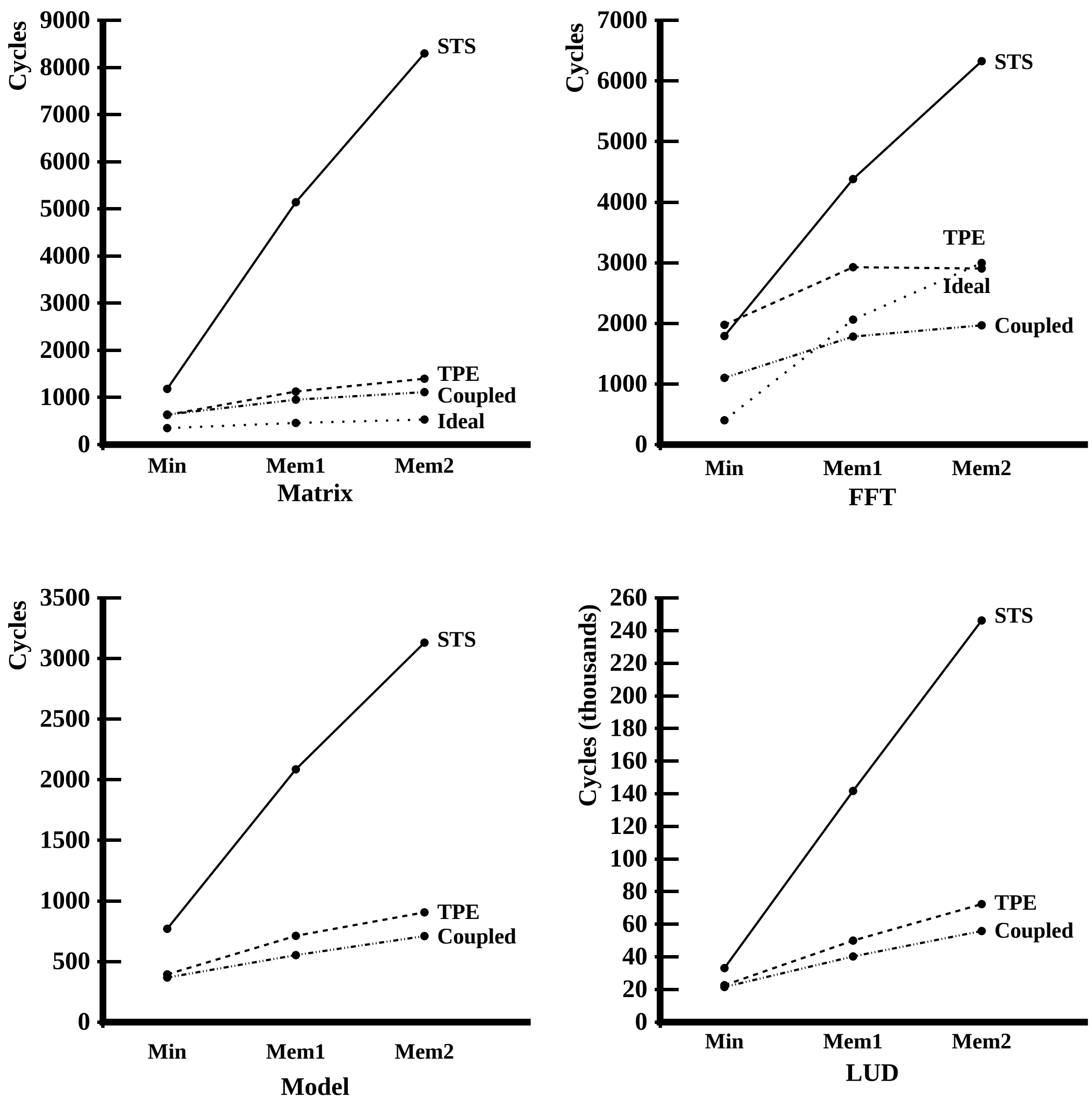

Figure 6.4: Cycle counts when memory latency is varied. Increased latency affects the single-threaded modes (STS and Ideal) more than the multithreaded modes (Coupled and TPE).

similar reasons. Nearly 5.5 times as many cycles are needed on average for execution with **Mem2** parameters as with **Min** for STS.

The cycle count for Coupled does not increase as greatly in any of the benchmarks since other threads are executed when one thread waits for a long latency reference. On average, execution with **Mem2** parameters requires twice as many cycles as **Min**. If the compiler knew which references would cause long delays, it could create a schedule to try to mask long latencies. However, since many memory latencies cannot be statically determined, runtime scheduling techniques like those of Processor Coupling can be used to mask the delay. Because memory latencies can be quite long in a distributed memory parallel machine, latency tolerance is a major advantage of Coupled over STS.

TPE is affected only slightly more severely than Coupled by long memory latencies. Execution in **Mem2** mode requires 2.3 times as many cycles as **Min**. Like Coupled mode, TPE has other threads to run while waiting for long latency memory references. However, TPE threads are allocated statically to specific clusters. If only one thread is resident on a cluster and it stalls waiting for a reference, the cluster resources go unused.

6.6 Effect of FPU Latency

Function unit latencies can also reduce the amount of available instruction-level parallelism by increasing the critical path through a basic block. However, if enough parallelism exists, the compiler can schedule operations into the gaps created by the long latency function units.

To demonstrate the effect of increasing function unit latencies, execution pipeline latencies in all floating point units are varied from 1 to 5 cycles. The rest of the machine remains the same as the baseline configuration. The floating point latency is specified in the configuration files of both the compiler and the simulator. The compiler uses this information to schedule operations into the vacancies left by long latency units.

The experimental cycle counts for STS, TPE, Coupled, and Ideal modes are shown in Figure 6.5. Results for each benchmark are quite similar. In STS mode the cycle count increases by an average of 90% when the floating point latency is increased from 1 to 5 cycles. Ideal mode does extremely well on **Matrix** and **FFT**, increasing only 6%. TPE and

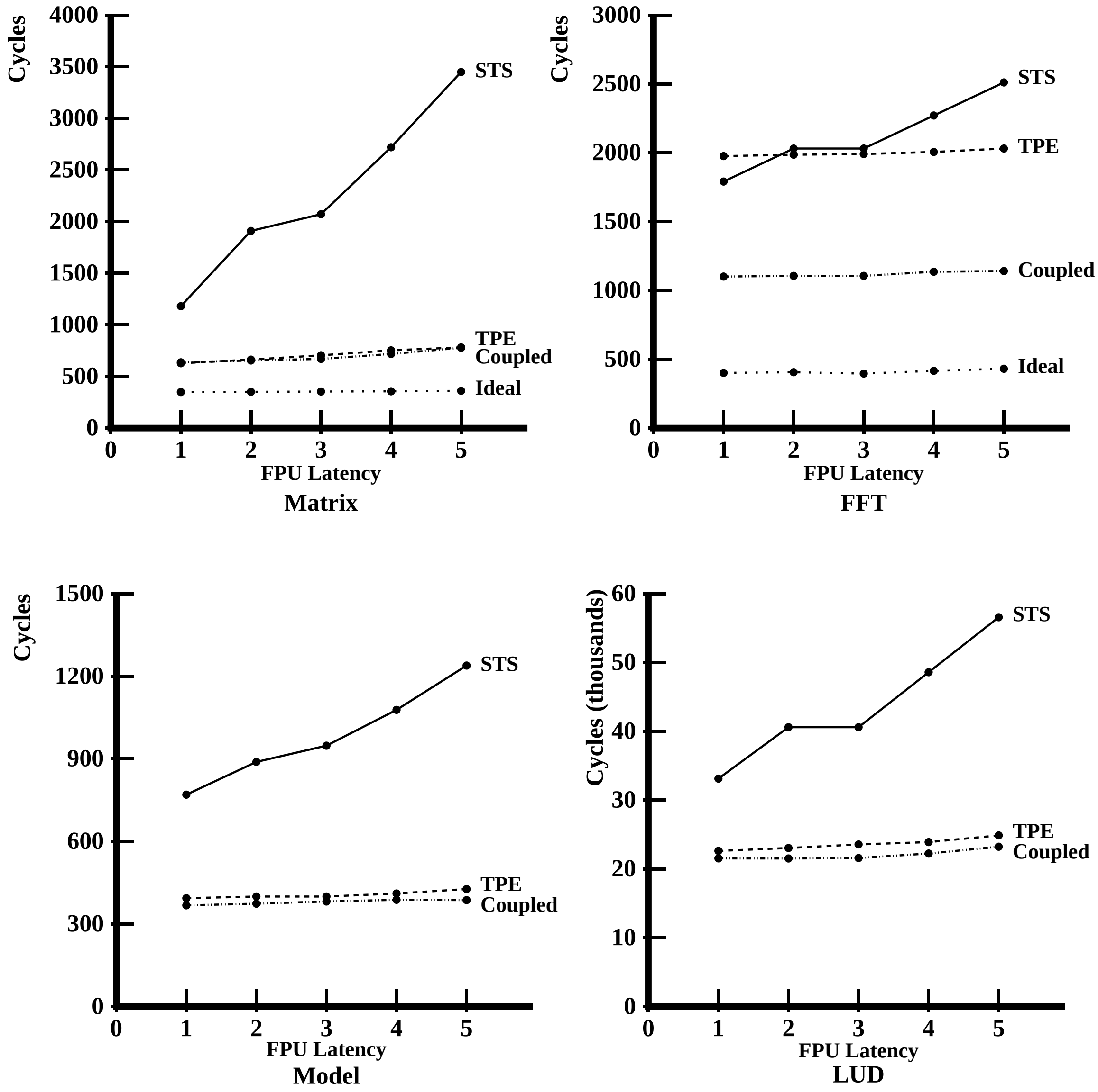

Figure 6.5: Cycle counts when floating point unit latency is varied.

Coupled fare much better than STS, requiring only 11% and 10% more cycles respectively.

Ideal mode performs well because its loops are unrolled, providing more operations to the compiler's scheduler to place in gaps created by longer latency floating point units. In contrast, the basic block boundaries of STS limit the number of operations that can be scheduled together. Unlike dynamically determined delays such as memory references, statically known function unit latencies can be used during compilation. Trace scheduling would pack instructions in STS more efficiently since operations from multiple basic blocks would be available.

The results from TPE and Coupled show that a similar effect can be attained using multiple threads. Instead of packing the instructions at compile time, the gaps in the instruction stream created by function unit latencies are filled by operations from other threads. Trace scheduling would not improve the performance of the multithreaded modes as dramatically as for STS. However, advanced compiler technology would certainly help TPE and Coupled as well.

6.7 Restricting Communication

Since a thread's register set is partitioned, data may need to be transferred between clusters. When two independent operations executing simultaneously on different clusters produce results needed by a subsequent operation, at least one of the results must be transferred. Thus data movement results from the compiler trying to exploit the maximum instruction-level parallelism. Another source of intercluster communication are results that must be used by multiple subsequent operations. One example is an eliminated common subexpression, such as redundant array index calculations. These values might be distributed to multiple clusters.

Since the number of buses and register input ports required to support fully connected function units is prohibitively expensive, some compromises must be made. Communication can be restricted between function units such that hardware cost is reduced without significantly affecting performance. The five different communication configurations that were simulated are described below:

1. Full: The function units are fully connected with no restrictions on the number of buses or register file write ports.

2. Tri-Port: Each register file has three write ports. One port is used locally within a cluster by those units sharing the register file. The other two ports have their own buses and may be used by a local or a remote function unit.

3. Dual-Port: Each register file has two write ports. This is similar to Tri-Port, with only one global register port.

4. Single-Port: Each register file has a single write port with its own bus. Any function unit can use the port without interfering with writes to other register files.

5. Single-Bus: Each register file has two write ports. One port is for use within a cluster, while the other port is is connected a bus shared with all of the other register files. Arbitration is performed to decide which function unit may use the bus on a given cycle.

Figure 6.6 demonstrates how Processor Coupled performance is affected by restricting the amount of communication between function units. Since each arithmetic cluster has three function units, if the local register file has fewer than three ports, conflicts within a cluster can occur. These internal conflicts account for some of the additional cycles for the Dual-Port, Single-Port, and Single-Bus networks. In most cases, the Single-Bus network performs better than Single-Port because of the dedicated local register port. Intra-cluster conflicts are highlighted by the statically scheduled SEQ and STS modes. For example, **Matrix** in SEQ mode requires many more cycles for Dual-Port than for Tri-Port. Dual-Port and Single-Bus have effectively the same network if only one cluster is used, but using Single-Port requires 50% more cycles. A better compiler could schedule around some of these conflicts to achieve better utilization, but other conflicts due to statically unknown memory latencies could still occur.

In TPE mode, since each thread is assigned to a different cluster, local communication is more important than intercluster communication. Thus, TPE requires the most cycles when using the Single-Port network, while the Full, Tri-Port, and Dual-Port networks execute

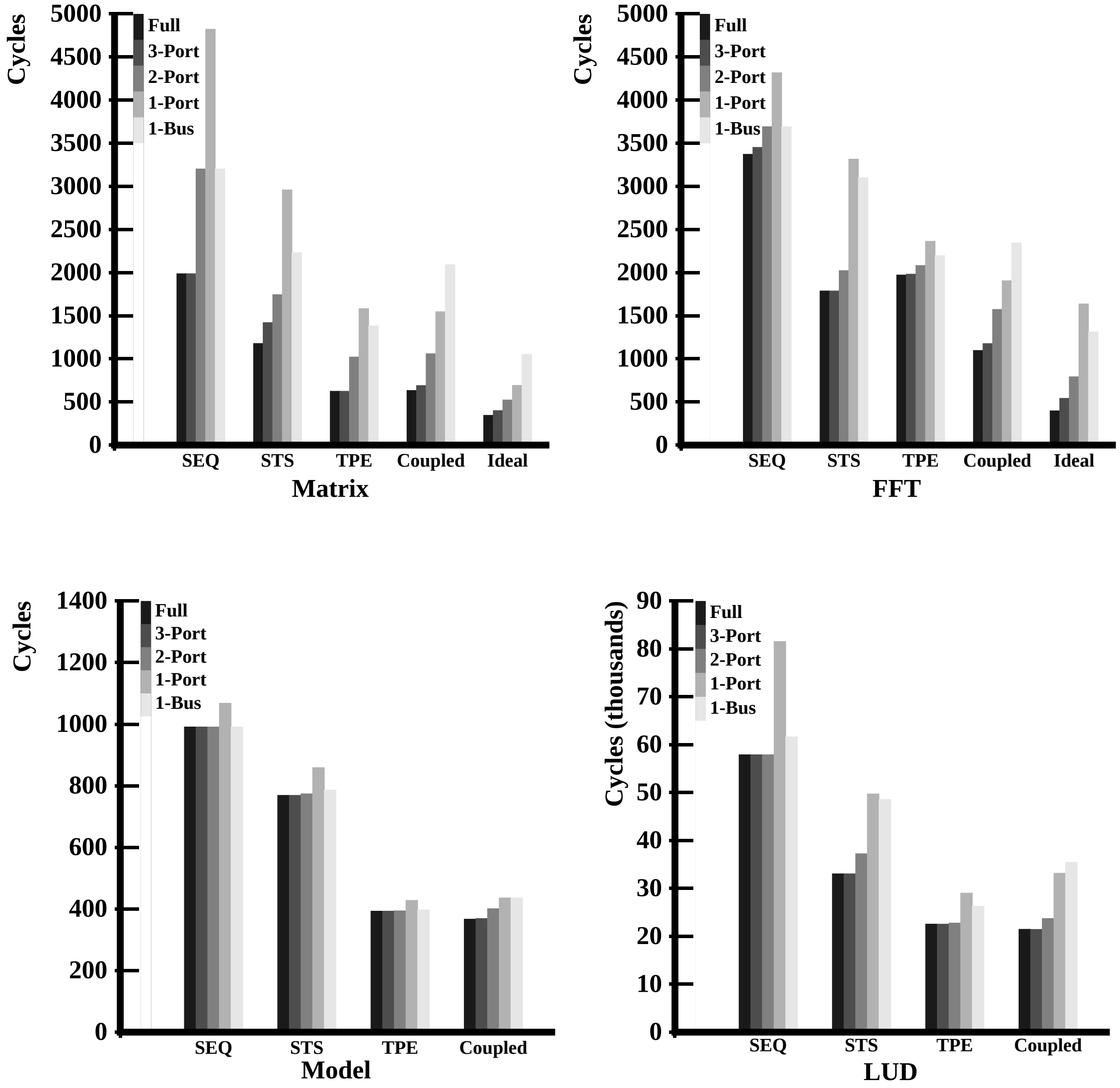

Figure 6.6: Cycle counts for restricted communication schemes of Processor Coupling. Cycle count increases dramatically when single buses and ports are used, but the Tri-Port scheme is nearly as effective as the fully connected configuration.

a comparable number of cycles in all benchmarks except **Matrix**. With Dual-Port, the unrolled inner loop and multiple threads per cluster in **Matrix** saturate the local register file ports, resulting in 60% more cycles than Full.

As expected, the number of cycles increases for Coupled mode when function units must contend for buses and register ports. **Matrix**, **FFT**, and **LUD**, have high integer unit utilization because they calculate many common array indices, and are sharply affected when using a Single-Port or Single-Bus network in Coupled mode. **Model** exhibits less instruction-level parallelism, has low unit utilization, and is hardly affected by changing communication strategies. Since Coupled mode takes advantage of as many clusters as possible, fast communication between clusters is necessary. Thus Single-Port with many global buses is better than Single-Bus.

Any restricted communication scheme trades chip area for increased cycle count. For Coupled mode, Tri-Port has the best performance of the restricted configurations examined, requiring an average of only 4% more cycles than the fully connected configuration. Tri-Port can be implemented using only 2 global buses per cluster. The number of buses to implement a fully connected scheme, on the other hand, is proportional to the number of function units times the number of clusters. Furthermore, the completely connected configuration will require additional register ports. In a four cluster system the interconnection and register file area for Tri-Port is 28% that of complete interconnection. Dual-Port might be an attractive alternative as well. This two write port configuration uses 32% more cycles than complete interconnection, but requires only 21% of the area.

6.8 Number and Mix of Function Units

To determine the proper ratio between different types of units, Processor Coupled machine configurations were simulated with up to four IUs and four FPUs, while keeping the number of memory units constant at four. Simulations of these benchmarks show that a single branch unit is sufficient. Figure 6.7 displays the cycle counts for all the benchmarks as a function of the number of IUs and FPUs. The number of FPUs and IUs are on the X and Y axes. Cycle count is displayed on the Z axis.

