UNIVERSITY OF JOENSUU
COMPUTER SCIENCE
DISSERTATIONS 2

MARTTI FORSELL

IMPLEMENTATION OF INSTRUCTION-LEVEL AND THREAD-LEVEL PARALLELISM IN COMPUTERS

ACADEMIC DISSERTATION

To be presented, with the permission of the Faculty of Science of the University of Joensuu, for public criticism in Auditorium M1 of the University, Yliopistonkatu 7, Joensuu, on October 10th, 1997, at 12 noon.

UNIVERSITY OF JOENSUU
1997

Julkaisija	Joensuun Yliopisto
Publisher	University of Joensuu
Toimittaja	FT Martti Penttonen
Editor	Dr Martti Penttonen
Vaihto	Joensuun yliopiston kirjasto, vaihdot
	PL 107, 80101 Joensuu
	Puh. 013-251 2672, fax 013-251 2691
Exchange	Joensuu University Library, exchange
	P.O. Box 107, FIN-80101 Joensuu, Finland
	Fax + 358 13 251 2691
Myynti	Joensuun yliopiston kirjasto, julkaisujen myynti
	PL 107, 80101 Joensuu
	puh. 013-251 2652, 251 2662, fax 013-251 2691
	Email: lavikainen@joyl.joensuu.fi
Sale	Joensuu University Library, sale of publications
	P.O. Box 107, FIN-80101 Joensuu, Finland
	fax + 358 13 251 2691
	Email: lavikainen@joyl.joensuu.fi

ISSN 1238-6944
ISBN 951-708-557-5
UDK 681.3.02
Computing Reviews (1991) Classification: C.1.2
Yliopistopaino
Joensuu 1997

IMPLEMENTATION OF INSTRUCTION-LEVEL AND THREAD-LEVEL PARALLELISM IN COMPUTERS

Martti Forsell

Department of Computer Science
University of Joensuu
P.O. Box 111, FIN-80101 Joensuu, Finland
Martti.Forsell@cs.joensuu.fi

University of Joensuu, Computer Science, Dissertations 2
Joensuu, October 1997, 121 pages
ISSN 1238-6944, ISBN 951-708-557-5

Keywords: computer architecture, pipelining, VLIW, parallelism, shared memory, PRAM, multithreading, interconnection network

There are many theoretical and practical problems that should be solved before parallel computing can become the mainstream of computing technology. Among them are problems caused by architectures originally designed for sequential computing—the low utilization of functional units due to the false dependencies between instructions, inefficient message passing due to the send, receive and thread switch overheads, undeterministic operation due to the dynamic scheduling of instructions, and the long memory access delays due to the high memory system latency. In this thesis we try to eliminate these problems by designing new processor, communication and memory system architectures with parallel computation in mind. As a result, we outline a parallel computer architecture, which uses a theoretically elegant shared memory programming model. The obvious VLSI implementation of a large machine using such a shared memory is shown impossible with current technology. There exists, however, an indirect implementation—one can simulate a machine using a shared memory by a machine using a physically distributed memory. Our proposal for such an indirect implementation—Instruction-Level Parallel Shared-Memory Architecture (IPSM)—combines elements from both instruction-level parallelism and thread-level parallelism. IPSM features a static VLIW-style scheduling of instructions and the absence of message passing and thread switch overheads. In the execution of parallel programs, IPSM exploits parallel slackness to hide the high memory system latency, and interthread instruction-level parallelism to eliminate delays caused by the dependencies between instructions belonging to a single thread. In the execution of sequential programs, IPSM uses minimal pipelining to minimize the delays caused by the dependencies between instructions.

PREFACE

My interest in processor architectures began in the early 80's. At those days home computers were slow with memory hungry high-level language interpreters. Assembly language programming seemed to give almost endless speedup possibilities for simple game programs.

The introduction of powerful 32-bit CISC microprocessors, like Motorola MC68020, indicated that the architecture of the 8-bit processors used in home computers was far from perfect. At the same time the discussion between RISC and CISC instruction-set strategies revealed architectural weaknesses also in the CISC processors.

In the beginning of 90's I became familiar with the theory of parallel computation. It became evident, that parallel computers potentially provide better performance than sequential ones in most computational problems. Since then I have had a opportunity of applying my interest in computer architectures also to parallel architectures.

This thesis is the result of the research I carried out in 1992-1996 at the Department of Computer Science, University of Joensuu under the supervision of Martti Penttonen and Jyrki Katajainen. It consists of general introduction and four individual articles.

God bless you,

Martti Forsell

Kaiken tämän olen viisauden avulla koetellut. Minä sanoin: "Tahdon tulla viisaaksi." Mutta viisaus pysytteli tavoittamattomissa.

Kaukana on kaiken sisin olemus, syvällä, syvällä—kuka voi sen löytää?

Minä ryhdyin itsekseni pohtimaan ja etsimään viisautta ja kaiken lopputulosta ja ymmärsin, että jumalattomuus johtuu järjettömyydestä ja tyhmyys mielen sokaistumisesta.

From the Finnish translation of the Bible

ACKNOWLEDGEMENTS

I would like to thank Martti Penttonen, Ville Leppänen, Simo Juvaste and Jyrki Katajainen for great many conversations and valuable co-operation. I have had a good time as a member of the parallel computation research group "rinnakkaisjengi" in University of Joensuu.

Professors Daniel Litaize and Iiro Hartimo kindly accepted the role of a reviewer. I wish to thank for their time and efforts.

Then warm thanks to Coca-Cola Company and Apple Computer, Inc. Without your products—namely Coke® and Macintosh™, this thesis would never have been completed.

I also want to express my gratitude to my beloved wife Marjut for sharing life with me. Marjut, I am happy with you after being so many years alone.

This research was funded by the University of Joensuu and the Finnish Academy.