The function unit requirements depend greatly upon the application. For **Matrix**, cycle

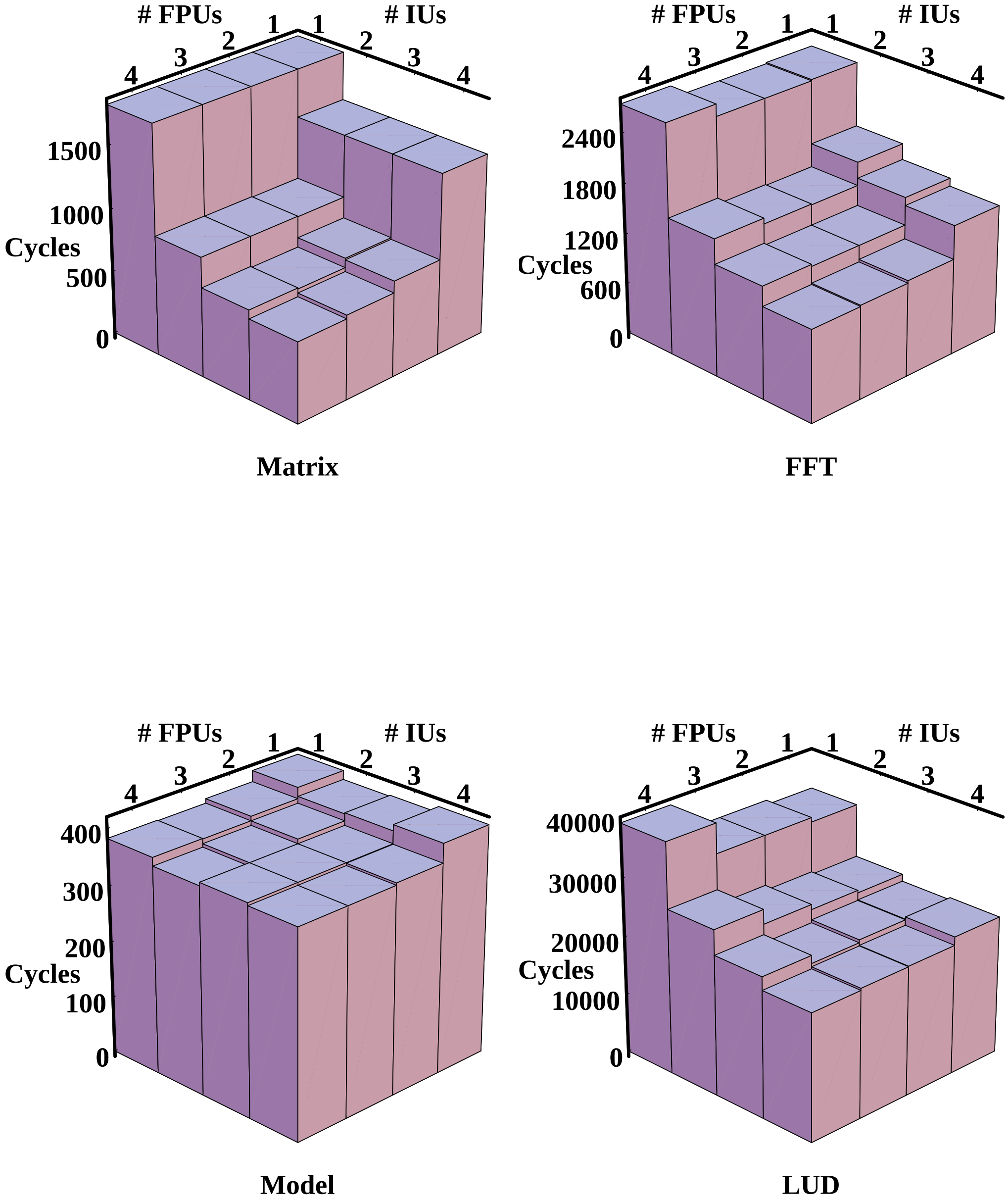

Figure 6.7: Cycle count as a function of number of Floating Point Units (FPUs) and Integer Units (IUs). For a fixed number of function units, cycle count is minimized when equal numbers of IUs and FPUs are used.

count is highest when only one IU and one FPU are used, and decreases when more units are added. For two or more IUs, if the number of IUs is held constant and the number of FPUs is increased, the cycle count drops. The same holds true if the number of FPUs is constant and the number of IUs is varied. One FPU will saturate a single IU, but two IUs are needed to saturate a single FPU. Thus, even though each benchmark consists primarily of floating point operations, integer units can also be a bottleneck since they are used for synchronization and loop control. With a fixed number of function units, cycle count is minimized when the number of FPUs and IUs are equal.

The results for **FFT** are similar to those of **Matrix**. However, the cycle count for four FPUs and one IU is greater than that of three FPUs and one IU. This is due to additional IU operations required to move floating point array indices to memory units in remote clusters. Like **Matrix**, one FPU will saturate a single IU, but each additional IU improves performance. **LUD** shows much the same behavior as **FFT** with cycle count increasing as FPUs are added.

Model exhibits much less instruction-level parallelism and does not benefit as greatly as the other benchmarks from additional function units. Cycle count is still minimized when four IUs and four FPUs are used.

For these benchmarks, the incremental benefit in cycles decreases as more function units are added. The results indicate that the performance of a Processor Coupled node does not increase significantly for the sample benchmarks when more than four IUs and four FPUs are used.

6.9 Methods of Expressing Parallel Loops

This section shows how benchmark performance changes when different methods are used to express parallel loops. **Model** uses a parallel loop to evaluate each device model concurrently. **LUD** has a parallel loop to update each target row once a source row has been selected. Neither **Model** nor **LUD** can use `forall` since the number of threads needed depends upon the input data. The three methods examined here are:

- **Fork**: The compiler creates a `for` loop containing one version of the loop body. At runtime a fork operation creates a new thread on each loop iteration. Since each thread executes the same code, the function unit access patterns will be identical.

- **Iter**: The `forall-iterate` construct is used to tell the compiler to generate different versions of the loop body, each with a different function unit access pattern. On each iteration of the loop, a new thread is created, but the version of the code depends on the loop iteration number. This attempts to balance the work across the clusters.

- **Auto**: The compiler generates code for four threads which will be created when the program begins. Each thread retrieves a task from the head of a common work queue and executes it. When a thread has completed its task, it gets another job from the queue. Since each thread has a different function unit access pattern, the work will be balanced across the clusters. In **Model** the queue is a counter holding the index of the next device to evaluate. Each thread atomically increments the counter and evaluates the appropriate device. Threads exit when there are no more devices to evaluate. **LUD** uses a counter to select all of the target rows to update.

In the previous experiments, **Iter** was used as the threaded version of the **Model** and **LUD** benchmarks.

6.9.1 Base Results

Figure 6.8 shows the cycle counts for **Model** and **LUD** using the above parallel loop techniques. In Coupled mode, **Iter** is slightly worse than **Fork** on both benchmarks due to the additional time required to choose the version of the code to execute. The selection overhead costs slightly more than the advantage gained from load balancing. **Auto** performs better than both since the overhead to create threads has been eliminated and the workload is well balanced.

In TPE mode, **Fork** requires between 1.5 and 3 times as many cycles as **Iter** and **Auto**. Since **Fork** has no load balancing, all of the operations execute on a single cluster. The performance of **Iter** is similar to **Auto**, but because of the code selection overhead, **Iter** is slightly worse.

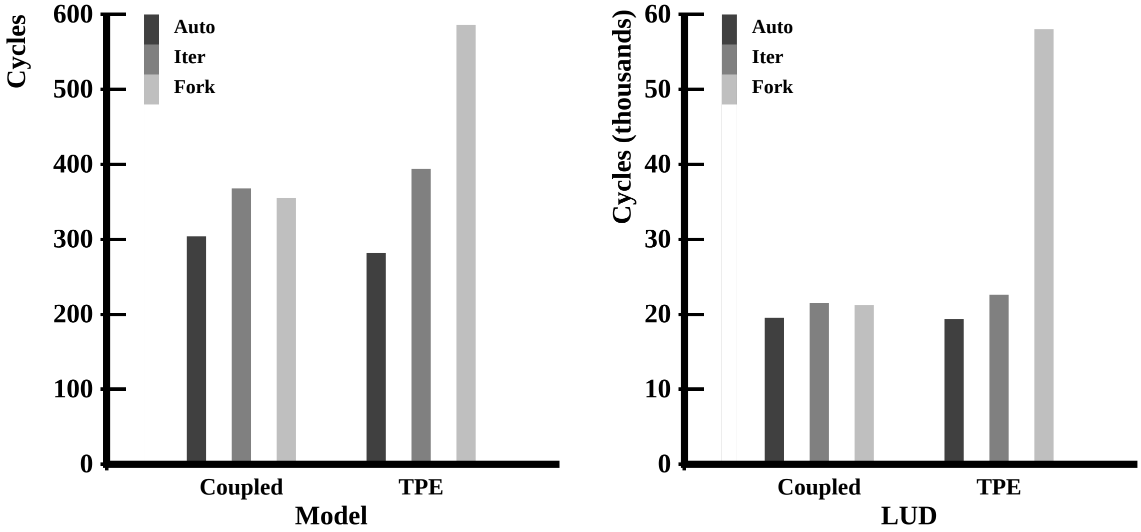

Figure 6.8: Cycle counts for **Model** and **LUD** using different parallel loops.

Aside from **Fork**, Coupled and TPE performance is quite similar for both **Iter** and **Auto**. On **Iter**, Coupled has a slight advantage because the sequential work of the top level thread can be distributed over multiple units. However for **Auto**, TPE has the advantage, since each thread has its own cluster and does not interfere with threads on other clusters.

6.9.2 Memory

Figures 6.9 and 6.10 shows the effect of memory latencies when using the different parallel loops. For **LUD** the cycles required for **Auto** increases dramatically. The setup code, including many memory references, is duplicated in each of the four threads; since cache behavior does not model locality, more memory references cause more cache misses to occur. **Iter** and **Fork** execute a similar number of memory operations and their cycle counts track when memory latency is increased. **Model** exhibits similar behavior.

6.9.3 Communication

Figures 6.11 and 6.12 show the effect of restricting communication between clusters for the different forms of parallel loops. The behavior exhibited here is similar to that seen in

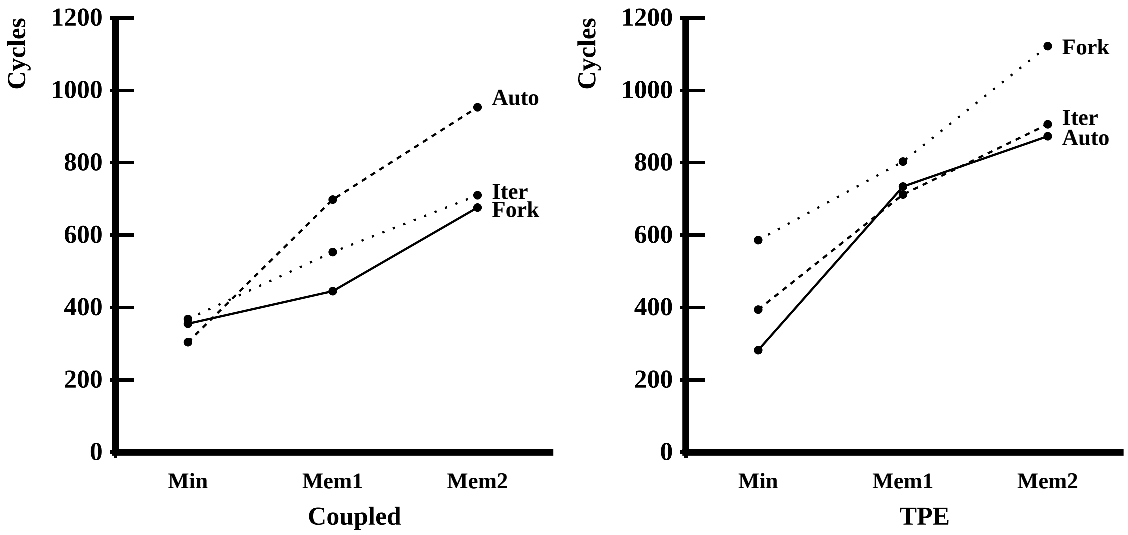

Figure 6.9: Effect of memory latency on **Model** using different parallel loops.

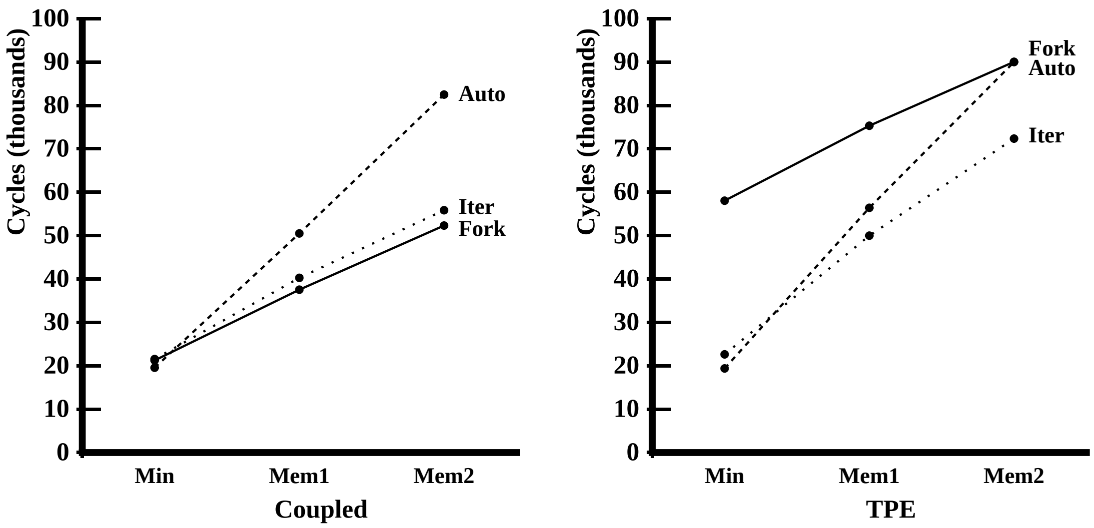

Figure 6.10: Effect of memory latency on **LUD** using different parallel loops.

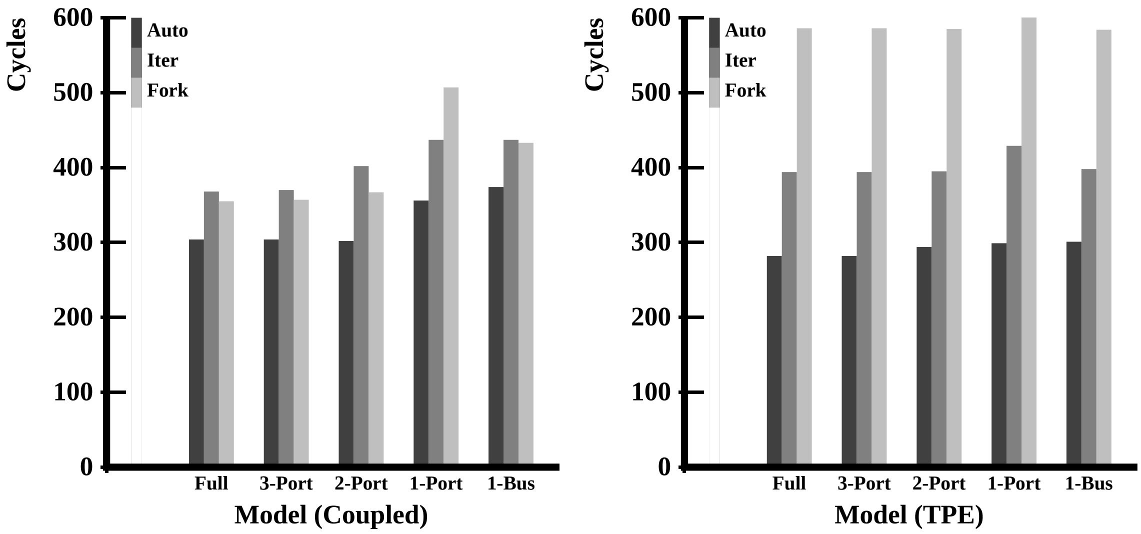

Figure 6.11: Restricting communication on **Model** using different parallel loops.

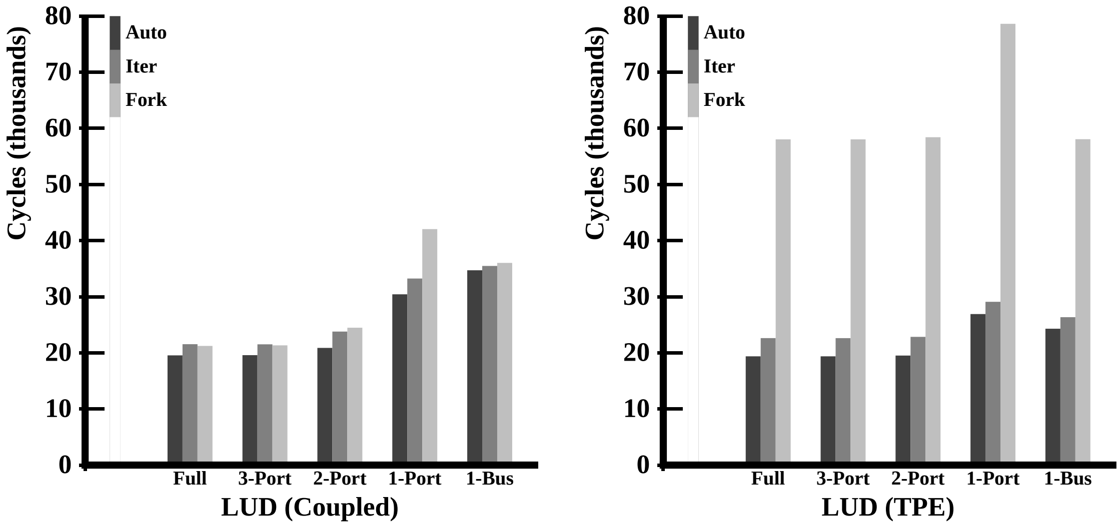

Figure 6.12: Restricting communication on **LUD** using different parallel loops.

Section 6.7. In Coupled mode, the cycle counts of both **Auto** and **Iter** are similarly affected by all communication networks; neither has a significant advantage. **Fork** is most affected by communication restrictions, particularly by Single-Port. Because load balancing is not performed, more operations execute within a single cluster and saturate the local register write ports while bandwidth in other clusters goes unused. TPE exhibits similar behavior. **Iter** and **Auto** have the same relative performance, while **Fork** is hardest hit when using the Single-Port network.

6.10 Data Movement Strategy

This section shows the effect on cycle count from changing the way data is transferred between clusters. Within a thread, data is moved by specifying a target register in another cluster. Using multiple destination registers to deliver a result to two different clusters eliminates the move operation required if only one destination is allowed. When all of the destination specifiers in the operation generating the value have been filled, the compiler must use an explicit move operation. If dedicated data movement units are not included in the hardware model, the compiler will schedule move operations on arithmetic units.

The experiments here vary the number of destination register specifiers for configurations with and without data movement units. The remaining machine resources are identical to the baseline configuration used in previous sections. The clusters are fully connected and have enough register ports and buses so that function units are not stalled. Figure 6.13 shows the cycle count when the number of destination specifiers in the operation is varied from 1 to 4.

For both configurations, the most gain is achieved when going from 1 to 2 destination specifiers. In fact, with more than 2 targets, no move operations are required at all. The cycle count for **LUD** and **FFT** decreases the most, largely because they have the most common subexpressions, such as array indices and multiply used variables.

In **Matrix**, **Model**, and **FFT**, when 4 destination registers are used, the cycle count actually increases even though the number of move operations is reduced. Since the scheduling phase of the compiler has a cost model for moving operands, it may decide to schedule an operation in the cluster where the data is already located, rather than transfer the data

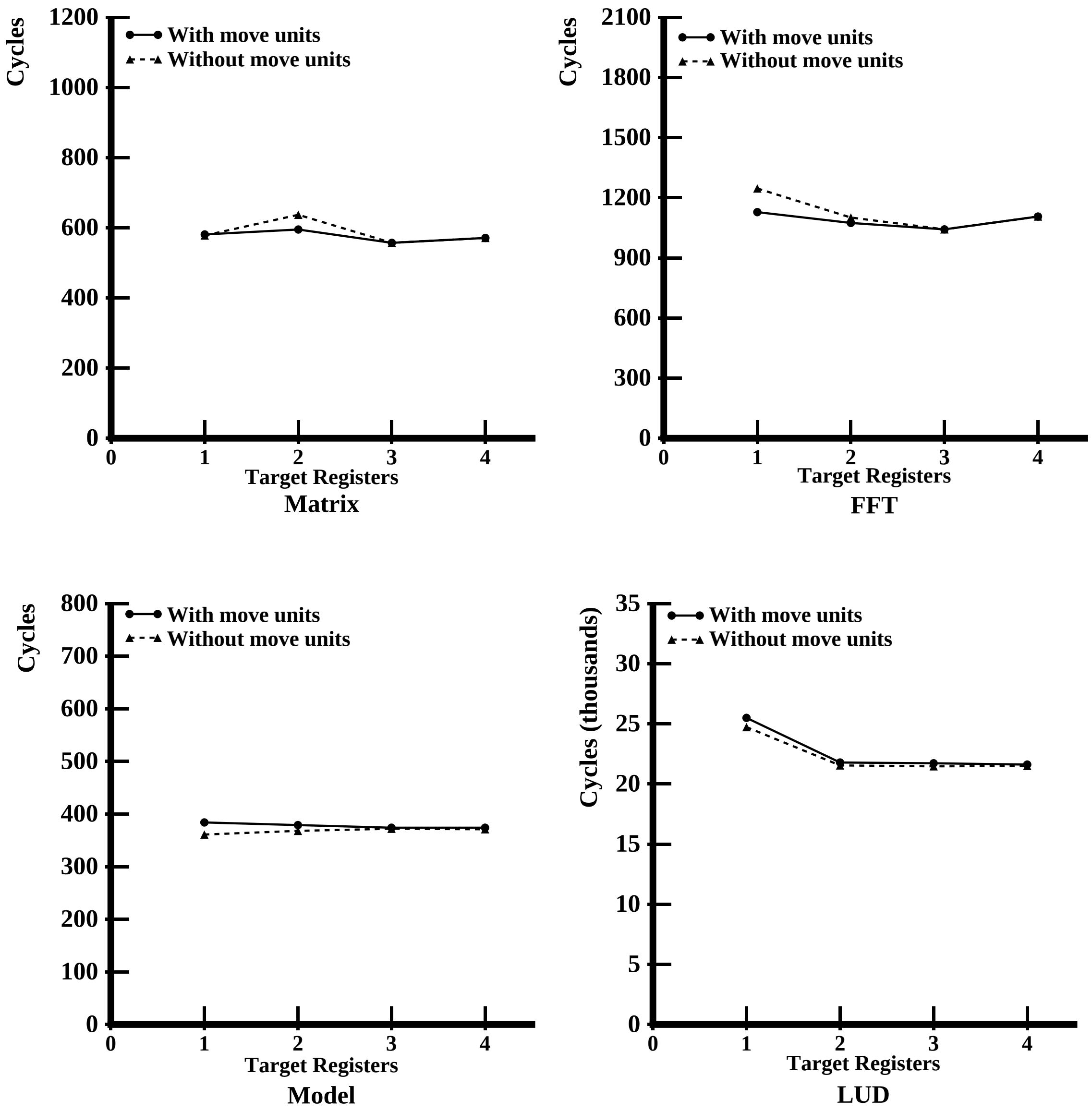

Figure 6.13: The effect of varying the number of target registers specified in an operation in Coupled mode both with and without dedicated move units.

to another cluster. However, additional target registers allow the scheduler to spread the data and operations over all of the function units; this may increase interference between threads. The runtime behavior of this experiment is extremely sensitive to the assumptions made in the compiler's scheduler. Additionally, although it is not modeled here, spreading the data around increases contention for shared Cluster Interconnection Network resources. Optimizing dynamic behavior using compile time analysis, especially when multiple threads are permitted, is very much an open research question.

Only **FFT** with one target and **Matrix** with two targets display an advantage of using dedicated move units. From these experiments, the best configuration has two destination register specifiers, and does not need dedicated move units.

6.11 Summary

In this chapter, four simple benchmarks, **Matrix**, **FFT**, **Model**, and **LUD**, were run on a variety of different machine models. On these programs, a Processor Coupled node executes 60% fewer cycles, and achieves almost twice the utilization of a statically scheduled processor without coupling. Processor Coupling is tolerant of variable memory and execution unit latencies. In the experiments with variable memory latency, a Processor Coupled node executes 75% fewer cycles than a statically scheduled processor. Processor Coupling is also more tolerant of function unit latencies, executing only 10% more cycles when the floating point unit latency is increased from 1 to 5 cycles. The statically scheduled processor requires 90% more cycles for the same change in floating point unit latency.

On threaded code TPE and Coupled modes achieve nearly the same performance. However, on sequential sections of code, Coupled is able to use all of the function units while TPE is restricted to only those function units within a single cluster. In sequential sections the TPE model requires on average 2.9 times as many cycles as Coupled. On **FFT** which has a significant sequential section, TPE executes 79% more cycles than Coupled.

The simulations show that the Cluster Interconnection Network greatly affects cycle count. A scheme with only three write ports in each register file achieves performance approaching that of a fully connected network. One port is devoted to local register writes, and the remaining ports are connected to buses that can be driven by function units in other clusters. This configuration gives nearly the same performance as complete interconnection (only 4% more cycles) for less cost (28% of the interconnect area for a four cluster machine). A two write port configuration executes 32% more cycles than complete interconnection, but requires only 21% of the area. Performance also depends on the right balance of function unit types. The simulations suggest a configuration with four floating point units, four integer units, and one branch unit.

The experiments with parallel loops indicate that balancing workload across the clusters without incurring runtime overhead is important. Autoscheduling accomplishes these goals but seems to pay a penalty when memory latency is increased, due to additional memory references. However the actual time to create threads would increase the cycle count for **Iter**. A simulation system that models locality and the cost of creating threads would provide better insight. Finally, the results from the data movement experiments indicate that using arithmetic units to execute move operations and allowing an operation to specify two destination registers is sufficient. However, the configuration with only one destination specifier has performance within 15% of that with four destinations.

Chapter 7

Implementation Issues

This chapter describes some of the issues involved in building a Processor Coupled node. Section 7.1 introduces the Processor Coupling pipeline, including the additional synchronization stage required. Section 7.2 discusses further implementation issues such as synchronization and communication between function units, the memory system, and multi-threading. Finally, Section 7.3 discusses the chip area required for the different components of the processor and explores some of the tradeoffs in register file design and intercluster communication. This chapter is meant to demonstrate the feasibility of a Processor Coupled node but does not present all of the details necessary to build one.

7.1 Function Unit Pipeline Stages

A block diagram of a function unit is shown in Figure 7.1. The function unit has a five-stage pipeline consisting of operation fetch (OF), scoreboard check (SC), register read (RR), execution (EX), and write back (WB) stages. Each active thread has an entry in the operation pointer set, in the operation prefetch buffer, and in the operation buffer. The operation buffer shown has space for operations from four active threads. To keep the pipeline full, the scoreboard is updated as an operation is sent from the SC stage to the RR stage. The value of the result is written to the register file during the WB stage. The paths to the register file and scoreboard from other clusters are not shown. Memory units and branch units have additional interfaces and logic but use a similar five stage pipeline.

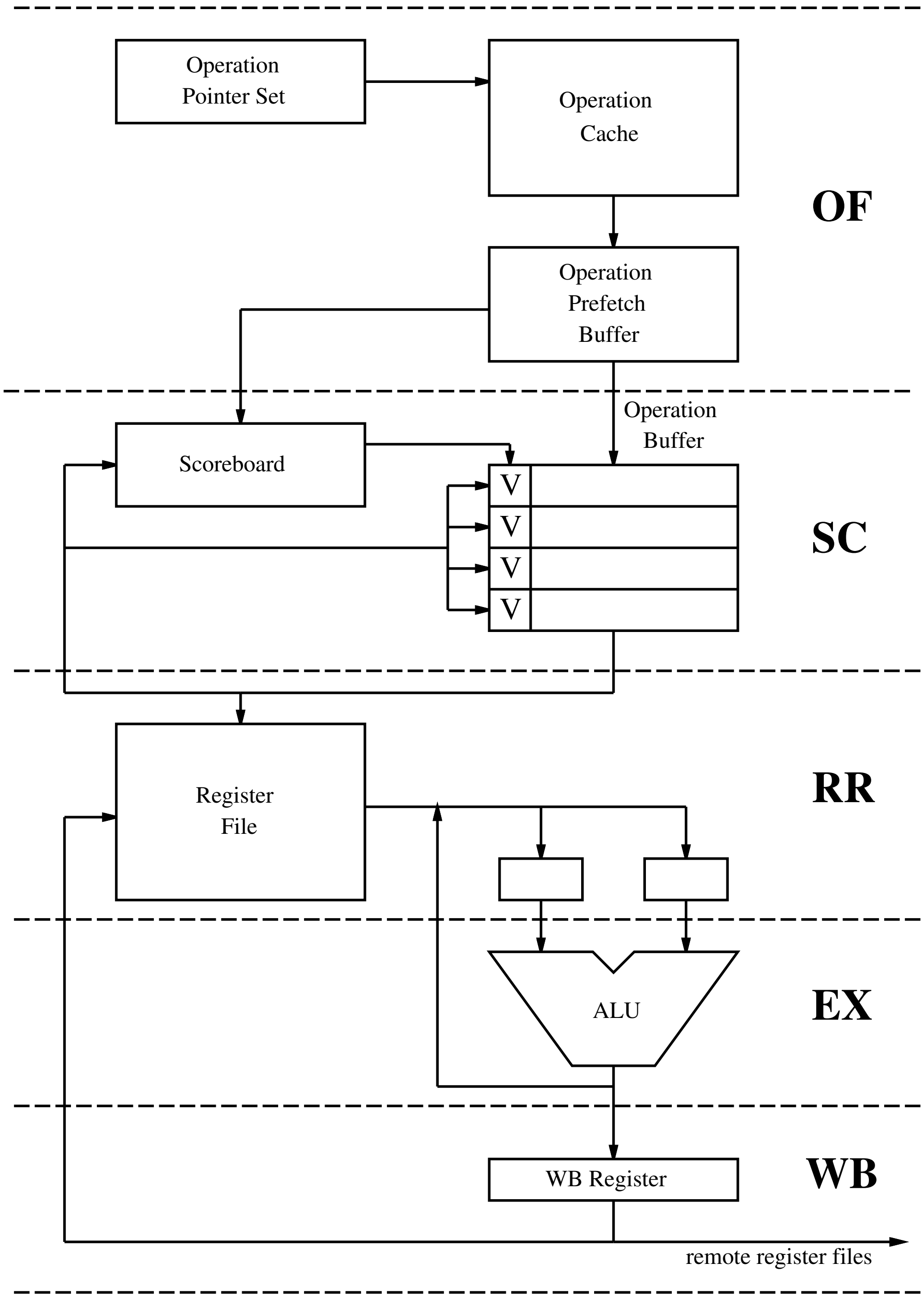

Figure 7.1: A sample arithmetic unit with a single EX stage. OF fetches an operation from the Operation Cache, SC checks the scoreboard to determine which of the source registers are ready, RR reads the register file, EX executes the operation, and WB writes back the result.

7.1.1 Operation Fetch

During the operation fetch (OF) stage, an active thread addresses the operation cache and places an operation into the operation prefetch buffer. The operation prefetch buffer holds fetched operations that are waiting to read the scoreboard and arrive in the operation buffer. The OF fetch logic determines which active thread accesses the operation cache by examining the operation that issues to the execution pipeline. Thus, the thread that issues an operation is allowed to fetch an operation. Linking the operation issue determined in the SC stage with the operation fetch performed in the OF stage permits a single thread to keep its pipeline full and issue an operation each cycle.

7.1.2 Scoreboard Check

The fetched operation accesses the scoreboard during the scoreboard check (SC) stage. The scoreboard determines which of the source registers required by the operation are ready. The operation, along with the register status obtained from the scoreboard, is deposited in the operation buffer. The operation buffer can hold one pending operation from each active thread.

The operation buffer maintains the status for each pending operation to determine when that operation is ready to issue. Its function is similar to that of a reservation station [Tom67], except that operations are buffered before reading their registers instead of after. An operation is enabled to issue when all of its source registers are ready and when all operations from previous instructions in that thread have issued. Operation issue is defined as sending a valid operation to the RR stage. A pending operation's status is updated as source registers become available and as operations from previous instructions complete. This update is performed in place and does not require access to the scoreboard. An operation that finds all of its registers ready when it accesses the scoreboard may move directly from the SC stage to the RR stage without entering the operation buffer.

Figure 7.2 shows the operations from two different threads (T1 and T2) in the pipeline of a function unit. The status of T1's register R4 is shown at the bottom of diagram; the other source registers are valid, and their status is not shown. T1's operation is fetched during cycle 1, and during cycle 2 it checks the scoreboard, finds R4 invalid, and waits in

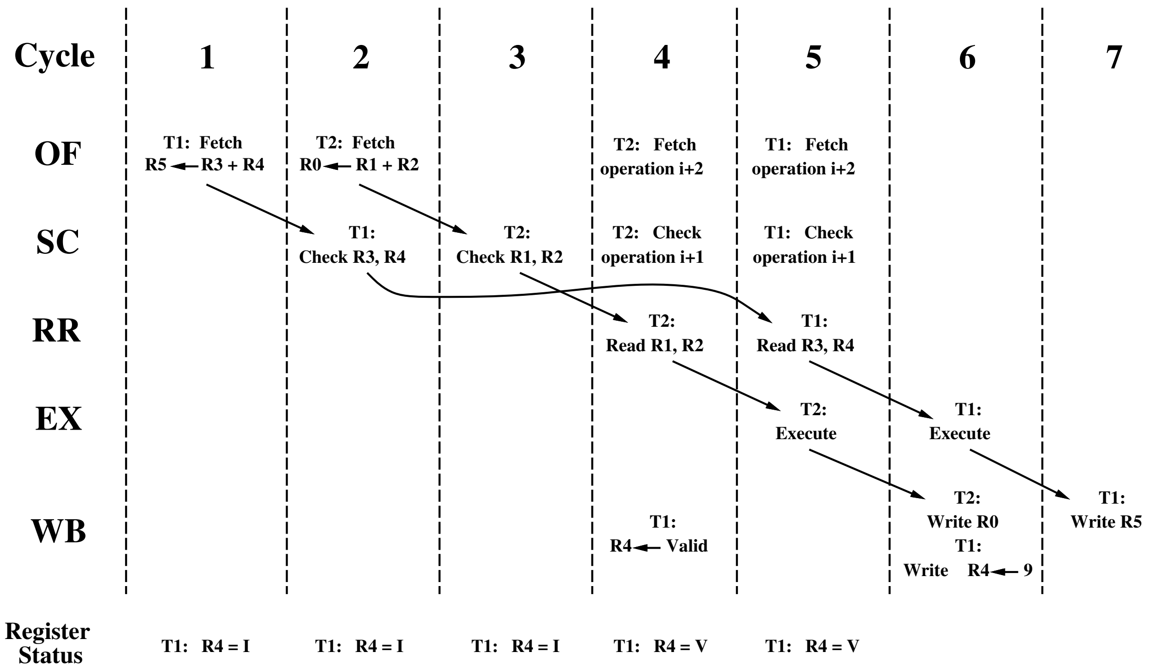

Figure 7.2: The procession through the pipeline of operations from two different threads (T1 and T2). T1's operation is fetched during cycle 1, and during cycle 2 it checks the scoreboard and finds R4 invalid. T2's operation is fetched in cycle 2, checks the scoreboard during the cycle 3, and since both source registers are valid, issues to the RR stage during cycle 4. T1's register R4 becomes valid at the end of cycle 4 and it follows T2's operation into the RR stage in cycle 5.

---

the operation buffer. T2's operation is fetched in cycle 2, checks the scoreboard during cycle 3, and since both source registers are valid, issues to the RR stage in cycle 4. A T1 operation from a remote cluster updates the scoreboard during cycle 4, allowing the T1 operation waiting in the operation buffer to issue to the RR stage. The value 9 is forwarded to the start of the EX stage in cycle 6. When operation i is issued to the RR stage, operation $i+1$ in the prefetch buffer enters the SC stage and operation $i+2$ is fetched into the prefetch buffer. Thus on cycle 4 when T2's operation reads the registers, the two subsequent operations enter the SC and OF stages.

The results of the T2 and T1 operations are written back to the register file during cycles 6 and 7 respectively. The remote T1 operation writes the value 9 into the register file during cycle 6 as well, using one of the additional register write ports. The arrows between pipeline stages show the progress of each thread. Because of the scoreboard synchronization in the SC stage, thread 2 is able to pass thread 1 in the pipeline during cycle 3.

7.1.3 Register Read

During each cycle, the operation buffer selects an enabled operation and forwards it to the RR stage where it reads its source registers. These registers are guaranteed to be available since only operations with all dependencies satisfied are issued to the RR stage. Because the presence bits used for synchronization are kept in the scoreboard and are accessed during the SC stage, the RR stage is the only portion of the pipeline that uses the register read ports. Values bypassed from execution stages will join the operation at the beginning of the first EX stage.

7.1.4 Execution

The operation along with its arguments proceeds to the execution (EX) stage(s). All operations, including arithmetic operations, loads, stores, and branches, are executed during EX stages. Depending on the type of operation performed, more than one cycle may be required for execution. For example, a floating point multiply may require 5 cycles to execute, while an integer addition may only need 1. Some operations, such as memory references, take a variable number of cycles to complete. Since the scoreboard manages data dependencies, out of order completion of operations is permitted.

7.1.5 Write Back

The results of an operation are written to the destination register file in the write back (WB) stage. A destination register may reside in the local register file or in a remote register file. Writes to the local register file can use the local bypass paths to skip the WB stage, allowing results to be used on the next cycle. Remote writes are bypassed to the destination clusters but arrive one cycle later due to the communication delay. Scoreboard update takes place before the operation completes so that the dependent operation can issue and use the bypassed data. Allocating the communication resources for transmitting data between function units during the WB stage is discussed in Section 7.2.4.

7.2 Operational Issues

7.2.1 Scoreboard Update

Two cycles before an operation is to complete, the destination register file's scoreboard is updated to indicate that the register is available. This allows an operation to issue in time to use a result that is bypassed directly from the execution unit. Figure 7.3 shows an operation pipeline with a one stage execution unit. The first operation updates the scoreboard as soon as it issues to the RR stage at the beginning of cycle 3. This permits the subsequent operation to issue in cycle 4, and use the result that arrives via the bypass path shown from one EX stage to the next.

The scoreboard requires the same number of read ports and write ports as the register file. The read ports are used during the SC stage to check the status of operation source registers while the write ports are used two cycles before WB to update the scoreboard to reflect subsequent register writes. Between different function units, arbitration required for access to register file write ports is performed at the same time as the scoreboard update. The data from a load operation is returned to the destination cluster where it is placed directly in the register file.

7.2.2 Comparison to Conventional Pipeline

The Processor Coupled function unit adds a scoreboard check stage to the conventional four-stage execution pipeline. In a traditional pipeline, instructions check the availability of their operands in parallel with the register read. Speculative register reads before dependencies are satisfied cause the pipeline to stall whenever a source register is not available. Because the register read ports have been consumed by the stalled instruction, no other instruction is available to proceed to the execution unit on the next cycle. Instead a bubble must be inserted into the pipeline and resources are wasted.

In a Processor Coupled pipeline, dependencies are examined in the SC stage without using the register read ports. Only operations that are enabled by having all dependencies

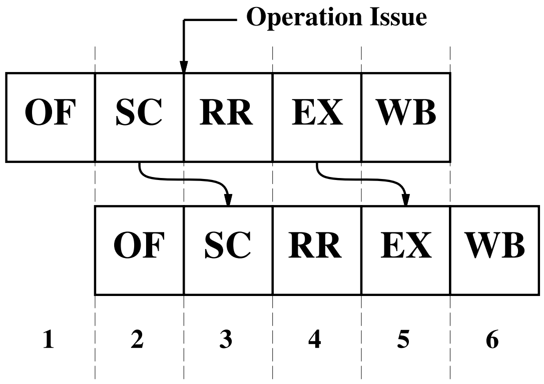

Figure 7.3: A five stage pipeline corresponding to the function unit in Figure 7.1. The first operation updates the scoreboard as it issues to the RR stage, two cycles before the result is ready. The next operation can issue immediately and use the bypassed result from the EX stage.

---

satisfied are issued to the RR stage. This early check permits an operation from another thread to be issued from the operation buffer if the operation just accessing the scoreboard is not ready. The scoreboard check adds a stage to the pipeline and thus increases the mispredicted branch penalty by one cycle. It does not increase arithmetic latency since the scoreboard can be updated early to permit immediate use of bypassed arithmetic results.

7.2.3 Thread Selection

A Processor Coupled node interleaves a set of active threads. Every cycle, each function unit's SC stage selects an enabled operation (if one exists) from the operation prefetch buffer or the operation buffer for issue to the RR stage. If more than one operation is enabled, selection is made using a fixed priority.

7.2.4 Intercluster Communication

An operation must read its source registers from the local register file but may write its result to a register in a different cluster. An intercluster interconnection network is provided to support this communication. Several alternative structures for this network were evaluated in Chapter 6. For these networks, a single cycle is required to transport a result to a remote

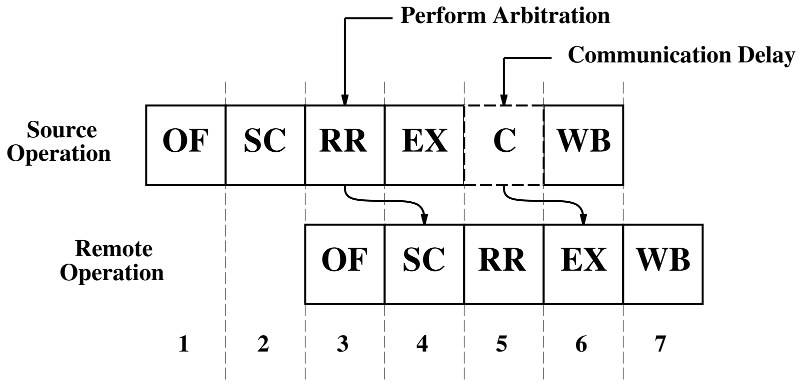

Figure 7.4: Bypassing data from the EX stage of a source unit to the EX stage of a remote unit. Arbitration for communication resources is performed during the RR stage. The single cycle communication latency delays the result from arriving at the destination until cycle 6.

---

cluster. In a real system, a result produced by the WB stage in cycle i can be used by the execution unit in the local cluster during cycle i, but will not be available to a remote cluster until cycle $i + 1$.