Table of Contents

List of abbreviations:

ADR	= Address Unit
ALU	= Arithmetic and Logical Unit
CBM	= Coated Block Mesh
CFPP	= Counter Flow Pipeline Processor Architecture
CISC	= Complex Instruction-Set Computer
CM	= Coated Mesh
CMP	= Compare Unit
CPU	= Central Processing Unit
CRCW	= Concurrent-Read Concurrent-Write
CREW	= Concurrent-Read Exclusive-Write
DAG	= Directed Acyclic Graph
DINC	= Decoded Instruction Cache
DMM	= Distributed Memory Machine
EREW	= Exclusive-Read Exclusive-Write
FIFO	= First-In First-Out
FU	= Functional Unit
IPSM	= Instruction-Level Parallel Shared-Memory Architecture
MPA	= Minimal Pipeline Architecture
MTAC	= Multithreaded Architecture with Chaining
MU	= Memory Unit
PRAM	= Parallel Random Access Machine
RAM	= Random Access Machine
RAM	= Random Access Memory
REG	= Register Unit
RISC	= Reduced Instruction-Set Computer
SEQ	= Sequencer
SMM	= Shared Memory Machine
SRAM	= Static Random Access Memory
VLIW	= Very Long Instruction Word
VLSI	= Very Large Scale Integrated

List of original publications included in this thesis:

M. Forsell, Are Multiport Memories Physically Feasible? *Computer Architecture News* **22**, 4 (1994), 47-54.

M. Forsell, V. Leppänen and M.Penttonen, Efficient Two-Level Mesh based Simulation of PRAMs, *Proceedings of the International Symposium on Parallel Architectures*, Algorithms, and Networks, June 12-14, 1996, Beijing, China, 29-35.

M. Forsell, Minimal Pipeline Architecture—an Alternative to Superscalar Architecture, *Microprocessors and Microsystems* **20**, 5 (1996), 277-284.

M. Forsell, MTAC—A Multithreaded VLIW Architecture for PRAM Simulation, to be published in *Journal of Universal Computer Science* (http://cs.joensuu.fi:8080/jucs_root), 1997.

Reprinted with permission from the publishers.

Note: The chapters 2,3,4, and 5 of this thesis are directly based on the four original articles. Only the lay-out, the referencing convention, and the numbering of sections, figures, and tables were unified during the inclusion of the articles for this thesis.

Chapter 1

Introduction

When Silicon Valley wants to look good, it measures itself against Detroit. The comparison goes like this: If automotive technology had kept pace with computer technology over the past few decades, you would now be driving a V-32 instead of a V-8, and it would have a top speed of 10,000 miles per hour. Or you could have an economy car that weighs 30 pounds and gets a thousand miles to a gallon of gas. In either case the sticker price of a new car would be less than $50.

In response to all this goading, Detroit grumbles: Yes, but would you really want to drive a car that crashes twice a day?

—Brian Hayes, A Computer with Its Head Cut Off,
American Scientist, 03 (1995)

Since computer was invented in the 40's, computer programmers and users have been requesting faster computers to solve larger and more complex computational tasks. This need for more computing power is going to be endless, because there will always be problems that any computer cannot solve fast enough. Consider for example the simulation of the whole universe from the big bang to a moment when sun runs out of hydrogen with a computer by calculating every move of every nucleon of the universe.

The request for more powerful computers has been, however, realized amazingly well. The performance of computers has been doubled in every second year during the last five decades. This has been the result of faster and smaller components, better integration of circuitries and better processor architectures.

The limits of the performance of sequential processors are soon to be reached. We can see this simply by looking at four laws of physics and computer science based on our current knowledge:

- Signals cannot travel faster than light.
- Wires carrying signals cannot be narrower than atoms.
- Components cannot be smaller than atoms.
- The architecture of processors cannot continuously be improved.

Despite of these laws, there is still one way left to improve the performance of computers: One can use multiple processors together to solve large or time-consuming computational tasks. This, however, introduces several new problems:

- How can we exploit several processors for solving a problem?
- How can the processors communicate with each others?
- How can the processors be synchronized?
- How can we program such machines?
- What kind of computer architecture is needed?

These problems should be solved before parallel computing can become the mainstream of computing technology. In this thesis we try to find solutions for the last problem. We outline a computer architecture in which the communication system, memories and processors are particularly designed for efficient parallel computation without compromising the performance in sequential computation.

We will continue this chapter by taking a look at the history of processor architectures and the models of parallel computation. In Sections 1.3 to 1.7 we describe basic concepts, example processors, example topologies, benchmark programs, and simulators, which are used when evaluating the goodness of the proposed architectures. In Section 1.8 we explain our approach to the main theme of this thesis—Instruction-Level Parallel Shared-Memory Architecture (IPSM). Section 1.9 reveals why a two-unit architecture is needed. Two final sections outline the contributions and the conclusions of this thesis as well as introduce the individual articles, which constitute the main body of this thesis.

1.1 A short history of processor architectures

A *processor* is a device which executes computing tasks according to given instructions. Physically it consists of numerous interconnected digital gates. These gates form logical entities that preserve data (*registers*, *latches*, *buffers*), carry out the calculation (*arithmetic and logical unit*, ALU), take care of the communication to and from memory (*memory unit*, MU), or control the processor (*sequencers*, SEQ). ALU, MU, and SEQ are called *processing elements* or *functional units* (FU), because they process the data provided by the instructions.

Some functional units may be assigned to special uses like *address units* (ADR) and *compare units* (CMP), which are actually ALUs dedicated to address calculations and comparing operands. A *register unit* (REG) is not considered as a functional unit in this thesis, because registers do not alter data, they only temporarily preserve it.

1.1.1 Instruction scheduling

Different trends in scheduling the execution of instructions in a processor reflect distinctively the development of processor architectures:

The first processors were designed so that they executed instructions strictly sequentially [Hennessy90] (see Figure 1.1). These processors are called *non-pipelined processors* or *scalar processors*.