Arbitration for shared communication resources is performed at the time of scoreboard update, two cycles before the result is available. An operation that wins the arbitration updates the scoreboard in the remote cluster immediately and uses the data paths granted by the arbitration two cycles later to provide the result. An operation that loses the arbitration stalls its pipeline and retries on subsequent cycles until the data paths are granted. The networks discussed in Chapter 6 are designed so that all resources (paths and register ports) are granted in a single arbitration cycle; multiple arbitrations which can potentially lead to deadlock are not required.

Figure 7.4 shows the bypassing of an operand from a source function unit to a remote unit. The compiler has statically scheduled the source operation to begin on cycle 1 and the remote operation to begin on cycle 3. The source operation issues to the RR stage at the beginning of cycle 3. Since the data will become available during cycle 5, arbitration for communication resources is performed and the remote unit's scoreboard is updated during cycle 3 as well. The remote operation can issue during cycle 5. By the time the registers

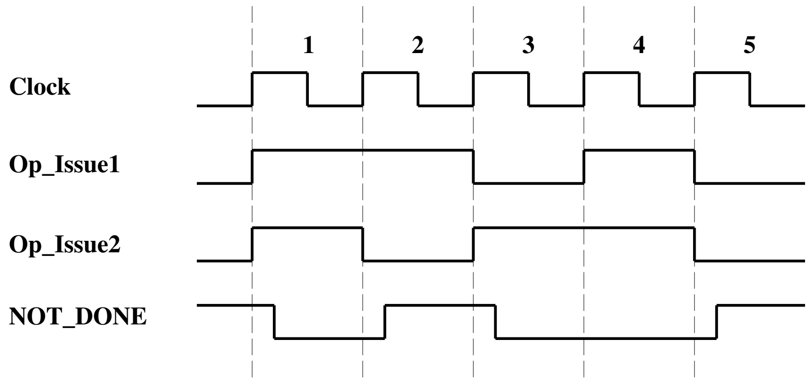

Figure 7.5: The NOT_DONE signal indicates when all of the operations scheduled in an instruction have been issued. The **op_issue** waveforms are asserted when an operation issues from the SC stage to the RR stage. NOT_DONE remains asserted until both operation 1 and operation 2 have issued.

---

have been read and the EX stage begins, the data from the EX stage of the source unit has arrived.

7.2.5 Intercluster Synchronization

To provide in-order execution of operations from a given thread, a wired-or NOT_DONE line is provided for each active thread. If a function unit holds an unissued operation from a thread's previous instruction, the unit will assert the corresponding NOT_DONE line. When all non-null operations in a particular thread's previous instruction have issued, this line is deasserted. As long as the line remains deasserted, each function unit is capable of issuing an operation from that thread every cycle. The OF and SC stages fill the operation buffer with operations from each active thread independent of the state of the thread's NOT_DONE line. If there are holes in a function unit's operation stream due to implicit *nops*, the OF logic will fetch operations ahead of where the thread is currently executing on other units. The function unit can count the number of NOT_DONE signals seen to determine when the rest of the units have caught up. Each unit will then be able to issue an operation for a thread as soon as the NOT_DONE line is deasserted.

Figure 7.5 shows a timing diagram for two function units' operation issue events and the corresponding NOT_DONE signal. Both units issue operations from the SC to the RR stage in cycle 1, enabling the next operations to be issued in cycle two. Although the operation in unit 1 issues during cycle 2, the corresponding operation in unit 2 must wait until cycle 3 due to a dependency that is not shown. The NOT_DONE line remains high until operation 2 has issued during cycle 3. The next operations waiting in the operation buffers are enabled and can be issued during cycle 4.

7.2.6 Memory System

Each memory unit sees the memory system as a request port and a response port. A memory unit sends read and write requests to the request port. Each request consists of an address, as well as data (for writes) or a destination specifier (for reads). Results of read operations are directed to the destination cluster's response port. Each response updates the scoreboard and stores its data into the specified register.

The details of the memory system implementation are beyond the scope of this thesis. An interleaved cache will be required on-chip to satisfy the bandwidth demands of the arithmetic units. This on-chip cache will be backed up by a larger off-chip memory and by the global machine memory accessed over an inter-node interconnection network.

7.3 Single Chip Feasibility

In 1995, CMOS process technology will allow $0.5\mu m$ gate lengths and three layers of metal interconnect. Chips $17.5mm$ on a side, providing $4.9\times10^9\lambda^2$ of area, will be economical. The parameter λ is, to first order, process independent and is equivalent to one half of the minimum feature size [MC80]. For a $0.5\mu m$ process, λ is 0.25. This section presents a plan to implement a Processor Coupled node on such a chip. Resource requirements, such as the number of registers and the amount of operation cache memory per function unit, are still speculative and are subject to change based upon further design studies. Section 7.3.1 outlines the chip area calculations, while Section 7.3.2 examines the tradeoffs involved in register file design and intercluster communication.

Component	Area
64-bit FPU	$8\times10^7\lambda^2$
Integer Unit	$10^7\lambda^2$
128 64-bit Registers (4 read and 2 write ports)	$4\times10^7\lambda^2$
Operation Cache (512Bytes/thread, 4 threads)	$2\times10^7\lambda^2$
Operation Buffer	$10^6\lambda^2$
Scoreboard	$5\times10^6\lambda^2$
Control	$2\times10^7\lambda^2$
Total	$1.8\times10^8\lambda^2$

Table 7.1: Cluster components with area estimates.

7.3.1 Area Calculations

A four cluster coupled multi-ALU processor (MAP) with resources for four active threads is feasible with 1995 technology. Table 7.1 lists each component of a cluster, with its corresponding area requirement. Area estimates are made using λ design rules for current technologies. Each cluster requires a total area of $1.8\times10^8\lambda^2$. The register file is divided into an integer register bank and a floating point register bank, each with 16 registers. The clusters are connected by 8 64-bit buses. With 6λ wire pitch, these 512 cluster interconnect wires running the length of four clusters ($5\times10^4\lambda$) consume $1.5\times10^8\lambda^2$. Area for the 4 64-bit buses to connect clusters to the on-chip cache is $8\times10^7\lambda^2$. The remaining $4\times10^9\lambda^2$ is used for 8Mbits of SRAM. Figure 7.6 shows a floorplan for a MAP chip.

A multi-ALU node with multiple threads but without coupling requires the integer and floating point ALUs, the register files, and the operation caches. The additional components required by each cluster to implement Processor Coupling are the scoreboard ($5\times10^6\lambda^2$), the operation buffer ($10^6\lambda^2$), and approximately half of the control ($10^7\lambda^2$). This overhead is $1.6\times10^7\lambda^2$ per cluster for a total of $6.4\times10^7\lambda^2$, which is 9 percent of the entire cluster area, but less than 2 percent of the chip area.

7.3.2 Register Organization

In terms of speed and area, building register files with more than six ports is inefficient. A 32-port register cell requires at least 32 word lines and 32 bit lines. If a 6λ wiring pitch is used, the cell will be nearly 200λ on a side with an area of $4\times10^4\lambda^2$. The area of 512

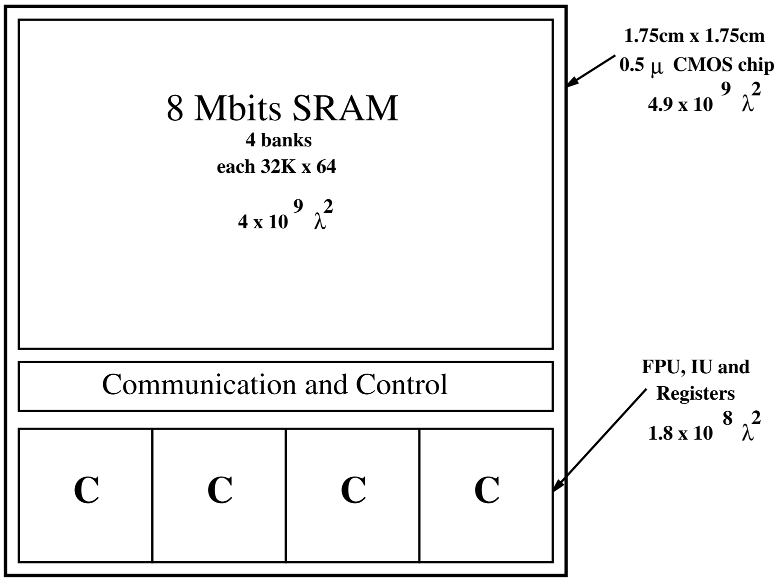

Figure 7.6: Floorplan for a four cluster multi-ALU processing node.

64-bit registers (the same number specified for a Processor Coupled node) organized in a unified 32-port register file would be $1.3{\times}10^9\lambda^2$. This is roughly one-quarter of the total chip area. Furthermore, the long wires required by a single register file are likely to induce clock frequency limiting delays. Processor Coupling distributes the registers into individual register files local to a cluster. Partitioning a cluster's registers into integer registers and floating point registers further reduces the number of ports required. The area of a six-ported register cell is $5{\times}10^3\lambda^2$. Building eight register files of 64 64-bit registers requires $1.6{\times}10^8\lambda^2$. With the function unit interconnect area of $1.5{\times}10^8\lambda^2$ needed to move data from cluster to cluster, the total area required is $3.1{\times}10^8\lambda^2$. This is less than seven percent of the total chip area. Although distributing the register files will require more operations to be executed, cycle time is reduced and area for other important components becomes available.

7.4 Summary

Processor Coupling employs a five stage pipeline. In addition to the four stages of a conventional pipeline, one additional stage, scoreboard check (SC), is necessary to perform synchronization on registers and fast selection of threads. On each cycle a different thread's operation may be selected for execution from the operation buffer. Arbitration for communication channels between threads is performed concurrently with operation execution two cycles before the operation's result is ready. If an operation cannot gain access to the interconnection network wires, the pipeline on which it is executing will stall. A wired-or `NOT_DONE` line is used to synchronize a thread's operations across function units. The `NOT_DONE` line is asserted until all function units have issued their operation from that instruction. Once all operations from an instruction have been issued, operations from the next instruction may begin to issue. The operation prefetch buffer and operation buffer are organized so that a thread can issue one operation every cycle.

Processor Coupling can be realized in a single chip implementation. A multi-ALU processor (MAP) consisting of four clusters, each with an integer unit, a memory unit, and a floating point unit, is feasible. The remaining area can be used for 8Mbits of SRAM. If process technology allows, 16Mbits of DRAM may be substituted for the SRAM. A thread's register set is distributed across multiple clusters to reduce the number of register file ports and limit the amount of register file area required. If one global register file with enough ports to satisfy all of the function units were used, it would require one quarter of the entire chip area.

Chapter 8

Conclusion

8.1 Summary

Processor Coupling combines compile time and runtime scheduling to exploit instruction-level parallelism while maintaining high function unit utilization. A compiler schedules each thread across multiple ALUs to exploit instruction-level parallelism. The schedules of several active threads are interleaved at runtime by coupling threads to ALUs on a cycle-by-cycle basis. This interleaving makes use of resources that would otherwise be left idle by vacancies in a single thread's schedule and by stalls caused by synchronization and statically indeterminate latencies. Processor Coupling combines the ability to exploit parallelism at the level of a single operation as in a VLIW machine with the latency tolerance and high utilization of a multithreaded architecture.

An experimental environment consisting of a compiler and a simulator was built to test the viability of Processor Coupling and to compare coupling with statically scheduled and multiprocessor machine models. Programs are written in PCTL (Processor Coupling Test Language) and are compiled using ISC (Instruction Scheduling Compiler). A configuration file provides the relevant machine parameters, such as the number and type of function units. PCS (Processor Coupling Simulator) executes the code generated by ISC and produces runtime statistics. The memory system model and intercluster communication parameters are specified in the PCS configuration file and can be changed by the user.

Four simple benchmarks (**Matrix**, **FFT**, **Model**, and **LUD**) were run on several types of simulated machines. On these particular benchmarks, a Processor Coupled node executes in 60% fewer cycles and achieves almost twice the utilization of a statically scheduled processor without coupling. Processor Coupling is more tolerant of both dynamic and static latencies than a single-threaded statically-scheduled machine. In the experiments with variable memory latency, a Processor Coupled node executes in 75% fewer cycles than a statically scheduled processor. Processor Coupling is also more tolerant of statically known function unit latencies, executing in only 10% more cycles when the floating point unit latency is increased from 1 to 5 cycles. The statically scheduled processor requires 90% more cycles for the same change in floating point unit latency.

Because Processor Coupling uses the low interaction latency available between function units, it has an advantage over a multiprocessor machine model. On sequential sections of code, Processor Coupling allows a thread to use all of the function units while the multiprocessor restricts use to only those function units within a single cluster. In sequential sections the multiprocessor model requires on average 2.9 times as many cycles as Processor Coupling. On the **FFT** benchmark, which has a significant sequential section, a total of 79% more cycles are required.

The simulations suggest the use of an interconnection scheme with three write ports on each register file. One port is devoted to intracluster communication, and the remaining ports are connected to buses that can be driven by function units in any cluster. This configuration gives nearly the same performance as complete connection (only 4% more cycles) for less cost (28% of the interconnect area for a four cluster machine). Performance also depends on the right balance of function unit types. Simulations suggest a configuration with four floating point units, four integer units, and one branch unit.

Processor Coupling is implemented by adding a scoreboard check (SC) stage containing an operation buffer to a standard four-stage pipeline. The SC stage ensures that an operation is enabled to issue only if all of its dependencies are satisfied. This check avoids wasting register bandwidth reading operands for instructions that are not ready. Adding this stage to the pipeline adds one cycle to the penalty for an incorrectly predicted branch but does not affect arithmetic latency. Preliminary circuit estimates indicate that since the machine's cycle time will be dominated by arithmetic and memory latencies, Processor

Coupling will not affect clock frequency.

A single-chip, four-cluster, multi-ALU processor is feasible with 1995 technology (0.5μ CMOS). Each cluster of such a chip consists of a floating point unit, an integer unit, registers, and control, occupying a total of $1.8 \times 10^8 \lambda^2$. The remaining area is allocated to on-chip communication ($2.3 \times 10^8 \lambda^2$) and 8Mbits of memory ($4 \times 10^9 \lambda^2$). The additional control logic required to implement Processor Coupling is less than 2% of the total area cost of the chip.

8.2 Future Work

The results presented in this thesis suggest that Processor Coupling is an attractive method for controlling multi-ALU processors. Because analysis is performed at an architectural level, the simulation environment was designed to be flexible so that many aspects of the design could be quickly evaluated. However many important details must be considered in much further depth. To complement this examination of multi-ALU processor control, studies of high-bandwidth memory systems, mechanisms for thread management, and compiler techniques are needed.

A high performance memory system is an extremely important aspect of Processor Coupling. Multiple operation issue requires high instruction fetch bandwidth, while using multiple threads increases the number of data references that need to be serviced. An interleaved memory system and a split phase memory transaction protocol can both be used to increase memory system performance. Segmentation can be used to provide a level of protection between threads. Threads that need to communicate with one another share segments. The cache must be organized to allow simultaneous requests and prevent interference from multiple active threads from limiting performance.

Thread management mechanisms require further exploration as well. Two parameters that must be studied are the number of threads in the active set and the number of active threads reserved for fault handling and system functions. Mechanisms and strategies for swapping threads in and out of the active set must be developed. The overhead to swap threads will have a profound impact on the granularity of tasks and the amount of resources, such as registers and cache memory, that will be required.

Processor Coupling provides further opportunities to develop compiler technology. In addition to scheduling instructions within a thread, a compiler can try to anticipate memory latencies. For example if the compiler believes that a memory read will be local, it can immediately schedule the operation using that data. If the latency is predicted to be long, the compiler can schedule the consuming operation later in the pipe or even emit code to suspend the thread. The hardware interlocks in Processor Coupling prevent incorrect execution if the latency of a reference is mispredicted. Another class of optimizations are those between threads. The compiler may be able to use some heuristics to balance the load from several threads across all of the clusters.

As more powerful computers are built, advanced techniques for exploiting parallelism must be employed. Although instruction-level parallelism techniques have appeared in superscalar uniprocessors and in VLIW machines, these ideas have not been used in the growing number of massively parallel computers. Since multiple threads are used to increase utilization, a Processor Coupled multi-ALU chip is well suited as a node in a massively parallel machine. The M-Machine currently being designed is intended to exploit a hierarchy of parallelism, from the instruction-level parallelism techniques of Processor Coupling, to the coarser grained concurrency that can be discovered in parallel algorithms.

Appendix A

Benchmarks

A.1 Matrix Multiply

The PCTL programs in this section implement a 9×9 matrix multiply. The program `vliw-matmul` is used by the SEQ and STS modes described in Chapter 5. The threaded modes (Coupled and TPE) use the program `thr-matmul`. The ideal statically scheduled machine (Ideal) uses `unrolled-vliw-matmul`.

A.1.1 `vliw-matmul`

```
(declare
 ((a (array (9 9) float))
  (b (array (9 9) float))
  (c (array (9 9) float))
  (temp1 float) (i int) (j int))
 (for ((:= i 0) (< i 9) (:= i (+ i 1)))
      (begin
       (for ((:= j 0) (< j 9) (:= j (+ j 1)))
            (begin
             (:= temp1 (* (aref a (i 0)) (aref b (0 j))))
             (:= temp1 (+ temp1 (* (aref a (i 1)) (aref b (1 j)))))
             (:= temp1 (+ temp1 (* (aref a (i 2)) (aref b (2 j)))))
             (:= temp1 (+ temp1 (* (aref a (i 3)) (aref b (3 j)))))
             (:= temp1 (+ temp1 (* (aref a (i 4)) (aref b (4 j)))))
             (:= temp1 (+ temp1 (* (aref a (i 5)) (aref b (5 j)))))
             (:= temp1 (+ temp1 (* (aref a (i 6)) (aref b (6 j)))))
             (:= temp1 (+ temp1 (* (aref a (i 7)) (aref b (7 j)))))
             (:= temp1 (+ temp1 (* (aref a (i 8)) (aref b (8 j)))))
             (:= (aref c (i j)) temp1)))))))
```

A.1.2 thr-matmul

```
(declare
 ((a (array (9 9) float))
  (b (array (9 9) float))
  (c (array (9 9 ) float))
  (temp float) (i int) (j int)
  (inner-loop
   (lambda (a b c i j)
     (declare
      ((temp float))
      (begin
       (:= temp (* (aref a (i 0)) (aref b (0 j))))
       (:= temp (+ temp (* (aref a (i 1)) (aref b (1 j)))))
       (:= temp (+ temp (* (aref a (i 2)) (aref b (2 j)))))
       (:= temp (+ temp (* (aref a (i 3)) (aref b (3 j)))))
       (:= temp (+ temp (* (aref a (i 4)) (aref b (4 j)))))
       (:= temp (+ temp (* (aref a (i 5)) (aref b (5 j)))))
       (:= temp (+ temp (* (aref a (i 6)) (aref b (6 j)))))
       (:= temp (+ temp (* (aref a (i 7)) (aref b (7 j)))))
       (:= temp (+ temp (* (aref a (i 8)) (aref b (8 j)))))
       (:= (aref c (i j)) temp)))))
  (med-loop (lambda (a b c i)
             (declare ((j int))
                      (begin
                       (for ((:= j 0) (< j 9) (:= j (+ j 1)))
                            (call inner-loop a b c i j)))))))
 (begin
  (forall ((:= i 0) (< i 9) (:= i (+ i 1)))
          (call med-loop a b c i)))))
```

A.1.3 unrolled-vliw-matmul

```
(declare
 ((a (array (9 9) float))
  (b (array (9 9) float))
  (c (array (9 9) float))
  (i int) (j int)
  (inner-loop
   (lambda (a b c i j)
     (declare ((temp float))
              (begin
               (:= temp (* (aref a (i 0)) (aref b (0 j))))
               (:= temp (+ temp (* (aref a (i 1)) (aref b (1 j)))))
               (:= temp (+ temp (* (aref a (i 2)) (aref b (2 j)))))
               (:= temp (+ temp (* (aref a (i 3)) (aref b (3 j)))))
               (:= temp (+ temp (* (aref a (i 4)) (aref b (4 j)))))
               (:= temp (+ temp (* (aref a (i 5)) (aref b (5 j)))))
               (:= temp (+ temp (* (aref a (i 6)) (aref b (6 j)))))
               (:= temp (+ temp (* (aref a (i 7)) (aref b (7 j)))))
               (:= temp (+ temp (* (aref a (i 8)) (aref b (8 j)))))
               (:= (aref c (i j)) temp)))))
  (mid-loop (lambda (a b c i)
             (begin
              (call inner-loop a b c i 0)
```

```
              (call inner-loop a b c i 1)
              (call inner-loop a b c i 2)
              (call inner-loop a b c i 3)
              (call inner-loop a b c i 4)
              (call inner-loop a b c i 5)
              (call inner-loop a b c i 6)
              (call inner-loop a b c i 7)
              (call inner-loop a b c i 8)))))
 (begin
  (call mid-loop a b c 0)
  (call mid-loop a b c 1)
  (call mid-loop a b c 2)
  (call mid-loop a b c 3)
  (call mid-loop a b c 4)
  (call mid-loop a b c 5)
  (call mid-loop a b c 6)
  (call mid-loop a b c 7)
  (call mid-loop a b c 8))))
```

A.2 FFT

The code for the 32 point fast Fourier transform described in Chapter 6 appears below. The program `vliw-fft` is used by SEQ and STS modes, while the threaded modes (Coupled and TPE) use the program `thr-fft`. The ideal statically scheduled (Ideal) machines uses `unrolled-vliw-fft`.