In the 50's *pipelined execution* or *pipelining* was invented to speed up the execution of instructions [Bloch59, Bucholz62, Kogge81, Flynn95]: The execution of instructions is divided into several parts called *pipeline stages*. The stages are connected to the next to form a pipe. Several instructions can be overlapped in a pipeline by executing different stages of consecutive instructions simultaneously (see Figure 1.1). We call processors that execute instructions in this manner *pipelined processors*. A pipelined processor is called *superpipelined* if the actual execution part of an instruction is divided into multiple parts, so that the actual execution is no longer sequential (see Figure 1.1). In the early 70's *vector processors* were invented. They use superpipelining and multi-element vector registers to achieve high performance in certain scientific applications [Hintz72, Watson72, Russell78].

Theoretically a pipelined processor is up to the number of pipeline stages times faster than a non-pipelined processor with the same number of functional units. The speedup comes from the parallel execution of multiple instructions in the pipeline of the processor. In practice balancing the length of stages, latches between stages, and dependencies between instructions decrease the performance of a pipelined processor remarkably from the theoretical maximum [Hennessy90, Forsell94c].

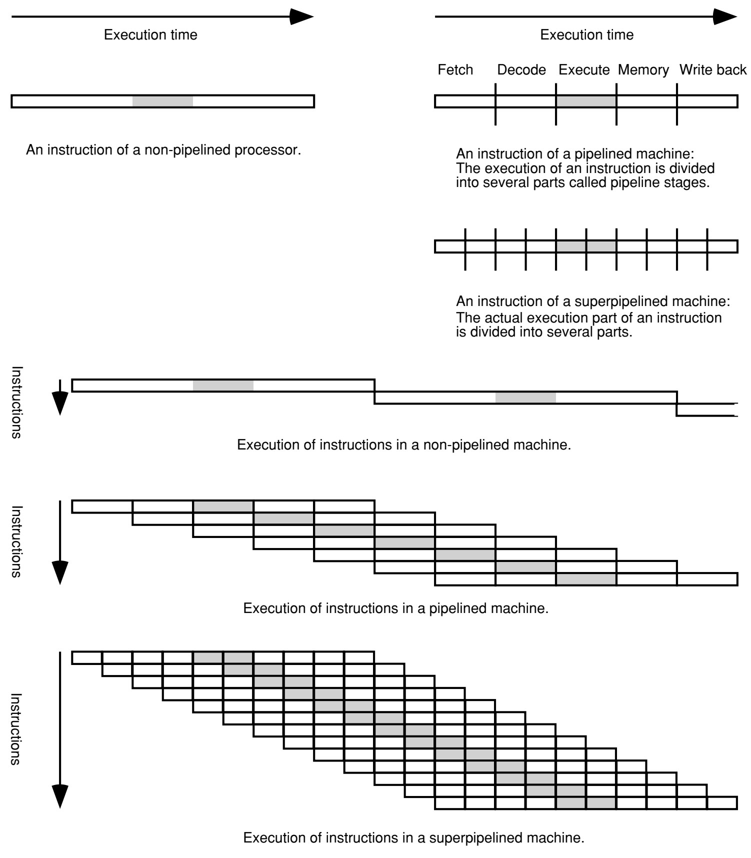

Execution time
An instruction of a non-pipelined processor.
Execution time
Fetch
Decode
Execute
Memory
Write back
An instruction of a pipelined machine: The execution of an instruction is divided into several parts called pipeline stages.
An instruction of a superpipelined machine: The actual execution part of an instruction is divided into several parts.
Instructions
Execution of instructions in a non-pipelined machine.
Instructions
Execution of instructions in a pipelined machine.
Instructions
Execution of instructions in a superpipelined machine.

Figure 1.1 Non-pipelined, pipelined and superpipelined execution. Grey rastering is used to emphasize the actual execution part of an instruction.

In the mid 60's another way of exploiting parallelism on the instruction-level called *superscalar execution* [Thornton64, Anderson67, Tomasulo67] was invented: Multiple instructions are executed in multiple functional units simultaneously. It is left for a programmer to write programs that make a sensible use of the functional units.

There are two kinds of processors using superscalar execution—superscalar processors invented in the 60's and VLIW processors invented in the 80's.

A *superscalar processor* is a processor in which superscalar execution is dynamic. The processor decides which instructions are executed in parallel during the execution of a program (see Figure 1.2) [Thornton64, Anderson67, Johnson89, Hennessy90]. Usually the execution hardware and functional units are also pipelined.

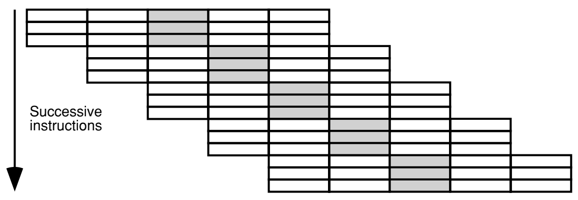


Figure 1.2 Execution in a superscalar processor with three functional units, which are pipelined like in Figure 1.1.

Due to dynamic scheduling of instructions in superscalar processors instructions may not be executed in the original order determined by the program. This is called *out of order execution* [Johnson89].

A *very long instruction word* (VLIW) *processor* is a processor which executes single instructions consisting of the fixed number of smaller subinstructions [Fisher83, Nicolau84] (see Figure 1.3). Subinstructions are executed in multiple functional units. Subinstructions filling actual instructions are determined under compile time.

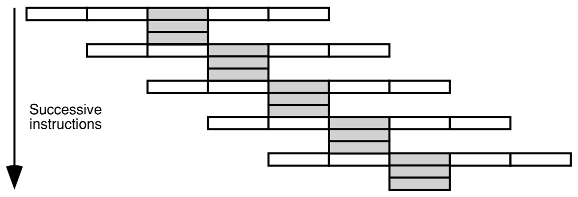


Figure 1.3 Execution in a VLIW processor with three functional units.

In order to achieve high speedups, the programs for VLIW processors must be compiled using advanced compilation techniques breaking the basic block structure of the program [Fisher81, Chang91].