A.2.1 `vliw-fft`

```
(declare
 ((data (array 64 float))
  (coeff (array 32 float))
  (nn int) (tempnn int) (group-size int) (num-groups int)
  (i int) (j int)
  (swap (lambda (a b)
          (declare ((temp float))
                   (begin
                    (:= temp a)
                    (:= a b)
                    (:= b temp)))))
  (mod (lambda (result dividend divisor)
         (declare ((temp int))
                  (begin
                   (:= temp (/ dividend divisor))
                   (:= temp (* temp divisor))
                   (:= result (- dividend temp))))))
  (bit-reverse
   (lambda (data nn)
     (declare
      ((i int) (j int)
```

```
      (b0 int) (b1 int) (b2 int) (b3 int) (b4 int))
     (begin
      (for ((:= i 0) (< i nn) (:= i (+ i 1)))
           (begin
            (call mod b0 i 2)
            (call mod b1 (>> i 1) 2)
            (call mod b2 (>> i 2) 2)
            (call mod b3 (>> i 3) 2)
            (call mod b4 (>> i 4) 2)
            (:= j (+ (+ (+ (* 16 b0)
                           (* 8 b1))
                        (+ (* 4 b2)
                           (* 2 b3)))
                     b4))
            (if (> j i)
                (begin
                 (call swap (aref data (* 2 j)) (aref data (* 2 i)))
                 (call swap (aref data (+ (* 2 j) 1))
                            (aref data (+ (* 2 i) 1)))))))))))
 (comp-bfly
  (lambda (data index group-size num-groups coeff)
    (declare ((index-1 int) (index-2 int) (coeff-index int)
              (wr float) (wi float) (tempr float) (tempi float)
              (group int) (group-offset int))
             (begin
              (:= group-offset (/ index num-groups))
              (:= group (- index (* group-offset num-groups)))
              (:= coeff-index (* 2 (* group-offset num-groups)))
              (:= index-1 (* 2 (+ (* group group-size)
                                  group-offset)))
              (:= index-2 (+ index-1 group-size))
              (:= wr (aref coeff coeff-index))
              (:= wi (aref coeff (+ coeff-index 1)))
              (:= tempr (- (* wr (aref data index-2))
                           (* wi (aref data (+ 1 index-2)))))
              (:= tempi (+ (* wr (aref data (+ 1 index-2)))
                           (* wi (aref data index-2))))
              (:= (aref data index-2)
                  (- (aref data index-1) tempr))
              (:= (aref data (+ 1 index-2))
                  (- (aref data (+ index-1 1)) tempi))
              (:= (aref data index-1)
                  (+ (aref data index-1) tempr))
              (:= (aref data (+ index-1 1))
                  (+ (aref data (+ index-1 1)) tempi)))))))
(begin
 (call bit-reverse data nn)
 (for ((:= i 1) (< i nn) (:= i (<< i 1)))
      (begin
       (:= group-size (* 2 i))
       (:= num-groups (/ nn group-size))
       (for ((:= j 0) (< j 16) (:= j (+ j 1)))
            (begin
             (call comp-bfly data j group-size
                   num-groups coeff))))))))
```

A.2.2 thr-fft

```
(declare
 ((data (array 64 float))
  (coeff (array 32 float))
  (nn int) (tempnn int) (group-size int) (num-groups int)
  (i int) (j int) (temp1 int) (temp2 int)
  (sync int) (sync-count int) (sync-start int)
  (swap (lambda (a b)
          (declare ((temp float))
                   (begin
                    (:= temp a)
                    (:= a b)
                    (:= b temp)))))
  (mod (lambda (result dividend divisor)
         (declare ((temp int))
                  (begin
                   (:= temp (/ dividend divisor))
                   (:= temp (* temp divisor))
                   (:= result (- dividend temp))))))
  (bit-reverse
   (lambda (data nn)
     (declare
      ((i int) (j int)
       (b0 int) (b1 int) (b2 int) (b3 int) (b4 int))
      (begin
       (for ((:= i 0) (< i nn) (:= i (+ i 1)))
            (begin
             (call mod b0 i 2)
             (call mod b1 (>> i 1) 2)
             (call mod b2 (>> i 2) 2)
             (call mod b3 (>> i 3) 2)
             (call mod b4 (>> i 4) 2)
             (:= j (+ (+ (+ (* 16 b0)
                            (* 8 b1))
                         (+ (* 4 b2)
                            (* 2 b3)))
                      b4))
             (if (> j i)
                 (begin
                  (call swap (aref data (* 2 j)) (aref data (* 2 i)))
                  (call swap (aref data (+ (* 2 j) 1)) (aref data (+ (* 2 i) 1))))))))))
  (comp-bfly
   (lambda (data index group-size num-groups coeff)
     (declare ((index-1 int) (index-2 int) (coeff-index int)
               (wr float) (wi float) (tempr float) (tempi float)
               (group int) (group-offset int) (in-sync int)
               (loc-group-size int) (loc-num-groups int))
              (begin
               (:= loc-group-size (leave group-size))
               (:= loc-num-groups (leave num-groups))
               (:= group-offset (/ index loc-num-groups))
               (:= group (- index
                            (* group-offset loc-num-groups)))
               (:= coeff-index (* 2
                                  (* group-offset loc-num-groups)))
```

```
                (:= index-1 (* 2 (+ (* group loc-group-size)
                                    group-offset)))
                (:= index-2 (+ index-1 loc-group-size))
                (:= wr (aref coeff coeff-index))
                (:= wi (aref coeff (+ coeff-index 1)))
                (:= tempr (- (* wr (aref data index-2))
                             (* wi (aref data (+ 1 index-2)))))
                (:= tempi (+ (* wr (aref data (+ 1 index-2)))
                             (* wi (aref data index-2))))
                (:= (aref data index-2)
                    (- (aref data index-1) tempr))
                (:= (aref data (+ 1 index-2))
                    (- (aref data (+ index-1 1)) tempi))
                (:= (aref data index-1)
                    (+ (aref data index-1) tempr))
                (:= (aref data (+ index-1 1))
                    (+ (aref data (+ index-1 1)) tempi))
                (begin-sync
                 (:= in-sync (consume sync))
                 (if (< in-sync 16)
                     (begin
                      (:= (produce sync) (+ in-sync 1)))
                   (:= (produce sync-start) in-sync)))))))
 (begin
  (call bit-reverse data nn)
  (:= (uncond sync-start) 0)
  (for ((:= i 1) (< i nn) (:= i (<< i 1)))
       (begin-sync
        (:= (uncond sync) 0)
        (:= temp1 (consume sync-start))
        (:= temp1 (+ temp1 1))
        (:= sync-count 0)
        (:= (uncond group-size) (* 2 i))
        (:= (uncond num-groups) (/ nn group-size))
        (begin-sync
         (forall ((:= j 0) (< j 16) (:= j (+ j 1)))
                 (begin
                  (call comp-bfly
                        data j group-size num-groups coeff))))
        (begin-sync
         (:= sync-count (consume sync))
         (if (< sync-count 16)
             (begin
              (:= (produce sync) (+ sync-count 1))
              (begin-sync
               (:= (uncond sync-count) (leave sync-start))))
           (:= (produce sync-start) sync-count))))))))
```

A.2.3 unrolled-vliw-fft

```
(declare
 ((data (array 64 float))
  (coeff (array 32 float))
  (nn int) (tempnn int) (group-size int) (num-groups int)
  (i int) (j int)
```

```
(swap (lambda (a b)
        (declare ((temp float))
                 (begin
                  (:= temp a)
                  (:= a b)
                  (:= b temp)))))
(mod (lambda (result dividend divisor)
       (declare ((temp int))
                (begin
                 (:= temp (/ dividend divisor))
                 (:= temp (* temp divisor))
                 (:= result (- dividend temp))))))
(bit-reverse-one
 (lambda (data i)
   (declare
    ((j int) (b0 int) (b1 int) (b2 int) (b3 int) (b4 int))
    (begin
     (call mod b0 i 2)
     (call mod b1 (>> i 1) 2)
     (call mod b2 (>> i 2) 2)
     (call mod b3 (>> i 3) 2)
     (call mod b4 (>> i 4) 2)
     (:= j (+ (+ (+ (* 16 b0)
                    (* 8 b1))
                 (+ (* 4 b2)
                    (* 2 b3)))
              b4))
     (call swap (aref data (* 2 j)) (aref data (* 2 i)))
     (call swap (aref data (+ (* 2 j) 1)) (aref data (+ (* 2 i) 1)))))))
(bit-reverse
 (lambda (data nn)
   (begin
    (call bit-reverse-one data 1)
    (call bit-reverse-one data 2)
    (call bit-reverse-one data 3)
    (call bit-reverse-one data 5)
    (call bit-reverse-one data 6)
    (call bit-reverse-one data 7)
    (call bit-reverse-one data 9)
    (call bit-reverse-one data 11)
    (call bit-reverse-one data 13)
    (call bit-reverse-one data 15)
    (call bit-reverse-one data 19)
    (call bit-reverse-one data 23))))
(comp-bfly
 (lambda (data index group-size num-groups coeff)
   (declare ((index-1 int) (index-2 int) (coeff-index int)
             (wr float) (wi float) (tempr float) (tempi float)
             (group int) (group-offset int))
            (begin
             (:= group-offset (/ index num-groups))
             (:= group (- index (* group-offset num-groups)))
             (:= coeff-index (* 2 (* group-offset num-groups)))
             (:= index-1 (* 2 (+ (* group group-size)
                                 group-offset)))
             (:= index-2 (+ index-1 group-size))
```

```
              (:= wr (aref coeff coeff-index))
              (:= wi (aref coeff (+ coeff-index 1)))
              (:= tempr (- (* wr (aref data index-2))
                           (* wi (aref data (+ 1 index-2)))))
              (:= tempi (+ (* wr (aref data (+ 1 index-2)))
                           (* wi (aref data index-2))))
              (:= (aref data index-2)
                  (- (aref data index-1) tempr))
              (:= (aref data (+ 1 index-2))
                  (- (aref data (+ index-1 1)) tempi))
              (:= (aref data index-1)
                  (+ (aref data index-1) tempr))
              (:= (aref data (+ index-1 1))
                  (+ (aref data (+ index-1 1)) tempi)))))))
 (begin
  (call bit-reverse data nn)
  (for ((:= i 1) (< i nn) (:= i (<< i 1)))
       (begin
        (:= group-size (* 2 i))
        (:= num-groups (/ nn group-size))
        (call comp-bfly data 0 group-size num-groups coeff)
        (call comp-bfly data 1 group-size num-groups coeff)
        (call comp-bfly data 2 group-size num-groups coeff)
        (call comp-bfly data 3 group-size num-groups coeff)
        (call comp-bfly data 4 group-size num-groups coeff)
        (call comp-bfly data 5 group-size num-groups coeff)
        (call comp-bfly data 6 group-size num-groups coeff)
        (call comp-bfly data 7 group-size num-groups coeff)
        (call comp-bfly data 8 group-size num-groups coeff)
        (call comp-bfly data 9 group-size num-groups coeff)
        (call comp-bfly data 10 group-size num-groups coeff)
        (call comp-bfly data 11 group-size num-groups coeff)
        (call comp-bfly data 12 group-size num-groups coeff)
        (call comp-bfly data 13 group-size num-groups coeff)
        (call comp-bfly data 14 group-size num-groups coeff)
        (call comp-bfly data 15 group-size num-groups coeff))))))
```

A.3 Model Evaluation

The code for the circuit simulation model evaluator described in Chapter 6 appears below. The program `vliw-model` is used by the SEQ and STS modes, while the base threaded mode program which performs load balancing for Coupled and TPE is `thr-iter-model`. The threaded version that creates a new thread for each loop iteration without load balancing is `thr-model`. The autoscheduling version is found in `thr-auto-model`.

A.3.1 vliw-model

```
(declare
 ((num-elements int) (element-index (array 100 int))
```

```
(element (array 500 int))
(value-array (array 200 float))
(voltage (array 500 float))
(delta-t float)
(delta-diff float)
(Vtn float) (Kn float)
(Vtp float) (Kp float)
(current (array 500 float))
(element-address int)
(i int) (type int) (tempi int)
(error int)
(eval-res
 (lambda (in-element)
   (declare ((delta-c float)
             (element int)
             (e1 int) (e2 int) (e3 int))
            (begin
             (:= element in-element)
             (:= e1 (aref element 1))
             (:= e2 (aref element 2))
             (:= e3 (aref element 3))
             (:= delta-c (/ (- (aref voltage e1)
                               (aref voltage e2))
                            (aref value-array e3)))
             (:= (aref (produce current) e1)
                 (+ (aref (consume current) e1)
                    delta-c))
             (:= (aref (produce current) e2)
                 (- (aref (consume current) e2)
                    delta-c))))))
(eval-cap
 (lambda (in-element)
   (declare ((delta-c float)
             (delta-v float)
             (conductance float)
             (element int)
             (e1 int) (e2 int) (e3 int))
            (begin
             (:= element in-element)
             (:= e1 (aref element 1))
             (:= e2 (aref element 2))
             (:= e3 (aref element 3))
             (:= conductance (/ (aref value-array e3)
                                delta-t))
             (:= delta-v  (- (aref voltage e1)
                             (aref voltage e2)))
             (:= delta-c (- (* delta-v conductance)
                            (* delta-v
                               (* delta-diff conductance))))
             (:= (aref (produce current) e1)
                 (+ (aref (consume current) e1)
                    delta-c))
             (begin-sync
              (:= (aref (produce current) e2)
                  (- (aref (consume current) e2)
                     delta-c)))))))
```

```
(eval-nfet
 (lambda (in-element)
   (declare ((drain int) (gate int) (source int) (Vg float)
             (Vgst float) (Vd float) (Vs float) (Vds float)
             (element int)
             (Ids float) (KS float) (WoL float) (temp float)
             (e1 int) (e2 int) (e3 int) (e4 int) (e5 int))
            (begin
             (:= element in-element)
             (:= e1 (aref element 1))
             (:= e2 (aref element 2))
             (:= e3 (aref element 3))
             (:= Vd (aref voltage e1))
             (:= Vs (aref voltage e3))
             (:= Vg (aref voltage e2))
             (if (> Vs Vd)
                 (begin
                  (:= drain e3)
                  (:= source e1)
                  (:= temp Vd)
                  (:= Vd Vs)
                  (:= Vs temp))
               (begin
                (:= drain e1)
                (:= source e3)))
             (:= Vgst (- Vg (+ Vs Vtn)))
             (if (> Vgst 0)
                 (begin
                  (:= e4 (aref element 4))
                  (:= e5 (aref element 5))
                  (:= Vds (- Vd Vs))
                  (:= KS (* Kn (/ (aref value-array e4)
                                  (aref value-array e5))))
                  (if (>= Vgst Vds)
                      (begin
                       (:= Ids (* KS (- (* 2 (* Vgst Vds))
                                        (* Vds Vds)))))
                    (begin
                     (:= Ids (* KS (* Vgst Vgst)))))
                  (:= (aref (produce current) drain)
                      (+ (aref (consume current) drain) Ids))
                  (begin-sync
                   (:= (aref (produce current) source)
                       (- (aref (consume current) source) Ids)))))))))
(eval-pfet
 (lambda (in-element)
   (declare ((drain int) (gate int) (source int) (Vg float)
             (Vgst float) (Vd float) (Vs float) (Vds float)
             (element int)
             (Ids float) (KS float) (WoL float) (temp float)
             (e1 int) (e2 int) (e3 int) (e4 int) (e5 int))
            (begin
             (:= element in-element)
             (:= e1 (aref element 1))
             (:= e2 (aref element 2))
             (:= e3 (aref element 3))
```

```
                (:= Vd (aref voltage e1))
                (:= Vs (aref voltage e3))
                (:= Vg (aref voltage e2))
                (if (< Vs Vd)
                    (begin
                     (:= drain e3)
                     (:= source e1)
                     (:= temp Vd)
                     (:= Vd Vs)
                     (:= Vs temp))
                  (begin
                   (:= drain e1)
                   (:= source e3)))
                (:= Vgst (- Vg (+ Vs Vtp)))
                (if (< Vgst 0)
                    (begin
                     (:= e4 (aref element 4))
                     (:= e5 (aref element 5))
                     (:= Vds (- Vd Vs))
                     (:= KS (* Kp (/ (aref value-array e4)
                                     (aref value-array e5))))
                     (if (<= Vgst Vds)
                         (begin
                          (:= Ids (* KS (- (* 2 (* Vgst Vds))
                                           (* Vds Vds)))))
                       (begin
                        (:= Ids (* KS (* Vgst Vgst)))))
                     (:= (aref (produce current) drain)
                         (- (aref (consume current) drain) Ids))
                     (begin-sync
                      (:= (aref (produce current) source)
                          (+ (aref (consume current) source) Ids)))))))))
(begin
 (for ((:= i 0) (< i num-elements) (:= i (+ i 1)))
      (begin
       (:= tempi (aref element-index i))
       (:= type (aref element tempi))
       (:= (uncond element-address) (+ element tempi))
       (if (== type 2)
           (begin
            (call eval-nfet element-address))
         (begin
          (if (== type 3)
              (begin
               (call eval-pfet element-address))
            (begin
             (if (== type 0)
                 (begin
                  (call eval-res element-address))
               (begin
                (if (== type 1)
                    (begin
                     (call eval-cap element-address))
                  (begin
                   (:= error type)))))))))))))
```

A.3.2 thr-iter-model

```
(declare
 ((num-elements int) (element-index (array 100 int))
  (top-element (array 500 int))
  (value-array (array 200 float))
  (voltage (array 500 float))
  (delta-t float)
  (delta-diff float)
  (Vtn float) (Kn float)
  (Vtp float) (Kp float)
  (current (array 500 float))
  (i int) (sync int) (sync-temp int)
  (error int)
  (eval-res
   (lambda (element)
     (declare ((delta-c float)
               (e1 int) (e2 int) (e3 int))
              (begin
               (:= e1 (aref element 1))
               (:= e2 (aref element 2))
               (:= e3 (aref element 3))
               (:= delta-c (/ (- (aref voltage e1)
                                 (aref voltage e2))
                              (aref value-array e3)))
               (:= (aref (produce current) e1)
                   (+ (aref (consume current) e1)
                      delta-c))
               (begin-sync
                (:= (aref (produce current) e2)
                    (- (aref (consume current) e2)
                       delta-c)))))))
  (eval-cap
   (lambda (element)
     (declare ((delta-c float)
               (delta-v float)
               (conductance float)
               (e1 int) (e2 int) (e3 int))
              (begin
               (:= e1 (aref element 1))
               (:= e2 (aref element 2))
               (:= e3 (aref element 3))
               (:= conductance (/ (aref value-array e3)
                                  delta-t))
               (:= delta-v  (- (aref voltage e1)
                               (aref voltage e2)))
               (:= delta-c (- (* delta-v conductance)
                              (* delta-v
                                 (* delta-diff conductance))))
               (:= (aref (produce current) e1)
                   (+ (aref (consume current) e1)
                      delta-c))
               (begin-sync
                (:= (aref (produce current) e2)
                    (- (aref (consume current) e2)
                       delta-c)))))))
```

```
(eval-nfet
 (lambda (element)
   (declare ((drain int) (gate int) (source int) (Vg float)
             (Vgst float) (Vd float) (Vs float) (Vds float)
             (Ids float) (KS float) (WoL float) (temp float)
             (e1 int) (e2 int) (e3 int) (e4 int) (e5 int))
            (begin
             (:= e1 (aref element 1))
             (:= e2 (aref element 2))
             (:= e3 (aref element 3))
             (:= Vd (aref voltage e1))
             (:= Vs (aref voltage e3))
             (:= Vg (aref voltage e2))
             (if (> Vs Vd)
                 (begin
                  (:= drain e3)
                  (:= source e1)
                  (:= temp Vd)
                  (:= Vd Vs)
                  (:= Vs temp))
               (begin
                (:= drain e1)
                (:= source e3)))
             (:= Vgst (- Vg (+ Vs Vtn)))
             (if (> Vgst 0)
                 (begin
                  (:= e4 (aref element 4))
                  (:= e5 (aref element 5))
                  (:= Vds (- Vd Vs))
                  (:= KS (* Kn (/ (aref value-array e4)
                                  (aref value-array e5))))
                  (if (>= Vgst Vds)
                      (begin
                       (:= Ids (* KS (- (* 2 (* Vgst Vds))
                                        (* Vds Vds)))))
                    (begin
                     (:= Ids (* KS (* Vgst Vgst)))))
                  (:= (aref (produce current) drain)
                      (+ (aref (consume current) drain) Ids))
                  (begin-sync
                   (:= (aref (produce current) source)
                       (- (aref (consume current) source)
                          Ids)))))))))
(eval-pfet
 (lambda (element)
   (declare ((drain int) (gate int) (source int) (Vg float)
             (Vgst float) (Vd float) (Vs float) (Vds float)
             (Ids float) (KS float) (WoL float) (temp float)
             (e1 int) (e2 int) (e3 int) (e4 int) (e5 int))
            (begin
             (:= e1 (aref element 1))
             (:= e2 (aref element 2))
             (:= e3 (aref element 3))
             (:= Vd (aref voltage e1))
             (:= Vs (aref voltage e3))
             (:= Vg (aref voltage e2))
```

```
                (if (< Vs Vd)
                    (begin
                     (:= drain e3)
                     (:= source e1)
                     (:= temp Vd)
                     (:= Vd Vs)
                     (:= Vs temp))
                  (begin
                   (:= drain e1)
                   (:= source e3)))
                (:= Vgst (- Vg (+ Vs Vtp)))
                (if (< Vgst 0)
                    (begin
                     (:= e4 (aref element 4))
                     (:= e5 (aref element 5))
                     (:= Vds (- Vd Vs))
                     (:= KS (* Kp (/ (aref value-array e4)
                                     (aref value-array e5))))
                     (if (<= Vgst Vds)
                         (begin
                          (:= Ids (* KS (- (* 2 (* Vgst Vds))
                                           (* Vds Vds)))))
                       (begin
                        (:= Ids (* KS (* Vgst Vgst)))))
                     (:= (aref (produce current) drain)
                         (- (aref (consume current) drain) Ids))
                     (begin-sync
                      (:= (aref (produce current) source)
                          (+ (aref (consume current) source)
                             Ids))))))))
(eval-ctl
 (lambda (el-num sync)
   (declare ((tempi int) (type int)
             (element-address int) (local-i int))
            (begin
             (:= local-i (leave el-num))
             (begin-sync
              (:= (produce sync) 0)
              (:= tempi (aref element-index local-i))
              (:= type (aref top-element tempi))
              (:= (uncond element-address) (+ top-element tempi))
              (if (== type 2)
                  (begin
                   (call eval-nfet element-address))
                (begin
                 (if (== type 3)
                     (begin
                      (call eval-pfet element-address))
                   (begin
                    (if (== type 0)
                        (begin
                         (call eval-res element-address))
                      (begin
                       (if (== type 1)
                           (begin
                            (call eval-cap element-address))
```

```
                          (begin
                           (:= error type))))))))))))))
 (begin
  (forall-iterate ((:= i 0) (< i num-elements) (:= i (+ i 1)))
                  (begin
                   (fork (call eval-ctl i sync))
                   (begin-sync
                    (:= sync-temp (+ (consume sync) 1))))))))
```

A.3.3 thr-model

```
(declare
 ((num-elements int) (element-index (array 100 int))
  (top-element (array 500 int))
  (value-array (array 200 float))
  (voltage (array 500 float))
  (delta-t float)
  (delta-diff float)
  (Vtn float) (Kn float)
  (Vtp float) (Kp float)
  (current (array 500 float))
  (i int) (sync int) (sync-temp int)
  (error int)
  (eval-res
   (lambda (element)
     (declare ((delta-c float)
               (e1 int) (e2 int) (e3 int))
              (begin
               (:= e1 (aref element 1))
               (:= e2 (aref element 2))
               (:= e3 (aref element 3))
               (:= delta-c (/ (- (aref voltage e1)
                                 (aref voltage e2))
                              (aref value-array e3)))
               (:= (aref (produce current) e1)
                   (+ (aref (consume current) e1)
                      delta-c))
               (begin-sync
                (:= (aref (produce current) e2)
                    (- (aref (consume current) e2)
                       delta-c)))))))
  (eval-cap
   (lambda (element)
     (declare ((delta-c float)
               (delta-v float)
               (conductance float)
               (e1 int) (e2 int) (e3 int))
              (begin
               (:= e1 (aref element 1))
               (:= e2 (aref element 2))
               (:= e3 (aref element 3))
               (:= conductance (/ (aref value-array e3)
                                  delta-t))
               (:= delta-v  (- (aref voltage e1)
                               (aref voltage e2)))
```

```
            (:= delta-c (- (* delta-v conductance)
                           (* delta-v
                              (* delta-diff conductance))))
            (:= (aref (produce current) e1)
                (+ (aref (consume current) e1)
                   delta-c))
            (begin-sync
             (:= (aref (produce current) e2)
                 (- (aref (consume current) e2)
                    delta-c))))))
(eval-nfet
 (lambda (element)
   (declare ((drain int) (gate int) (source int) (Vg float)
            (Vgst float) (Vd float) (Vs float) (Vds float)
            (Ids float) (KS float) (WoL float) (temp float)
            (e1 int) (e2 int) (e3 int) (e4 int) (e5 int))
           (begin
            (:= e1 (aref element 1))
            (:= e2 (aref element 2))
            (:= e3 (aref element 3))
            (:= Vd (aref voltage e1))
            (:= Vs (aref voltage e3))
            (:= Vg (aref voltage e2))
            (if (> Vs Vd)
                (begin
                 (:= drain e3)
                 (:= source e1)
                 (:= temp Vd)
                 (:= Vd Vs)
                 (:= Vs temp))
              (begin
               (:= drain e1)
               (:= source e3)))
            (:= Vgst (- Vg (+ Vs Vtn)))
            (if (> Vgst 0)
                (begin
                 (:= e4 (aref element 4))
                 (:= e5 (aref element 5))
                 (:= Vds (- Vd Vs))
                 (:= KS (* Kn (/ (aref value-array e4)
                                 (aref value-array e5))))
                 (if (>= Vgst Vds)
                     (begin
                      (:= Ids (* KS (- (* 2 (* Vgst Vds))
                                       (* Vds Vds)))))
                   (begin
                    (:= Ids (* KS (* Vgst Vgst)))))
                 (:= (aref (produce current) drain)
                     (+ (aref (consume current) drain) Ids))
                 (begin-sync
                  (:= (aref (produce current) source)
                      (- (aref (consume current) source)
                         Ids)))))))))
(eval-pfet
 (lambda (element)
   (declare ((drain int) (gate int) (source int) (Vg float)
```

```
             (Vgst float) (Vd float) (Vs float) (Vds float)
             (Ids float) (KS float) (WoL float) (temp float)
             (e1 int) (e2 int) (e3 int) (e4 int) (e5 int))
            (begin
             (:= e1 (aref element 1))
             (:= e2 (aref element 2))
             (:= e3 (aref element 3))
             (:= Vd (aref voltage e1))
             (:= Vs (aref voltage e3))
             (:= Vg (aref voltage e2))
             (if (< Vs Vd)
                 (begin
                  (:= drain e3)
                  (:= source e1)
                  (:= temp Vd)
                  (:= Vd Vs)
                  (:= Vs temp))
               (begin
                (:= drain e1)
                (:= source e3)))
             (:= Vgst (- Vg (+ Vs Vtp)))
             (if (< Vgst 0)
                 (begin
                  (:= e4 (aref element 4))
                  (:= e5 (aref element 5))
                  (:= Vds (- Vd Vs))
                  (:= KS (* Kp (/ (aref value-array e4)
                                  (aref value-array e5))))
                  (if (<= Vgst Vds)
                      (begin
                       (:= Ids (* KS (- (* 2 (* Vgst Vds))
                                        (* Vds Vds)))))
                    (begin
                     (:= Ids (* KS (* Vgst Vgst)))))
                  (:= (aref (produce current) drain)
                      (- (aref (consume current) drain) Ids))
                  (begin-sync
                   (:= (aref (produce current) source)
                       (+ (aref (consume current) source)
                          Ids)))))))))
(eval-ctl
 (lambda (el-num sync)
   (declare ((tempi int) (type int)
             (element-address int) (local-i int))
            (begin
             (:= local-i (leave el-num))
             (begin-sync
              (:= (produce sync) 0)
              (:= tempi (aref element-index local-i))
              (:= type (aref top-element tempi))
              (:= (uncond element-address) (+ top-element tempi))
              (if (== type 2)
                  (begin
                   (call eval-nfet element-address))
                (begin
                 (if (== type 3)
```

```
                    (begin
                     (call eval-pfet element-address))
                  (begin
                   (if (== type 0)
                       (begin
                        (call eval-res element-address))
                     (begin
                      (if (== type 1)
                          (begin
                           (call eval-cap element-address))
                        (begin
                         (:= error type))))))))))))))
 (begin
  (for ((:= (uncond i) 0) (< i num-elements) (:= (uncond i) (+ i 1)))
       (begin
        (fork (call eval-ctl i sync))
        (begin-sync
         (:= sync-temp (+ (consume sync) 1))))))))
```

A.3.4 thr-auto-model

```
(declare
 ((num-elements int) (element-index (array 100 int))
  (top-element (array 500 int))
  (value-array (array 200 float))
  (voltage (array 500 float))
  (delta-t float)
  (delta-diff float)
  (Vtn float) (Kn float)
  (Vtp float) (Kp float)
  (current (array 500 float))
  (index int)
  (i int) (sync int) (sync-temp int)
  (error int)
  (eval-res
   (lambda (element)
     (declare ((delta-c float)
               (e1 int) (e2 int) (e3 int))
              (begin
               (:= e1 (aref element 1))
               (:= e2 (aref element 2))
               (:= e3 (aref element 3))
               (:= delta-c (/ (- (aref voltage e1)
                                 (aref voltage e2))
                              (aref value-array e3)))
               (:= (aref (produce current) e1)
                   (+ (aref (consume current) e1)
                      delta-c))
               (begin-sync
                (:= (aref (produce current) e2)
                    (- (aref (consume current) e2)
                       delta-c)))))))
  (eval-cap
   (lambda (element)
     (declare ((delta-c float)
```

```
                (delta-v float)
                (conductance float)
                (e1 int) (e2 int) (e3 int))
               (begin
                (:= e1 (aref element 1))
                (:= e2 (aref element 2))
                (:= e3 (aref element 3))
                (:= conductance (/ (aref value-array e3)
                                   delta-t))
                (:= delta-v  (- (aref voltage e1)
                                (aref voltage e2)))
                (:= delta-c (- (* delta-v conductance)
                               (* delta-v
                                  (* delta-diff conductance))))
                (:= (aref (produce current) e1)
                    (+ (aref (consume current) e1)
                       delta-c))
                (begin-sync
                 (:= (aref (produce current) e2)
                     (- (aref (consume current) e2)
                        delta-c))))))
 (eval-nfet
  (lambda (element)
    (declare ((drain int) (gate int) (source int) (Vg float)
                (Vgst float) (Vd float) (Vs float) (Vds float)
                (Ids float) (KS float) (WoL float) (temp float)
                (e1 int) (e2 int) (e3 int) (e4 int) (e5 int))
               (begin
                (:= e1 (aref element 1))
                (:= e2 (aref element 2))
                (:= e3 (aref element 3))
                (:= Vd (aref voltage e1))
                (:= Vs (aref voltage e3))
                (:= Vg (aref voltage e2))
                (if (> Vs Vd)
                    (begin
                     (:= drain e3)
                     (:= source e1)
                     (:= temp Vd)
                     (:= Vd Vs)
                     (:= Vs temp))
                   (begin
                    (:= drain e1)
                    (:= source e3)))
                (:= Vgst (- Vg (+ Vs Vtn)))
                (if (> Vgst 0)
                    (begin
                     (:= e4 (aref element 4))
                     (:= e5 (aref element 5))
                     (:= Vds (- Vd Vs))
                     (:= KS (* Kn (/ (aref value-array e4)
                                     (aref value-array e5))))
                     (if (>= Vgst Vds)
                         (begin
                          (:= Ids (* KS (- (* 2 (* Vgst Vds))
                                           (* Vds Vds)))))
```

```
                    (begin
                     (:= Ids (* KS (* Vgst Vgst)))))
                 (:= (aref (produce current) drain)
                     (+ (aref (consume current) drain) Ids))
                 (begin-sync
                  (:= (aref (produce current) source)
                      (- (aref (consume current) source)
                         Ids)))))))))
(eval-pfet
 (lambda (element)
   (declare ((drain int) (gate int) (source int) (Vg float)
             (Vgst float) (Vd float) (Vs float) (Vds float)
             (Ids float) (KS float) (WoL float) (temp float)
             (e1 int) (e2 int) (e3 int) (e4 int) (e5 int))
            (begin
             (:= e1 (aref element 1))
             (:= e2 (aref element 2))
             (:= e3 (aref element 3))
             (:= Vd (aref voltage e1))
             (:= Vs (aref voltage e3))
             (:= Vg (aref voltage e2))
             (if (< Vs Vd)
                 (begin
                  (:= drain e3)
                  (:= source e1)
                  (:= temp Vd)
                  (:= Vd Vs)
                  (:= Vs temp))
               (begin
                (:= drain e1)
                (:= source e3)))
             (:= Vgst (- Vg (+ Vs Vtp)))
             (if (< Vgst 0)
                 (begin
                  (:= e4 (aref element 4))
                  (:= e5 (aref element 5))
                  (:= Vds (- Vd Vs))
                  (:= KS (* Kp (/ (aref value-array e4)
                                  (aref value-array e5))))
                  (if (<= Vgst Vds)
                      (begin
                       (:= Ids (* KS (- (* 2 (* Vgst Vds))
                                        (* Vds Vds)))))
                    (begin
                     (:= Ids (* KS (* Vgst Vgst)))))
                  (:= (aref (produce current) drain)
                      (- (aref (consume current) drain) Ids))
                  (begin-sync
                   (:= (aref (produce current) source)
                       (+ (aref (consume current) source)
                          Ids)))))))))
(eval-ctl
 (lambda ()
   (declare ((tempi int) (type int)
             (element-address int) (local-i int))
            (begin
```

```
                (:= local-i (consume index))
                (:= (produce index) (+ local-i 1))
                (while (< local-i num-elements)
                  (begin
                   (:= tempi (aref element-index local-i))
                   (:= type (aref top-element tempi))
                   (:= (uncond element-address) (+ top-element tempi))
                   (if (== type 2)
                       (begin
                        (call eval-nfet element-address))
                     (begin
                      (if (== type 3)
                          (begin
                           (call eval-pfet element-address))
                        (begin
                         (if (== type 0)
                             (begin
                              (call eval-res element-address))
                           (begin
                            (if (== type 1)
                                (begin
                                 (call eval-cap element-address))
                              (begin
                               (:= error type)))))))))
                   (begin-sync
                    (:= local-i (consume index))
                    (:= (produce index) (+ local-i 1))))))))))
   (begin
    (:= (uncond index) 0)
    (forall ((:= i 0) (< i 4) (:= i (+ i 1)))
            (begin
             (call eval-ctl))))))
```

A.4 LU Decomposition

The LU decomposition code described in Chapter 6 is listed in this section. The program `vliw-lud` is used by the SEQ and STS modes, while the base threaded mode program which performs load balancing for Coupled and TPE is `thr-iter-lud`. The threaded version that creates a new thread for each loop iteration without load balancing is `thr-lud`. The autoscheduling version is found in `thr-auto-lud`.

A.4.1 `vliw-lud`

```
(declare
 ((matrix (array 3600 float))
  (nrows int) (offset (array 110 int))
  (diag (array 500 int))
  (r-in-c (array 1000 int))
  (c-in-r (array 1000 int))
```

```
   (i int) (j int) (k int)
   (diag-val float)
   (elim-col-num int) (elim-col-index int)
   (elim-row-num int) (elim-row-index int)
   (index-i int) (index-j int) (index-k int)
   (temp1-index int) (temp2-index int))
  (begin
   (for ((:= i 0) (< i nrows) (:= i (+ i 1)))
        (begin
         (:= diag-val
             (/ 1 (aref matrix (+ i (aref offset i)))))
         (:= elim-row-num (aref diag (* 4 i)))
         (:= elim-row-index (aref diag (+ 1 (* 4 i))))
         (:= elim-col-num (aref diag (+ 2 (* 4 i))))
         (for ((:= j elim-row-num) (> j 0) (:= j (- j 1)))
              (begin
               (:= temp2-index (aref r-in-c elim-row-index))
               (:= temp1-index (+ i (aref offset temp2-index)))
               (:= (aref matrix temp1-index)
                   (* (aref matrix temp1-index) diag-val))
               (:= elim-col-index (aref diag (+ 3 (* 4 i))))
               (:= index-i i)
               (:= index-j (aref r-in-c elim-row-index))
               (for ((:= k elim-col-num) (> k 0) (:= k (- k 1)))
                    (begin
                     (:= index-k (aref c-in-r elim-col-index))
                     (:= (aref matrix
                                (+ index-k
                                   (aref offset index-j)))
                         (- (aref matrix
                                   (+ index-k
                                      (aref offset index-j)))
                            (* (aref matrix
                                      (+ index-i
                                         (aref offset index-j)))
                               (aref matrix
                                      (+ index-k
                                         (aref offset index-i))))))
                     (:= elim-col-index (+ elim-col-index 1))))
               (:= elim-row-index (+ elim-row-index 1)))))))))
```

A.4.2 thr-iter-lud

```
 (declare
  ((matrix (array 3600 float))
   (nrows int) (offset (array 110 int))
   (diag (array 500 int))
   (r-in-c (array 1000 int))
   (c-in-r (array 1000 int))
   (i int) (j int)
   (diag-val float)
   (elim-col-num int)
   (elim-row-num int) (elim-row-index int)
   (up-semaphore int) (sync int) (sync-count int)
   (temp1 int) (sync-start int)
```

```
   (mini-sync int)
   (row-norm
    (lambda (elim-row-index)
      (declare ((index-i int) (index-j int)
                (index-k int) (k int) (in-sync int)
                (temp1-index int) (temp2-index int)
                (elim-col-index int) (local-row-index int))
               (begin
                (:= local-row-index (consume elim-row-index))
                (:= (produce up-semaphore) 1)
                (:= temp2-index (aref r-in-c local-row-index))
                (:= index-i (leave i))
                (:= temp1-index (+ index-i (aref offset temp2-index)))
                (:= (aref matrix temp1-index)
                    (* (aref matrix temp1-index) diag-val))
                (:= elim-col-index (aref diag (+ 3 (* 4 index-i))))
                (:= index-j (aref r-in-c local-row-index))
                (for ((:= k (leave elim-col-num)) (> k 0) (:= k (- k 1)))
                     (begin
                      (:= index-k (aref c-in-r elim-col-index))
                      (:= (aref matrix
                                (+ index-k
                                   (aref offset index-j)))
                          (- (aref matrix
                                   (+ index-k
                                      (aref offset index-j)))
                             (* (aref matrix
                                      (+ index-i
                                         (aref offset index-j)))
                                (aref matrix
                                      (+ index-k
                                         (aref offset index-i))))))
                      (:= elim-col-index (+ elim-col-index 1))))
                (begin-sync
                 (:= in-sync (consume sync))
                 (if (< in-sync elim-row-num)
                     (begin
                      (:= (produce sync) (+ in-sync 1)))
                   (:= (produce sync-start) in-sync))))))))
  (begin
   (:= (uncond sync-start) 0)
   (for ((:= (uncond i) 0) (< i nrows) (:= (uncond i) (+ i 1)))
        (begin
         (:= (uncond sync) 0)
         (:= temp1 (consume sync-start))
         (:= temp1 (+ temp1 1))
         (:= sync-count 0)
         (:= (uncond diag-val)
             (/ 1 (aref matrix (+ i (aref offset i)))))
         (:= (uncond elim-row-num) (aref diag (* 4 i)))
         (:= (uncond elim-row-index) (aref diag (+ 1 (* 4 i))))
         (:= (uncond elim-col-num) (aref diag (+ 2 (* 4 i))))
         (forall-iterate
          ((:= j elim-row-num) (> j 0) (:= j (- j 1)))
          (begin
           (fork (call row-norm elim-row-index))
```

```
          (begin-sync
           (:= mini-sync (+ 1 (consume up-semaphore)))
           (:= (produce elim-row-index) (+ elim-row-index 1)))))
        (begin-sync
         (:= sync-count (consume sync))
         (if (< sync-count elim-row-num)
             (begin
              (:= (produce sync) (+ sync-count 1))
              (begin-sync
               (:= (uncond sync-count) (leave sync-start))))
           (:= (produce sync-start) temp1))))))))
```