Under ideal conditions a processor using superscalar execution and containing f functional units is f times faster than a non-pipelined processor containing single functional unit. The theoretical speedup is, however, difficult to achieve, because dependencies between the instructions decrease the performance of processors[Hennessy90, Forsell94c].

In the 80's yet another instruction scheduling technique called *multithreading* [Kowalik85, Moore96] was invented: a processor designed so that it is able to execute multiple processes (threads) of a single program in overlapped manner.

The advantages of multithreading include more flexible multitasking, better toleration for memory latencies, and better possibilities for extensive superpipelining. Processors using multithreading are called *multithreaded processors*.

1.1.2 The complexity of instructions

Another historical taxonomy of processor architectures used in this thesis is the classification according to the complexity of instructions.

In the 60's complex machine language instructions were invented to save silicon area and to make programming easier [Wilkes53, Tucker67, Hennessy90]. Complex instructions were implemented by a short programs that were stored in a *microcode memory* inside the processor. Processors using such complex instructions are called *complex instruction-set computer* (CISC) *processors*.

In the late 70's processors using a smaller number of simpler machine-language instructions were reinvented, because CISC processors were shown inefficient [Hennessy90]. The optimization of programs was left to the job of a compiler. Processors using a smaller number of simpler instructions are called *reduced instruction-set computer* (RISC) *processors*.

1.2 Models of parallel computation

In this section we describe a taxonomy of parallel computation, which is based on division into three categories—chip-level parallelism, machine-level parallelism, and network-level parallelism (see Table 1.1).

Chip-level parallelism (instruction-level parallelism)	
Pipelined execution	
Commercial microprocessors	
• *R3000*	http://www.sgi.com
• *first SPARCs*	http://www.sun.com
Vector processors	
• *T90*	http://www.cray.com
• *VPP700*	http://www.fujitsu.com
• *SX-4*	http://www.nec.com
Superscalar execution	
Superscalar processors	
• *PowerPC*	http://www.ibm.com
• *Pentium Pro*	http://www.intel.com
VLIW processors	
• *Multiflow TRACE*	[Almasi94]
Machine-level parallelism (thread-level parallelism)	
Distributed memory machine	
Networks of processors	
• *CM-5*	[Almasi94]
• *SP2*	http://www.ibm.com
• *T3E*	http://www.cray.com
Shared memory machine	
PRAM model	
• *TERA*	http://www.tera.com
• *SB-PRAM*	http://www-wjp.cs.uni-sb.de/sbpram
Network-level parallelism	
Networks of computers	
• *Local area networks of computers*	
• *Internet*	

Table 1.1 The models of parallel computation with example machines.

Consider a program consisting of an ordered set of partially independent instructions solving a computational problem. A scalar processor executes the program by executing instructions one by one in sequential order determined by the program.

A *chip-level parallel* or *instruction-level parallel* processor containing two or more functional units executes the same program so that two or more independent instructions are executed simultaneously in at least one place of the program. (There are certain machines, that cannot be classified by this taxonomy like Illiac IV [Barnes68] and CM-2 [Hillis85]. They use the superscalar execution model, although they are not chip-level designs. We call them *machine-level superscalar computers*.)

Consider a program consisting of a number of subprograms (threads) designed to be executed simultaneously and designed to communicate with each others to solve a common computational task. A *machine level parallel* or *thread-level parallel* computer containing a number of interconnected processors can execute such a program so that subprograms are executed in parallel by the different processors of the computer.

Physically distributed memory machine (DMM) is a model of most current machine level parallel computers like CM-5 [Almasi94], SP2 (see http://www.ibm.com) and T3E (see http://www.cray.com). In a distributed memory machine a set of processor-memory pairs is connected to each others by a network. Communication happens through messages that are sent to other processors using the network.

Another submodel of machine level parallelism is the *ideal shared memory machine* (SMM) [Fortune78, Leighton91, McColl92], in which multiple processors connected to a shared memory (or multiport memory) execute a common program synchronously. Communication and synchronization happens through the shared memory. The *parallel random access machine* (PRAM) model [Fortune78] is a popular shared memory machine model in the literature of parallel algorithms. Unfortunately no remarkable machines using ideal shared memory are currently available [Forsell94a].

In *network-level parallelism* or *distributed processing* the execution of a computational task is divided to computers connected with a network. Due to relatively high latencies of networks, programs for network-level parallel execution should not be communication intensive, if a modest speedup is to be achieved.

Roughly saying, chip-level parallelism deals with things that can be done in the chip-level, whereas machine-level parallelism deals with things that can be done in the computer-level and network-level parallelism deals with things that can be done in the computer-network level.

1.3 Basic concepts

In this section we present some basic concepts that are used in the next four chapters.

Let A be a processor and P a program. The *execution time* $t(A,P)$ of P on A is the product of the number of clock cycles $n(A,P)$ needed by P on A and the clock cycle time $c(A)$ of A. That is

$$t(A,P) = n(A,P)^*c(A).$$

Assuming that for two processors A and B, $t(A,P)<t(B,P)$ holds, we say that the *speedup* s(A,B,P) is $t(B,P)/t(A,P)$, if P is executed in processor A instead of in processor B.

Let $f(A)$ be the number of functional units in A and let $o(A,P)$ be the number of useful functional operations needed by P on A. The *utilization of functional units* $u(A,P)$ of P on A is

$$u(A,P) = \frac{o(A,P)}{f(A)^*t(A,P)}.$$

Let $P_1,P_2,\ldots,P_b$ be b programs and let $s(A,B,P_1),s(A,B,P_2),\ldots,s(A,B,P_b)$ the speedups of the programs achieved when switching from B to A. The *average speedup* $a(A;B;P_1,P_2,\ldots,P_b)$ of programs $P_1,P_2,\ldots,P_b$ is a geometric mean of speedups for a single program:

$$a(A;B;P_1,P_2,\ldots,P_b)= \sqrt[b]{\prod_{i=1}^{b} s(A,B,P_i)}$$

Let $O_1,O_2,\ldots,O_n$ be n operations, e.g., memory read instructions sending messages in a network and waiting for the answers before continuing. If operations are initiated at time $I_1,I_2,\ldots,I_n$ and completed at time $C_1,C_2,\ldots,C_n$, assuming $I_{i+1}<C_i$ for all $1\le i\le n-1$, we say that the *latency* $l(O_i)$ of operation O_i is C_i-I_i for all $1\le i\le n$.