A.4.3 thr-lud

```
(declare
 ((matrix (array 3600 float))
  (nrows int) (offset (array 110 int))
  (diag (array 500 int))
  (r-in-c (array 1000 int))
  (c-in-r (array 1000 int))
  (i int) (j int)
  (diag-val float)
  (elim-col-num int)
  (elim-row-num int) (elim-row-index int)
  (up-semaphore int) (sync int) (sync-count int)
  (temp1 int) (sync-start int)
  (mini-sync int)
  (row-norm
   (lambda (elim-row-index)
     (declare ((index-i int) (index-j int)
               (index-k int) (k int) (in-sync int)
               (temp1-index int) (temp2-index int)
               (elim-col-index int) (local-row-index int))
              (begin
               (:= local-row-index (consume elim-row-index))
               (:= (produce up-semaphore) 1)
               (:= temp2-index (aref r-in-c local-row-index))
               (:= index-i (leave i))
               (:= temp1-index (+ index-i (aref offset temp2-index)))
               (:= (aref matrix temp1-index)
                   (* (aref matrix temp1-index) diag-val))
               (:= elim-col-index (aref diag (+ 3 (* 4 index-i))))
               (:= index-j (aref r-in-c local-row-index))
               (for ((:= k (leave elim-col-num)) (> k 0) (:= k (- k 1)))
                    (begin
                     (:= index-k (aref c-in-r elim-col-index))
                     (:= (aref matrix
                               (+ index-k
                                  (aref offset index-j)))
                         (- (aref matrix
                                  (+ index-k
                                     (aref offset index-j)))
                            (* (aref matrix
                                     (+ index-i
                                        (aref offset index-j)))
```

```
                              (aref matrix
                                    (+ index-k
                                       (aref offset index-i))))))
                    (:= elim-col-index (+ elim-col-index 1))))
              (begin-sync
               (:= in-sync (consume sync))
               (if (< in-sync elim-row-num)
                   (begin
                    (:= (produce sync) (+ in-sync 1)))
                 (:= (produce sync-start) in-sync)))))))
 (begin
  (:= (uncond sync-start) 0)
  (for ((:= i 0) (< i nrows) (:= i (+ i 1)))
       (begin
        (:= (uncond sync) 0)
        (:= temp1 (consume sync-start))
        (:= temp1 (+ temp1 1))
        (:= sync-count 0)
        (:= (uncond diag-val)
            (/ 1 (aref matrix (+ i (aref offset i)))))
        (:= (uncond elim-row-num) (aref diag (* 4 i)))
        (:= (uncond elim-row-index) (aref diag (+ 1 (* 4 i))))
        (:= (uncond elim-col-num) (aref diag (+ 2 (* 4 i))))
        (for ((:= j elim-row-num) (> j 0) (:= j (- j 1)))
             (begin
              (fork (call row-norm elim-row-index))
              (begin-sync
               (:= mini-sync (+ 1 (consume up-semaphore)))
               (:= (produce elim-row-index) (+ elim-row-index 1)))))
        (begin-sync
         (:= sync-count (consume sync))
         (if (< sync-count elim-row-num)
             (begin
              (:= (produce sync) (+ sync-count 1))
              (begin-sync
               (:= (uncond sync-count) (leave sync-start))))
           (:= (produce sync-start) sync-count)))))))
```

A.4.4 thr-auto-lud

```
(declare
 ((matrix (array 3600 float))
  (nrows int) (offset (array 110 int))
  (diag (array 500 int))
  (r-in-c (array 1000 int))
  (c-in-r (array 1000 int))
  (i int) (j int)
  (diag-val float)
  (elim-col-num int)
  (elim-row-num int) (elim-row-index int)
  (num-threads int) (index int)
  (sync int) (sync-count int) (temp1 int) (sync-start int)
  (row-norm
   (lambda ()
     (declare ((index-i int) (index-j int)
```

```
                (index-k int) (k int) (in-sync int)
                (temp1-index int) (temp2-index int)
                (local-index int)
                (elim-col-index int) (local-row-index int))
               (begin
                (:= local-index (consume index))
                (:= (produce index) (+ local-index 1))
                (while (< local-index elim-row-num)
                  (begin
                   (:= local-row-index
                       (+ local-index elim-row-index))
                   (:= temp2-index (aref r-in-c local-row-index))
                   (:= index-i (leave i))
                   (:= temp1-index (+ index-i (aref offset temp2-index)))
                   (:= (aref matrix temp1-index)
                       (* (aref matrix temp1-index) diag-val))
                   (:= elim-col-index (aref diag (+ 3 (* 4 index-i))))
                   (:= index-j (aref r-in-c local-row-index))
                   (for ((:= k elim-col-num) (> k 0) (:= k (- k 1)))
                        (begin
                         (:= index-k (aref c-in-r elim-col-index))
                         (:= (aref matrix
                                   (+ index-k
                                      (aref offset index-j)))
                             (- (aref matrix
                                      (+ index-k
                                         (aref offset index-j)))
                                (* (aref matrix
                                         (+ index-i
                                            (aref offset index-j)))
                                   (aref matrix
                                         (+ index-k
                                            (aref offset index-i))))))
                         (:= elim-col-index (+ elim-col-index 1))))
                   (:= local-index (consume index))
                   (:= (produce index) (+ local-index 1))))
                (begin-sync
                 (:= in-sync (consume sync))
                 (if (< in-sync num-threads)
                     (begin
                      (:= (produce sync) (+ in-sync 1)))
                   (:= (produce sync-start) in-sync)))))))
(begin
 (:= (uncond num-threads) 4)
 (:= (uncond sync-start) 0)
 (:= (uncond index) 0)
 (for ((:= (uncond i) 0) (< i nrows) (:= (uncond i) (+ i 1)))
      (begin
       (:= (uncond sync) 0)
       (:= temp1 (consume sync-start))
       (:= temp1 (+ temp1 1))
       (:= sync-count 0)
       (:= (uncond diag-val)
           (/ 1 (aref matrix (+ i (aref offset i)))))
       (:= (uncond elim-row-num) (aref diag (* 4 i)))
       (:= (uncond elim-row-index) (aref diag (+ 1 (* 4 i))))
```

```
(:= (uncond elim-col-num) (aref diag (+ 2 (* 4 i))))
(:= (uncond index) 0)
(begin-sync
 (forall ((:= j 0) (< j 4) (:= j (+ j 1)))
         (begin (call row-norm))))
(begin-sync
 (:= sync-count (consume sync))
 (if (< sync-count num-threads)
     (begin
      (:= (produce sync) (+ sync-count 1))
      (begin-sync
       (:= (uncond sync-count) (leave sync-start))))
   (:= (produce sync-start) sync-count)))))))
```

Appendix B

Experimental Data

This appendix contains the experimental data used in the graphs seen in Chapter 6.

B.1 Baseline Results

Benchmark	Mode	#Cycles	Operation Counts FPU	IU	Memory	Branch	Total
Matrix	SEQ	1991	1377	1800	1810	101	5088
Matrix	STS	1181	1377	1800	1810	101	5088
Matrix	TPE	628	1377	1782	1800	109	5068
Matrix	Coupled	637	1377	1782	1800	109	5068
Matrix	Ideal	349	1377	99	243	1	1720
FFT	SEQ	3376	800	2070	1868	157	4895
FFT	STS	1791	800	2230	1868	157	5055
FFT	TPE	1976	800	2080	1894	397	5171
FFT	Coupled	1101	800	2240	1894	397	5331
FFT	Ideal	401	800	1021	1073	7	2901
Model	SEQ	992	212	97	811	136	1256
Model	STS	770	212	97	803	136	1248
Model	TPE	394	212	257	696	236	1401
Model	Coupled	368	212	257	668	236	1373
LUD	SEQ	57974	7985	26000	57014	4761	95760
LUD	STS	33125	7985	26000	57014	4761	95760
LUD	TPE	22626	7985	30598	61294	7954	107831
LUD	Coupled	21542	7985	30598	61294	7954	107831

Table B.1: Cycle and operation counts for the benchmarks on different machine models.

B.2 Variable Memory Latency

Benchmark	Mode	Memory Mode	#Cycles	Compared to Min	Cache Misses
Matrix	SEQ	Min	1991	1.00	0
Matrix	SEQ	Mem1	6305	3.17	97
Matrix	SEQ	Mem2	10388	5.22	180
Matrix	STS	Min	1181	1.00	0
Matrix	STS	Mem1	5142	4.35	97
Matrix	STS	Mem2	8299	7.03	180
Matrix	TPE	Min	628	1.00	0
Matrix	TPE	Mem1	1126	1.79	97
Matrix	TPE	Mem2	1397	2.22	179
Matrix	Coupled	Min	637	1.00	0
Matrix	Coupled	Mem1	953	1.50	97
Matrix	Coupled	Mem2	1113	1.75	179
Matrix	Ideal	Min	349	1.00	0
Matrix	Ideal	Mem1	459	1.32	13
Matrix	Ideal	Mem2	531	1.52	22

Table B.2: Effect of memory latency on **Matrix**.

Benchmark	Mode	Memory Mode	#Cycles	Compared to Min	Cache Misses
FFT	SEQ	Min	3376	1.00	0
FFT	SEQ	Mem1	6110	1.81	99
FFT	SEQ	Mem2	8240	2.44	184
FFT	STS	Min	1791	1.00	0
FFT	STS	Mem1	4381	2.45	99
FFT	STS	Mem2	6327	3.53	184
FFT	TPE	Min	1976	1.00	0
FFT	TPE	Mem1	2927	1.48	100
FFT	TPE	Mem2	2905	1.47	186
FFT	Coupled	Min	1101	1.00	0
FFT	Coupled	Mem1	1782	1.62	100
FFT	Coupled	Mem2	1968	1.79	186
FFT	Ideal	Min	401	1.00	0
FFT	Ideal	Mem1	2062	5.14	60
FFT	Ideal	Mem2	2997	7.47	108

Table B.3: Effect of memory latency on **FFT**.

Benchmark	Mode	Memory Mode	#Cycles	Compared to Min	Cache Misses
Model	SEQ	Min	992	1.00	0
Model	SEQ	Mem1	2398	2.42	48
Model	SEQ	Mem2	2960	2.98	80
Model	STS	Min	770	1.00	0
Model	STS	Mem1	2086	2.71	47
Model	STS	Mem2	3131	4.07	78
Model	TPE	Min	394	1.00	0
Model	TPE	Mem1	712	1.81	35
Model	TPE	Mem2	906	2.30	66
Model	Coupled	Min	368	1.00	0
Model	Coupled	Mem1	553	1.50	32
Model	Coupled	Mem2	710	1.93	60

Table B.4: Effect of memory latency on **Model**.

Benchmark	Mode	Memory Mode	#Cycles	Compared to Min	Cache Misses
LUD	SEQ	Min	57974	1.00	0
LUD	SEQ	Mem1	167220	2.88	2722
LUD	SEQ	Mem2	274366	4.73	5546
LUD	STS	Min	33125	1.00	0
LUD	STS	Mem1	141710	4.27	2722
LUD	STS	Mem2	246180	7.43	5546
LUD	TPE	Min	22626	1.00	0
LUD	TPE	Mem1	49983	2.21	2944
LUD	TPE	Mem2	72348	3.20	5995
LUD	Coupled	Min	21542	1.00	0
LUD	Coupled	Mem1	40258	1.87	2944
LUD	Coupled	Mem2	55846	2.59	5995

Table B.5: Effect of memory latency on **LUD**.

B.3 Effect of FPU Latency

Benchmark	Mode	FPU Latency	#Cycles	Compared to 1 Cycle Latency
Matrix	SEQ	1	1991	1.00
Matrix	SEQ	2	2234	1.12
Matrix	SEQ	3	2558	1.28
Matrix	SEQ	4	3206	1.61
Matrix	SEQ	5	3854	1.94
Matrix	STS	1	1181	1.00
Matrix	STS	2	1910	1.62
Matrix	STS	3	2072	1.75
Matrix	STS	4	2720	2.30
Matrix	STS	5	3449	2.92
Matrix	TPE	1	628	1.00
Matrix	TPE	2	663	1.06
Matrix	TPE	3	705	1.12
Matrix	TPE	4	753	1.20
Matrix	TPE	5	782	1.25
Matrix	Coupled	1	637	1.00
Matrix	Coupled	2	655	1.03
Matrix	Coupled	3	670	1.05
Matrix	Coupled	4	718	1.13
Matrix	Coupled	5	778	1.22
Matrix	Ideal	1	349	1.00
Matrix	Ideal	2	351	1.01
Matrix	Ideal	3	354	1.01
Matrix	Ideal	4	356	1.02
Matrix	Ideal	5	361	1.03

Table B.6: Effect of floating point unit latency on **Matrix**.

Benchmark	Mode	FPU Latency	#Cycles	Compared to 1 Cycle Latency
FFT	SEQ	1	3376	1.00
FFT	SEQ	2	3456	1.02
FFT	SEQ	3	3616	1.07
FFT	SEQ	4	3856	1.14
FFT	SEQ	5	3936	1.17
FFT	STS	1	1791	1.00
FFT	STS	2	2031	1.13
FFT	STS	3	2031	1.13
FFT	STS	4	2271	1.27
FFT	STS	5	2511	1.40
FFT	TPE	1	1976	1.00
FFT	TPE	2	1986	1.01
FFT	TPE	3	1991	1.01
FFT	TPE	4	2006	1.02
FFT	TPE	5	2031	1.03
FFT	Coupled	1	1101	1.00
FFT	Coupled	2	1106	1.00
FFT	Coupled	3	1106	1.00
FFT	Coupled	4	1136	1.03
FFT	Coupled	5	1141	1.04
FFT	Ideal	1	401	1.00
FFT	Ideal	2	406	1.01
FFT	Ideal	3	396	0.99
FFT	Ideal	4	416	1.04
FFT	Ideal	5	431	1.07

Table B.7: Effect of floating point unit latency on **FFT**.

Benchmark	Mode	FPU Latency	#Cycles	Compared to 1 Cycle Latency
Model	SEQ	1	992	1.00
Model	SEQ	2	1102	1.11
Model	SEQ	3	1120	1.13
Model	SEQ	4	1277	1.29
Model	SEQ	5	1423	1.43
Model	STS	1	770	1.00
Model	STS	2	889	1.15
Model	STS	3	948	1.23
Model	STS	4	1078	1.40
Model	STS	5	1239	1.61
Model	TPE	1	394	1.00
Model	TPE	2	400	1.02
Model	TPE	3	400	1.02
Model	TPE	4	411	1.04
Model	TPE	5	427	1.08
Model	Coupled	1	368	1.00
Model	Coupled	2	374	1.02
Model	Coupled	3	382	1.04
Model	Coupled	4	388	1.05
Model	Coupled	5	387	1.05

Table B.8: Effect of floating point unit latency on **Model**.

Benchmark	Mode	FPU Latency	#Cycles	Compared to 1 Cycle Latency
LUD	SEQ	1	57974	1.00
LUD	SEQ	2	61711	1.06
LUD	SEQ	3	65448	1.13
LUD	SEQ	4	69185	1.19
LUD	SEQ	5	76659	1.32
LUD	STS	1	33125	1.00
LUD	STS	2	40599	1.23
LUD	STS	3	40599	1.23
LUD	STS	4	48584	1.47
LUD	STS	5	56569	1.71
LUD	TPE	1	22626	1.00
LUD	TPE	2	23045	1.02
LUD	TPE	3	23568	1.04
LUD	TPE	4	23905	1.06
LUD	TPE	5	24877	1.10
LUD	Coupled	1	21542	1.00
LUD	Coupled	2	21527	1.00
LUD	Coupled	3	21591	1.00
LUD	Coupled	4	22246	1.03
LUD	Coupled	5	23234	1.08

Table B.9: Effect of floating point unit latency on **LUD**.

B.4 Restricting Communication

Benchmark	Mode	Communication Scheme	#Cycles	Compared to Full	Bus Conflicts
Matrix	SEQ	Full	1991	1.00	0
Matrix	SEQ	Tri-Port	1991	1.00	0
Matrix	SEQ	Dual-Port	3206	1.61	1296
Matrix	SEQ	Single-Port	4826	2.42	4293
Matrix	SEQ	Single-Bus	3206	1.61	1296
Matrix	STS	Full	1181	1.00	0
Matrix	STS	Tri-Port	1424	1.21	243
Matrix	STS	Dual-Port	1748	1.48	1053
Matrix	STS	Single-Port	2963	2.51	3888
Matrix	STS	Single-Bus	2234	1.89	2187
Matrix	TPE	Full	628	1.00	0
Matrix	TPE	Tri-Port	628	1.00	0
Matrix	TPE	Dual-Port	1025	1.63	1456
Matrix	TPE	Single-Port	1586	2.53	4644
Matrix	TPE	Single-Bus	1385	2.21	3352
Matrix	Coupled	Full	637	1.00	0
Matrix	Coupled	Tri-Port	694	1.09	497
Matrix	Coupled	Dual-Port	1063	1.67	1992
Matrix	Coupled	Single-Port	1550	2.43	4760
Matrix	Coupled	Single-Bus	2096	3.29	5090
Matrix	Ideal	Full	349	1.00	0
Matrix	Ideal	Tri-Port	404	1.16	155
Matrix	Ideal	Dual-Port	527	1.51	528
Matrix	Ideal	Single-Port	696	1.99	917
Matrix	Ideal	Single-Bus	1056	3.03	2324

Table B.10: Effect of restricting communication on **Matrix**.

Benchmark	Mode	Communication Scheme	#Cycles	Compared to Full	Bus Conflicts
FFT	SEQ	Full	3376	1.00	0
FFT	SEQ	Tri-Port	3456	1.02	80
FFT	SEQ	Dual-Port	3696	1.09	320
FFT	SEQ	Single-Port	4322	1.28	866
FFT	SEQ	Single-Bus	3696	1.09	160
FFT	STS	Full	1791	1.00	0
FFT	STS	Tri-Port	1791	1.00	0
FFT	STS	Dual-Port	2027	1.13	428
FFT	STS	Single-Port	3320	1.85	1286
FFT	STS	Single-Bus	3103	1.73	968
FFT	TPE	Full	1976	1.00	0
FFT	TPE	Tri-Port	1986	1.01	145
FFT	TPE	Dual-Port	2086	1.06	560
FFT	TPE	Single-Port	2367	1.20	2216
FFT	TPE	Single-Bus	2201	1.11	1435
FFT	Coupled	Full	1101	1.00	0
FFT	Coupled	Tri-Port	1181	1.07	505
FFT	Coupled	Dual-Port	1577	1.43	1678
FFT	Coupled	Single-Port	1910	1.73	2786
FFT	Coupled	Single-Bus	2348	2.13	3973
FFT	Ideal	Full	401	1.00	0
FFT	Ideal	Tri-Port	546	1.36	325
FFT	Ideal	Dual-Port	796	1.99	1060
FFT	Ideal	Single-Port	1641	4.09	1770
FFT	Ideal	Single-Bus	1316	3.28	1105

Table B.11: Effect of restricting communication on **FFT**.

Benchmark	Mode	Communication Scheme	#Cycles	Compared to Full	Bus Conflicts
Model	SEQ	Full	992	1.00	0
Model	SEQ	Tri-Port	992	1.00	0
Model	SEQ	Dual-Port	992	1.00	0
Model	SEQ	Single-Port	1069	1.08	77
Model	SEQ	Single-Bus	992	1.00	0
Model	STS	Full	770	1.00	0
Model	STS	Tri-Port	770	1.00	0
Model	STS	Dual-Port	775	1.01	5
Model	STS	Single-Port	860	1.12	144
Model	STS	Single-Bus	787	1.02	40
Model	TPE	Full	394	1.00	0
Model	TPE	Tri-Port	394	1.00	3
Model	TPE	Dual-Port	395	1.00	62
Model	TPE	Single-Port	429	1.09	169
Model	TPE	Single-Bus	398	1.01	206
Model	Coupled	Full	368	1.00	0
Model	Coupled	Tri-Port	370	1.01	20
Model	Coupled	Dual-Port	402	1.09	140
Model	Coupled	Single-Port	437	1.19	358
Model	Coupled	Single-Bus	437	1.19	739

Table B.12: Effect of restricting communication on **Model**.