Let $U_1,U_2,\ldots,U_u$ be u processors each simulating the execution of v virtual processors so that the execution of the program of a simulated virtual processor is switched to the execution of the program of another virtual processor at every clock cycle. Let the clock cycle for $U_1,U_2,\ldots,U_u$ be $c(U)$, and let the latency of a memory reference for $U_1,U_2,\ldots,U_u$ be $l(U)$. Assuming $c(U)\ll l(U)<c(U)^*v$ holds, the latency of memory references can be hided, if the program be parallelized for u^*v processors. We call this *parallel slackness technique* or *processor*

overloading, and say that in this case the program has sufficient amount of *parallel slackness* to hide the latency of memory references.

1.4 Example processors

We will investigate eight example processors to figure out possible speedups of the proposed architectural solutions. The processors are DLX, superDLX, M5, M11, T5, T7, T11, and T19 (see Table 1.2).

	DLX	**sDLX**	**M5**	**M11**	**T5**	**T7**	**T11**	**T19**
Execution model	pl	ss	ss	ss	ss+pl	ss+pl	ss+pl	ss+pl
Instr scheduling	pl	ss	VLIW	VLIW	mt	mt	mt	mt
Pipeline stages	5	5	2	2	n	n	n	n
Functional units	4	10	4	10	4	6	10	18
ALU	1	4	1	4	1	3	6	12
AIC	1	1	1	1	1	1	1	1
MU	1	4	1	4	1	1	2	4
SEQ	1	1	1	1	1	1	1	1
Register unit	1	1	1	1	1	1	1	1

Table 1.2 Eight example processors referred in this thesis. (pl=pipelined, ss=superscalar, mt=multithreaded, n=number of stages, which is memory system dependent, presumably $64 \leq n \leq 512$.)

DLX [Hennessy90] is an experimental load/store RISC architecture closely resembling MIPS 3000 architecture (see http://www.sgi.com). DLX will be used as an example of the pipelined execution model. A short description of DLX processor can be found in Appendix A.

SuperDLX [Moura93] is an experimental superscalar processor derived from DLX architecture. It will be used to illustrate the function of superscalar execution model.

SuperDLX is described in Appendix B. For comparison purposes we did not evaluate the original superDLX, but a variation that does not have separate address units. Address calculation of this variation is assumed to be carried out in ALUs like in M5 and M11. We assumed also that this variation is able to detect and eliminate empty operations. Otherwise the performance of superDLX would be significantly lower than that showed in Chapter 4.

M5 and M11 are instances of our own abstract VLIW architecture (*Minimal Pipeline Architecture*, MPA) to be outlined in Chapter 4 (see also Appendix C). They are used to evaluate the scalar unit performance of IPSM.

T5, T7, T11 and T19 are instances of our abstract multithreaded architecture (*Multithreaded Architecture with Chaining*, MTAC) to be outlined in Chapter 5 (see also Appendix D). They are used to evaluate the parallel unit performance of IPSM.

The example processors are chosen in such a way that DLX, M5 and T5 have approximately the same amount of processing resources, i.e., functional units. Similarly, superDLX and M11 have the same amount of processing resources. In addition, T11 has twice the processing resources of T7 and T19 has twice the processing resources of T11.

Unless otherwise stated, we only consider the performance of processors in clock cycles, although with current technology the clock frequency of MTAC can be made much higher (perhaps 600 MHz or more) than the clock frequency of DLX, superDLX and MPA (perhaps 300 MHz). This is because MTAC supports higher degree of pipelining and does not need forwarding paths. The MHz numbers are derived from current PowerPC (see http://www.ibm.com), Alpha (see http://www.dec.com) and Exponential (see http://www.exp.com) processors.

1.5 Example topology

We evaluated a novel communication topology (CBM), which is specially designed for efficient simulation of a shared memory machine.

The *Coated Mesh* (CM) is a topology in which a mesh of routers is coated with processor-memory modules [Leppänen95]. This raises the communication capacity and volume of the network to a level where parallel slackness can be used to hide the latency of the network.

The *Coated Block Mesh* (CBM) is a variation of the coated mesh, in which routers are grouped and replaced with router blocks and processors are grouped with small multiport memories to balance costs of processing and communication hardware. CBM is defined in detail in Chapter 3.

1.6 Benchmarks

We used three benchmark sets to evaluate the performance and some other properties of CBM, MPA and MTAC.

Random communication pattern generation was used to evaluate the communication capacity of CBM: In every clock cycle every processor sent a message to a randomly selected target approximating the effect of randomized hashing of memory locations over the modules. This part of the article [Forsell96a] was created by Ville Leppänen.

The evaluation of MPA was made by simulating the execution of five hand compiled toy benchmarks (see Table 1.3). We measured the execution time, program code size and utilization of functional units in MPA.

Program	**Notes**
block	A program that moves a block of words to another location in the memory
fib	A program that calculates the fibonacci value
sieve	A program that calculates prime numbers using the sieve of Eratosthenes
sort	A program that sorts an array of words using the recursive quicksort algorithm
trans	A program that returns the transpose of a matrix of words
add	A program that calculates the sum of two matrices
block	A program that moves a block of words to another location in the memory
fib	A non-recursive program that calculates the fibonacci value
max	A program that finds the maximum value of matrix of words
pre	A program that calculates all prefix sums of a matrix of words
sort	A program that sorts an array of words using recursive mergesort algorithm
trans	A program that returns the transpose of a matrix of words

Table 1.3 The benchmarks programs for MPA (upper five) and MTAC (lower seven).

The evaluation of MTAC was made by simulating the execution of seven hand compiled toy benchmarks (see Table 1.3). These benchmarks represent much used primitives of parallel programs. Toy benchmark programs were chosen for simplicity, because we wanted to use hand compilation to make sure we were measuring the performance of architectures, not compilers.

Our hand compilation techniques include software pipelining and register renaming [Forsell94c]. Loop unrolling was not extensively used, because it changes the algorithms into more uninteresting ones, i.e., the programs tend to became long sequences of single instructions. To ensure the proper evaluation of the instruction-level parallel execution models, no time consuming instructions like multiplication, division and floating point operations were used in the benchmarks, except in the examples presented in Section 1.9.

1.7 Simulators

We used four simulator programs to evaluate the performance of our example architectures:

The execution of the DLX assembler programs was simulated using *dlxsim* (version 1.1) [Hennessy90] (ftp://ftp-acaps.cs.mcgill.ca). It contains debugger-like tools for managing the execution and statistical tools for collecting information on the usage of instructions, pipeline stalls, and branches.

We used the superDLX simulator (version 1.0) to investigate the superscalar version of the DLX processor [Moura93] (ftp://ftp-acaps.cs.mcgill.ca). Both the superDLX processor and the simulator are being developed at McGill University, Canada. The simulator is a configurable superscalar processor simulator including debugger-like tools for managing execution, visualization tools for investigating the state of the processor and statistical tools for collecting information on the execution of instructions, the usage of functional units and buffers, the branch prediction and the instruction issue.

To investigate the effectiveness of M5 and M11 processors we implemented a simulator called *MPAsim* (version 1.0.1) [Forsell94b] (ftp://cs.joensuu.fi). MPAsim includes debugger-like tools for managing execution, visualization tools for inspecting the state of processor and memory, and statistical tools for collecting information on the instruction execution, the utilization of functional units and the memory usage. The program includes also a code translation utility that converts DLX assembler programs into an MPA assembler and a parallelizer that compresses the basic blocks of MPA code.

We also implemented a simulator called *MTACsim* (version 1.1.0) [Forsell97b] (ftp://cs.joensuu.fi) to investigate the effectiveness of T5, T7, T11 and T19 processors. MTACsim includes debugger-like tools for managing execution, visualization tools for inspecting the state of processor and memory, and statistical tools for collecting information on the instruction execution, the utilization of functional units and the memory usage.

All benchmarks testing processors were run assuming ideal memory hierarchy (i.e., there were no cache misses or the cycle time of main memory was assumed smaller than that of the processor).

1.8 Approach taken

We selected to investigate the realization of the SMM model, because it is theoretically elegant and easy to program. In addition, if the model is realizable a large number of algorithms written for it become instantly usable [Fortune78, Leighton91, McColl92].

First, we investigated the feasibility of an obvious VLSI implementation of a SMM. Therefore we outlined two possible structures for limited-size multiport memory chips and estimated the cost-effectiveness of a memory system using proposed chips. Unfortunately it turned out that large machines using such multiport memories are not feasible.

After that we began to investigate the possibility of simulating a SMM by a DMM work optimally. This, however, introduces two problems:

- The latency of memory operations is high due to message passing
- Messages can be contended in a network

According to earlier investigations these problems can be solved by using two basic techniques—parallel slackness and randomized hashing [Schwartz80, Gottlieb83, Pfister85, Ranade87,Valiant90, Abolhassan93]. The proposed architectures, however, leave many practical problems open:

1. What kind of a network topology is realizable and fast enough?
2. What kind of processors are fast and still capable of exploiting parallel slackness?
3. How can we eliminate thread switch and message handling overheads?
4. How can we exploit instruction-level parallelism in a processor aimed for thread-level parallel execution?
5. How can we eliminate false dependencies between instructions?
6. How can we execute sequential programs efficiently in a parallel machine?
7. How can we synchronize execution between processors and threads?

We designed a CBM communication architecture to address problem 1. CBM is physically scalable, because it can be fitted into a three dimensional space, and the length of the wires is short and does not depend on the number of processors. CBM combines limited size multiport memories and a coated mesh topology providing a balancing method to more cost-effective systems.

In addition, we outlined MPA to address problems 5 and 6. MPA uses minimal pipelining to eliminate delays caused by the false dependencies between instructions. This leads to a high performance when executing sequential programs. Two implementations of MPA were evaluated with simulations.

Finally, we outlined MTAC to address problems 2, 3, 4, 5 and 7. MTAC uses multithreading to hide the latency of passing memory request messages in a network, chaining to exploit inter-thread instruction-level parallelism eliminating false dependencies between the instructions, and superpipelining to maximize performance. There are no thread switch and message handling overheads. Four implementations of MTAC were evaluated with simulations.

This thesis combines the mentioned investigations into a single architectural framework—*Instruction Level Parallel Shared Memory Architecture* (IPSM)—consisting of four main components: limited-size multiport memories, scalable two-level communication network, minimally pipelined scalar unit architecture and superpipelined parallel unit architecture. We selected to use a separate scalar unit because MTAC turned out to be slow with sequential programs (see Section 1.9 for further information). According to our experiments MPA is a natural choice for our scalar unit architecture, because it provides better performance than basic pipelined and superscalar processors with sequential programs.

The evaluations of the proposed components were made by running benchmark programs on processors and by sending random message patterns on communication networks using parametric simulators. The feasibility analysis was made by estimating the VLSI area needed for implementations. We limited the discussion only to VLSI techniques, because the state of optical computing is still rather undeveloped. The situation may, however, change in the near future.

1.9 Why a scalar unit is needed?

IPSM has a two unit architecture resembling the two unit architecture used in vector computers. All applications that are parallel enough are executed in the parallel unit using MTAC, and the remaining applications are executed in the scalar unit using MPA.

The advantages of the two unit architecture can be seen by looking at two applications—Blk and Rand. Blk is a program that moves a 6 MB memory block to another place in the memory. It is completely parallelizable, because the words of the memory block are independently movable. Rand is a program that calculates a 1024th random number using congruence generator method. It is strictly sequential, because the value of a generator is unpredictably dependent on the previous value of the generator.