Benchmark	Mode	Communication Scheme	#Cycles	Compared to Full	Bus Conflicts
LUD	SEQ	Full	57974	1.00	0
LUD	SEQ	Tri-Port	57974	1.00	0
LUD	SEQ	Dual-Port	57974	1.00	64
LUD	SEQ	Single-Port	81610	1.41	23828
LUD	SEQ	Single-Bus	61711	1.06	3865
LUD	STS	Full	33125	1.00	0
LUD	STS	Tri-Port	33125	1.00	0
LUD	STS	Dual-Port	37309	1.13	4184
LUD	STS	Single-Port	49798	1.50	16737
LUD	STS	Single-Bus	48648	1.47	26415
LUD	TPE	Full	22626	1.00	0
LUD	TPE	Tri-Port	22621	1.00	27
LUD	TPE	Dual-Port	22848	1.01	1491
LUD	TPE	Single-Port	29095	1.29	31184
LUD	TPE	Single-Bus	26353	1.16	20578
LUD	Coupled	Full	21542	1.00	0
LUD	Coupled	Tri-Port	21519	1.00	346
LUD	Coupled	Dual-Port	23792	1.10	8147
LUD	Coupled	Single-Port	33244	1.54	36558
LUD	Coupled	Single-Bus	35484	1.65	56725

Table B.13: Effect of restricting communication on **LUD**.

B.5 Number and Mix of Function Units

Benchmark	Mode	Number FPUs	Number IUs	#Cycles	Compared to Full
Matrix	Coupled	1	1	1812	2.84
Matrix	Coupled	1	2	1432	2.25
Matrix	Coupled	1	3	1432	2.25
Matrix	Coupled	1	4	1431	2.25
Matrix	Coupled	2	1	1812	2.84
Matrix	Coupled	2	2	939	1.47
Matrix	Coupled	2	3	772	1.21
Matrix	Coupled	2	4	759	1.19
Matrix	Coupled	3	1	1812	2.84
Matrix	Coupled	3	2	943	1.48
Matrix	Coupled	3	3	697	1.09
Matrix	Coupled	3	4	666	1.05
Matrix	Coupled	4	1	1812	2.84
Matrix	Coupled	4	2	944	1.48
Matrix	Coupled	4	3	706	1.11
Matrix	Coupled	4	4	637	1.00

Table B.14: Effect of varying the number and mix of units for **Matrix**.

Benchmark	Mode	Number FPUs	Number IUs	#Cycles	Compared to Full
FFT	Coupled	1	1	2605	2.37
FFT	Coupled	1	2	1834	1.67
FFT	Coupled	1	3	1642	1.49
FFT	Coupled	1	4	1546	1.40
FFT	Coupled	2	1	2590	2.35
FFT	Coupled	2	2	1554	1.41
FFT	Coupled	2	3	1312	1.19
FFT	Coupled	2	4	1146	1.04
FFT	Coupled	3	1	2590	2.35
FFT	Coupled	3	2	1564	1.42
FFT	Coupled	3	3	1312	1.19
FFT	Coupled	3	4	1116	1.01
FFT	Coupled	4	1	2735	2.48
FFT	Coupled	4	2	1634	1.48
FFT	Coupled	4	3	1337	1.21
FFT	Coupled	4	4	1101	1.00

Table B.15: Effect of varying the number and mix of units for **FFT**.

Benchmark	Mode	Number FPUs	Number IUs	#Cycles	Compared to Full
Model	Coupled	1	1	409	1.11
Model	Coupled	1	2	388	1.05
Model	Coupled	1	3	390	1.06
Model	Coupled	1	4	406	1.10
Model	Coupled	2	1	387	1.05
Model	Coupled	2	2	380	1.03
Model	Coupled	2	3	373	1.01
Model	Coupled	2	4	372	1.01
Model	Coupled	3	1	381	1.04
Model	Coupled	3	2	373	1.01
Model	Coupled	3	3	373	1.01
Model	Coupled	3	4	368	1.00
Model	Coupled	4	1	382	1.04
Model	Coupled	4	2	367	1.00
Model	Coupled	4	3	373	1.01
Model	Coupled	4	4	368	1.00

Table B.16: Effect of varying the number and mix of units for **Model**.

Benchmark	Mode	Number FPUs	Number IUs	#Cycles	Compared to Full
LUD	Coupled	1	1	33239	1.54
LUD	Coupled	1	2	24213	1.12
LUD	Coupled	1	3	23015	1.07
LUD	Coupled	1	4	23401	1.09
LUD	Coupled	2	1	34018	1.58
LUD	Coupled	2	2	24650	1.14
LUD	Coupled	2	3	22921	1.06
LUD	Coupled	2	4	21982	1.02
LUD	Coupled	3	1	34018	1.58
LUD	Coupled	3	2	25513	1.18
LUD	Coupled	3	3	22416	1.04
LUD	Coupled	3	4	21892	1.02
LUD	Coupled	4	1	39125	1.82
LUD	Coupled	4	2	28002	1.30
LUD	Coupled	4	3	23777	1.10
LUD	Coupled	4	4	21542	1.00

Table B.17: Effect of varying the number and mix of units for **LUD**.

B.6 Methods of Expressing Parallel Loops

Benchmark	Mode	Parallelism Type	#Cycles	Compared to Fork	Operation Count
Model	TPE	Fork	586	1.00	1161
Model	TPE	Iter	394	0.67	1401
Model	TPE	Auto	282	0.48	1283
Model	Coupled	Fork	355	1.00	1133
Model	Coupled	Iter	368	1.04	1373
Model	Coupled	Auto	304	0.86	1272
LUD	TPE	Fork	58040	1.00	102402
LUD	TPE	Iter	22626	0.39	107831
LUD	TPE	Auto	19381	0.33	100946
LUD	Coupled	Fork	21235	1.00	102402
LUD	Coupled	Iter	21542	1.01	107831
LUD	Coupled	Auto	19552	0.92	100946

Table B.18: Cycle counts using three different parallel loops for **Model** and **LUD**.

Benchmark	Mode	Parallelism Type	Memory Mode	#Cycles	Compared to Min	Cache Misses
Model	TPE	Fork	Min	586	1.00	0
Model	TPE	Fork	Mem1	803	1.37	32
Model	TPE	Fork	Mem2	1122	1.91	58
Model	TPE	Iter	Min	394	1.00	0
Model	TPE	Iter	Mem1	712	1.81	35
Model	TPE	Iter	Mem2	906	2.30	66
Model	TPE	Auto	Min	282	1.00	0
Model	TPE	Auto	Mem1	734	2.60	49
Model	TPE	Auto	Mem2	873	3.10	80
Model	Coupled	Fork	Min	355	1.00	0
Model	Coupled	Fork	Mem1	445	1.25	32
Model	Coupled	Fork	Mem2	676	1.90	56
Model	Coupled	Iter	Min	368	1.00	0
Model	Coupled	Iter	Mem1	553	1.50	32
Model	Coupled	Iter	Mem2	710	1.93	60
Model	Coupled	Auto	Min	304	1.00	0
Model	Coupled	Auto	Mem1	698	2.30	48
Model	Coupled	Auto	Mem2	953	3.13	80

Table B.19: Effect of memory latency on **Model** using different parallel loops.

Benchmark	Mode	Parallelism Type	Memory Mode	#Cycles	Compared to Min	Cache Misses
LUD	TPE	Fork	Min	58040	1.00	0
LUD	TPE	Fork	Mem1	75312	1.30	2892
LUD	TPE	Fork	Mem2	90038	1.55	5893
LUD	TPE	Iter	Min	22626	1.00	0
LUD	TPE	Iter	Mem1	49983	2.21	2944
LUD	TPE	Iter	Mem2	72348	3.20	5995
LUD	TPE	Auto	Min	19381	1.00	0
LUD	TPE	Auto	Mem1	56397	2.91	2872
LUD	TPE	Auto	Mem2	89996	4.64	5838
LUD	Coupled	Fork	Min	21235	1.00	0
LUD	Coupled	Fork	Mem1	37531	1.77	2892
LUD	Coupled	Fork	Mem2	52308	2.46	5893
LUD	Coupled	Iter	Min	21542	1.00	0
LUD	Coupled	Iter	Mem1	40258	1.87	2944
LUD	Coupled	Iter	Mem2	55846	2.59	5995
LUD	Coupled	Auto	Min	19552	1.00	0
LUD	Coupled	Auto	Mem1	50499	2.58	2872
LUD	Coupled	Auto	Mem2	82494	4.22	5838

Table B.20: Effect of memory latency on **LUD** using different parallel loops.

Benchmark	Mode	Parallelism Type	Communication Scheme	#Cycles	Compared to Full	Bus Conflicts
Model	TPE	Fork	Full	586	1.00	0
Model	TPE	Fork	Tri-Port	586	1.00	0
Model	TPE	Fork	Dual-Port	585	1.00	15
Model	TPE	Fork	Single-Port	622	1.06	399
Model	TPE	Fork	Single-Bus	584	1.00	20
Model	TPE	Iter	Full	394	1.00	0
Model	TPE	Iter	Tri-Port	394	1.00	3
Model	TPE	Iter	Dual-Port	395	1.00	62
Model	TPE	Iter	Single-Port	429	1.09	169
Model	TPE	Iter	Single-Bus	398	1.01	206
Model	TPE	Auto	Full	282	1.00	0
Model	TPE	Auto	Tri-Port	282	1.00	0
Model	TPE	Auto	Dual-Port	294	1.04	14
Model	TPE	Auto	Single-Port	299	1.06	70
Model	TPE	Auto	Single-Bus	301	1.07	40
Model	Coupled	Fork	Full	355	1.00	0
Model	Coupled	Fork	Tri-Port	357	1.01	2
Model	Coupled	Fork	Dual-Port	367	1.03	52
Model	Coupled	Fork	Single-Port	507	1.43	322
Model	Coupled	Fork	Single-Bus	433	1.22	134
Model	Coupled	Iter	Full	368	1.00	0
Model	Coupled	Iter	Tri-Port	370	1.01	20
Model	Coupled	Iter	Dual-Port	402	1.09	140
Model	Coupled	Iter	Single-Port	437	1.19	358
Model	Coupled	Iter	Single-Bus	437	1.19	739
Model	Coupled	Auto	Full	304	1.00	0
Model	Coupled	Auto	Tri-Port	304	1.00	2
Model	Coupled	Auto	Dual-Port	302	0.99	36
Model	Coupled	Auto	Single-Port	356	1.17	204
Model	Coupled	Auto	Single-Bus	374	1.23	266

Table B.21: Restricting communication on **Model** using different parallel loops.

Benchmark	Mode	Parallelism Type	Communication Scheme	#Cycles	Compared to Full	Bus Conflicts
LUD	TPE	Fork	Full	58040	1.00	0
LUD	TPE	Fork	Tri-Port	58040	1.00	0
LUD	TPE	Fork	Dual-Port	58424	1.01	384
LUD	TPE	Fork	Single-Port	78628	1.35	37598
LUD	TPE	Fork	Single-Bus	58069	1.00	3820
LUD	TPE	Iter	Full	22626	1.00	0
LUD	TPE	Iter	Tri-Port	22621	1.00	27
LUD	TPE	Iter	Dual-Port	22848	1.01	1491
LUD	TPE	Iter	Single-Port	29095	1.29	31184
LUD	TPE	Iter	Single-Bus	26353	1.16	20578
LUD	TPE	Auto	Full	19381	1.00	0
LUD	TPE	Auto	Tri-Port	19381	1.00	0
LUD	TPE	Auto	Dual-Port	19517	1.01	581
LUD	TPE	Auto	Single-Port	26912	1.39	24719
LUD	TPE	Auto	Single-Bus	24301	1.25	20767
LUD	Coupled	Fork	Full	21235	1.00	0
LUD	Coupled	Fork	Tri-Port	21336	1.00	522
LUD	Coupled	Fork	Dual-Port	24479	1.15	6124
LUD	Coupled	Fork	Single-Port	42046	1.98	47745
LUD	Coupled	Fork	Single-Bus	36030	1.70	26518
LUD	Coupled	Iter	Full	21542	1.00	0
LUD	Coupled	Iter	Tri-Port	21519	1.00	346
LUD	Coupled	Iter	Dual-Port	23792	1.10	8147
LUD	Coupled	Iter	Single-Port	33244	1.54	36558
LUD	Coupled	Iter	Single-Bus	35484	1.65	56725
LUD	Coupled	Auto	Full	19552	1.00	0
LUD	Coupled	Auto	Tri-Port	19590	1.00	352
LUD	Coupled	Auto	Dual-Port	20881	1.07	7098
LUD	Coupled	Auto	Single-Port	30430	1.56	35410
LUD	Coupled	Auto	Single-Bus	34713	1.78	63291

Table B.22: Restricting communication on **LUD** using different parallel loops.

B.7 Data Movement Strategy

Benchmark	Mode	Move Units	Target Registers	#Cycles	Compared to 4 Targets	Number of Move Operations
Matrix	Coupled	yes	1	581	1.02	324
Matrix	Coupled	yes	2	595	1.04	162
Matrix	Coupled	yes	3	557	0.98	0
Matrix	Coupled	yes	4	571	1.00	0
Matrix	Coupled	no	1	578	1.01	81
Matrix	Coupled	no	2	637	1.12	0
Matrix	Coupled	no	3	557	0.98	0
Matrix	Coupled	no	4	571	1.00	0
FFT	Coupled	yes	1	1128	1.02	384
FFT	Coupled	yes	2	1074	0.97	192
FFT	Coupled	yes	3	1042	0.94	0
FFT	Coupled	yes	4	1106	1.00	0
FFT	Coupled	no	1	1246	1.13	252
FFT	Coupled	no	2	1101	1.00	160
FFT	Coupled	no	3	1042	0.94	0
FFT	Coupled	no	4	1106	1.00	0
Model	Coupled	yes	1	384	1.03	40
Model	Coupled	yes	2	379	1.01	20
Model	Coupled	yes	3	374	1.00	0
Model	Coupled	yes	4	374	1.00	0
Model	Coupled	no	1	381	1.02	20
Model	Coupled	no	2	368	0.99	0
Model	Coupled	no	3	372	1.00	0
Model	Coupled	no	4	372	1.00	0
LUD	Coupled	yes	1	25494	1.18	4695
LUD	Coupled	yes	2	21783	1.01	447
LUD	Coupled	yes	3	21719	1.00	0
LUD	Coupled	yes	4	21622	1.00	0
LUD	Coupled	no	1	24717	1.15	3737
LUD	Coupled	no	2	21542	1.00	0
LUD	Coupled	no	3	21467	1.00	0
LUD	Coupled	no	4	21502	1.00	0

Table B.23: Effect of different data transfer strategies.

Bibliography

[AC86] Arvind and David E. Culler. Dataflow architectures. *Annual Reviews in Computer Science*, 1:225–53, February 1986.

[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.

[AHU83] Alfred V. Aho, John E. Hopcroft, and Jeffrey D. Ullman. *Data Structures and Algorithms*. Addison Wesley, 1983.

[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. ACM, 1990.

[AN+92] Fuad Abu-Nofal et al. A three-million-transistor microprocessor. In *Proceedings of the IEEE International Solid-State Circuits Conference*, pages 108–109, 1992.

[ASU88] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques and Tools*. Addison Wesley, 1988.

[CHJ+90] Robert P. Colwell, W. Eric Hall, Chandra S. Joshi, David B. Papworth, Paul K. Rodman, and James E. Tornes. Architecture and implementation of a VLIW supercomputer. In *Proceedings of Supercomputing '90*, pages 910–919. IEEE Computer Society Press, November 1990.

[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.

[CSS+91] David E. Culler, Anurag Sah, Klaus Erik Schauser, Thorsten von Eicken, and John Wawrzynek. Fine-grain parallelism with minimal hardware support: A compiler-controlled threaded abstract machine. In *Proceedings of the Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 164–175. ACM Press, April 1991.

[DT91] George E. Daddis and H. C. Torng. The concurrent execution of multiple instruction streams on superscalar processors. In *Proceedings of the 1991 International Conference on Parallel Processing*, pages 76–83, August 1991.

[Ell86] John R. Ellis. *Bulldog: A Compiler for VLIW Architectures.* MIT Press, Cambridge, MA, 1986.

[FR91] Joseph A. Fisher and B. Ramakrishna Rau. Instruction-level parallel processing. *Science*, 253:1233–1241, September 1991.

[GM84] Paul R. Gray and Robert G. Meyer. *Analysis and Design of Analog Integrated Circuits.* John Wiley and Sons, second edition, 1984.

[GW89] Anoop Gupta and Wolf-Dietrich Weber. Exploring the benefits of multiple hardware contexts in a multiprocessor architecture: preliminary results. In *Proceedings of the 16th Annual International Symposium on Computer Architecture*, pages 273–280. IEEE, May 1989.

[HF88] Robert H. Halstead and Tetsuya Fujita. MASA: A multithreaded processor architecture for parallel symbolic computing. In *Proceedings of the 15th Annual International Symposium on Computer Architecture*, pages 443–451. IEEE, 1988.

[Ian88] Robert A. Ianucci. Toward a dataflow/Von Neumann hybrid architecture. In *Proceedings of the 15th Annual International Symposium on Computer Architecture*, pages 131–140. IEEE, 1988.

[Joh91] William M. Johnson. *Superscalar Microprocessor Design.* Prentice Hall, Englewood Cliffs, NJ, 1991.

[JW89] Norman P. Jouppi and David W. Wall. Available instruction-level parallelism for superscalar and superpipelined machines. In *Proceedings of the Third International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 272–282. ACM Press, April 1989.

[Kec92a] Stephen W. Keckler. ISC: Instruction Scheduling Compiler listing. Concurrent VLSI Architecture Memo 43, Massachusetts Institute of Technology, Artificial Intelligence Laboratory, May 1992.

[Kec92b] Stephen W. Keckler. PCS: Processor Coupling Simulator listing. Concurrent VLSI Architecture Memo 42, Massachusetts Institute of Technology, Artificial Intelligence Laboratory, May 1992.

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

[MC80] Carver Mead and Lynn Conway. *Introduction to VLSI Systems.* Addison Wesley, 1980.

[ND91] Peter R. Nuth and William J. Dally. A mechanism for efficient context switching. In *Proceedings of the International Conference on Computer Design*, pages 301–304. IEEE, October 1991.

[NF84] Alexandru Nicolau and Joseph A. Fisher. Measuring the parallelism available for very long instruction word architectures. *IEEE Transactions on Computers*, C-33(11):968–976, November 1984.

[OS75] Alan V. Oppenheim and Ronald W. Schafer. *Digital Signal Processing.* Prentice Hall, Englewood Cliffs, NJ, 1975.

[PSS+91] Val Popescu, Merle Schultz, John Spracklen, Gary Gibson, Bruce Lightner, and David Isaman. The Metaflow architecture. *IEEE Micro*, June 1991.

[Smi81] Burton J. Smith. Architecture and applications of the HEP multiprocessor computer system. *SPIE*, 298:241–248, 1981.

[Ste90] Guy L. Steele. *Common Lisp.* Digital Press, second edition, 1990.

[Str87] Bjarne Stroustrup. *The C++ Programming Language.* Addison Wesley, 1987.

[SV88] P. Sadayappan and V. Visvanathan. Circuit simulation on shared-memory multiprocessors. *IEEE Transactions on Computers*, 37(12):1634–1642, December 1988.

[SV89] P. Sadayappan and V. Visvanathan. Efficient sparse matrix factorization for circuit simulation on vector supercomputers. *IEEE Transactions on Computer Aided Design*, 8(12):1276–1285, December 1989.

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

[VS83] Jiri Vlach and Kishore Singhal. *Computer Methods for Circuit Analysis and Design.* Van Nostrand Reinhold Company, 1983.

[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 Press, April 1991.

[WS91] Andrew Wolfe and John P. Shen. A variable instruction stream extension to the VLIW architecture. In *Proceedings of the Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 2–14. ACM Press, April 1991.

[YTL87] Pen-Chung Yew, Nian-Feng Tzeng, and Duncan H. Lawrie. Distributing hotspot addressing in large-scale multiprocessors. *IEEE Transactions on Computers*, 36(4):388–395, April 1987.