Assume that we have a 32-bit IPSM computer with a parallel unit consisting of 1536 512-thread T5 processors running at 600 MHz connected to each others with a 3-dimensional 16x16x16-router mesh, and a scalar unit consisting of a single M5 processor running at 300 Mhz. Assume also that the synchronization between consecutive steps of the parallel unit can be issued in zero time and that the data prefetching is perfect in the case of the scalar unit.

With these assumptions the execution time of Blk is 3.4 us in the parallel unit and 11 ms in the scalar unit. Thus, the achieved speedup is 3100. In the case of Rand the situation is totally different. The execution time of Rand is 3.5 ms in the parallel unit and 14 us in the scalar unit. Now, the scalar unit is 260 times faster than parallel unit.

1.10 Contents of the individual articles

The components of IPSM, except CBM, are original designs presented first time in the research articles [Forsell94a, Forsell96b, Forsell97a] which constitute the main body of thesis together with the joined article [Forsell96a]. The contents of these articles are presented in chronological order.

1.10.1 Are multiport memories physically feasible?

This article [Forsell94a] is the oldest one. At that time we believed that simulations of a shared memory machine by a distributed memory machine were too slow for real applications: Mesh-based structures did not have enough bandwidth to profit from parallel slackness [Prechelt93, Leppänen93] and efficient logarithmic diameter networks, like butterflies, were impossible to build [Ranade87]. The situation has changed after the introduction of remarkably faster communication components and better topologies like the coated meshes [Forsell96a].

We investigate the possibility of building multiport memory chips, which can be used as a building block of an ideal shared memory machine. Two possible structures for such chips are proposed. Unfortunately, the complexity of the proposed chips turns out to be very high. According to an evaluation, p-port memory is p^2 times more complex than a single port memory of the same size. It is hard to imagine that multiport memories could be implemented by another structure that is simpler than that proposed.

1.10.2 Efficient two-level mesh based simulation of PRAMs

This article [Forsell96a] is joined work with Ville Leppänen and Martti Penttonen. My contribution was mainly to propose limited size multiport memories for low-level communication, propose hardware techniques for retaining memory consistency and estimate the overall feasibility of the proposed communication and memory hardware.

We extend the concept of coated mesh to coated block mesh and introduce efficient routing techniques like synchronization wave and moving threads. This simplifies the structure of router blocks significantly, but makes processor-memory blocks much more complex. At the same time the latency of memory requests decreases implying lower overloading factors. According to our measurements the proposed routing techniques decrease the work of a parallel machine using coated mesh or coated block mesh in relation to the work done by an ideal shared memory machine below two. In addition, varying the size of the block according to available resources provides a good balancing method for more cost effective systems.

1.10.3 Minimal pipeline architecture—an alternative to superscalar architecture

Superscalar processor architects use very deep pipelines to dynamically eliminate delays caused by false data dependencies between instructions. This increases the delays caused by control dependencies. Architects have to use extensive branch prediction techniques to reduce the loss of performance due to control delays. Unfortunately this (together with dynamic instruction scheduling mechanism) leads to a complex internal structure of processors.

An alternative approach is to minimize the length of a pipeline so that both data and control delays are minimized. This requires novel techniques like general forwarding, uncoded instructions, simple instruction set and fast branching (see Chapter 4 for explanation). This article [Forsell96b] focuses on describing these techniques and outlining the structure of minimal pipeline architecture.

As the measurements indicate this approach gives better scalar unit performance than superscalar architectures using out of order execution while it requires no more silicon area.

1.10.4 MTAC—a multithreaded VLIW architecture for PRAM simulation

This article [Forsell97a] is the newest one. Most current parallel machines have been claimed inefficient. This is mainly because current processor are not designed with a shared memory simulation in mind. They lack proper mechanisms for handling parallel slackness, and they suffer from unnecessary communication overheads.

We outline a multithreaded architecture with chaining, which is specially designed for an efficient PRAM simulation. The list of techniques used is extensive: superpipelining, multithreading, functional unit chaining and

VLIW scheduling. The architecture uses multithreading to hide the latency of memory requests in a communication network. VLIW scheduling is needed to simplify the structure of the processor and to allow for instruction-wise synchronization. Functional unit chaining gives a possibility to run a block of code containing data dependencies within a single clock cycle. Finally, extensive superpipelining decreases the clock cycle to a minimum.

The described architecture yields high parallel unit performance for the shared memory simulation, but fails to speedup the sequential parts of programs (see Section 1.9). Thus, MTAC is not the ultimate solution for a general purpose parallel computer, but it need designs like MPA for an efficient scalar-unit execution.

1.11 Conclusions

The main contribution of this thesis is the IPSM. The following statements represent the conclusions of the research on this architecture:

- Our IPSM framework outlines a general purpose parallel computer architecture based on the shared memory model. It succeeds to avoid most architectural bottlenecks, which are present in most current parallel computes, because they use architectures originally designed for sequential computing. IPSM features a static VLIW-style scheduling of instructions, and the absence of message passing and thread switch overheads. In the execution of parallel programs, IPSM exploits parallel slackness to hide the high memory system latency, and interthread instruction-level parallelism to eliminate delays caused by the dependencies between instructions belonging to a single thread. In the execution of sequential programs, IPSM uses minimal pipelining to minimize the delays caused by the dependencies between instructions.

- Limited-size multiport memories are shown physically feasible, but unfortunately a proposed p-port memory system is p^2 times more expensive than a single-port memory system of the same size. However, the proposed multiport memories can be used as a building blocks for two-level communication systems or small shared memory machines. The structure of such a limited-size multiport memory chip is a simple extension of the structure of a single port memory chip.

- As a generalization of the coated mesh, the 2-dimensional and 3-dimensional coated block meshes simulate the PRAM model time-processor optimally with moderate simulation cost. Using proper amount of parallel slackness, the cost can be decreased clearly below 2 routing steps per a simulated PRAM step.

- A VLIW architecture using crossbar interconnection of functional units and minimal pipelining (MPA) is not more complex than a basic superscalar architecture using out of order execution and branch prediction, but offers better performance.

- According to our experiments chaining seems to improve the exploitation of instruction-level parallelism in MTAC to a level where the achieved speedup corresponds to the number of functional units in a processor.

- Minimal pipelining can be used to eliminate all false data and control dependencies resulting high scalar-unit performance whereas extensive superpipelining along with multithreading and functional unit chaining can be used to eliminate even true dependencies between the instructions belonging to a single thread resulting a good parallel unit performance.

References

[Abolhassan93]
F. Abolhassan, R. Drefenstedt, J. Keller, W. Paul, D. Scheerer, On the Physical Design of PRAMs, *Computer Journal* **36**, 8 (1993), 756-762.

[Almasi94]
G. Almasi and A. Gottlieb, *Highly Parallel Computing*, Benjamin/Cummings, Redwood City, 1994.

[Alverson 90]
R. Alverson, D. Callahan, D. Cummings, B. Kolblenz, A. Porterfield, B. Smith, The Tera Computer System, *Proceedings of the International Conference on Supercomputing*, Association for Computing Machinery, New York, 1990, 1-6.

[Anderson67]
D. Anderson, F. Sparacio and R. Tomasulo, The IBM 360 Model 91: Machine philosophy and instruction handling, *IBM Journal of Research and Development* **11**, 1 (1967), 8-24.

[Barnes68]
G. Barnes, R. Brown, M. Kato, D. Kuck, D. Slotnick and R. Stokes, The Illiac IV computer, *IEEE Transactions on Computing* **C-17**, (1968), 746-757.

[Bloch59]
E. Bloch, The engineering design of the Stretch computer, *Proceedings of the Fall Joint Computer Conference*, 1959, 48-59.

[Bucholz62]
W. Bucholz, *Planning a Computer System: Project Stretch*, McGraw-Hill, New York, 1962.

[Chang91]
P. Chang, S. Mahlke, W. Chen, N. Warter and W. Hwu, IMPACT: An architectural framework for multiple-instruction-issue processors, *Proceedings of the 18th Annual International Symposium on Computer Architecture*, Association for Computing Machinery, New York, 1991, 266-275.

[Fisher81]
J. Fisher, Trace Scheduling: A technique for global microcode compaction, *IEEE Transactions on Computers* **C-30**, (1981), 478-490.

[Fisher83]
J. Fisher, Very Long Instruction Word Architectures and ELI-512, *Proceedings of the 10th Annual International Symposium on Computer Architecture*, Computer Society Press of the IEEE, Washington, 140-150.

[Flynn95]
M. Flynn, *Computer Architecture—Pipelined and Parallel Processor Design*, Jones and Bartlett Publishers, Boston, 1995.

[Forsell94a]
M. Forsell, Are multiport memories physically feasible?, *Computer Architecture News* **22**, 4 (1994), 47-54.

[Forsell94b]
M. Forsell, MPASim - A simulator for MPA, *Report B-1994-3*, Department of Computer Science, University of Joensuu, Joensuu, 1994.

[Forsell94c]
M. Forsell, Design and Analysis of Some Chip Level Parallel Architectures, *Licentiate thesis*, Department of Computer Science, University of Joensuu, Joensuu, 1994.

[Forsell96a]
M. Forsell, V. Leppänen and M. Penttonen, Efficient Two-Level Mesh based Simulation of PRAMs, *Proceedings of the International Symposium on Parallel Architectures, Algorithms and Networks*, June 12-14, 1996, Beijing, China, 29-35.

[Forsell96b]
M. Forsell, Minimal Pipeline Architecture—an Alternative to Superscalar Architecture, *Microprocessors and Microsystems* **20**, 5 (1996), 277-284.

[Forsell97a]
M. Forsell, MTAC - a multithreaded VLIW architecture for PRAM simulation, to be published in *Journal of Universal Computer Science* (http://cs.joensuu.fi:8080/jucs_root), 1997.

[Forsell97b]
M. Forsell, MTACSim - A simulator for MTAC, *Report B-1997-1*, Department of Computer Science, University of Joensuu, Joensuu, 1997.

[Fortune78]
S. Fortune and J. Wyllie, Parallelism in Random Access Machines, *Proceedings of 10th ACM STOC*, Association for Computing Machinery, New York, 1978, 114-118.

[Hennessy90]
J. Hennessy and D. Patterson, *Computer Architecture: A Quantitative Approach*, Morgan Kaufmann Publishers Inc., Palo Alto, 1990.

[Hillis85]
W. D. Hillis, *The Connection Machine*, MIT Press, Cambridge, 1985.

[Hintz72]
R. Hintz and D. Tate, Control data STAR-100 processor design, *COMPCON*, February 1972, 1-4.

[Gottlieb83]
A. Gottlieb, R. Grishman, C. P. Kruskal, K. P. McAuliffe, L. Rudolph and M. Snir, The NYU Ultracomputer - Designing a MIMD, shared-memory parallel machine, *IEEE Transactions on Computers* **C-32**, (1983) 175-189.

[Johnson89]
W. M. Johnson, Super-Scalar Processor Design, *Technical Report No. CSL-TR-89-383*, Stanford University, Stanford, 1989.

[Kogge81]
P. M. Kogge, *The Architecture of Pipelined Computers*, Hemisphere Publishing Corporation, Washington, 1981.

[Kowalik85]
J. Kowalik (editor), *Parallel MIMD computation: the HEP supercomputer and its applications*, MIT Press, Cambridge, 1985.

[Leighton91]
T. F. Leighton, *Introduction to Parallel Algorithms and Architectures: Arrays, Trees, Hypercubes*, Morgan Kaufmann, San Mateo, 1992.

[Leppänen93]
V. Leppänen and M. Penttonen, Simulation of PRAM Models on Meshes, *Research Report R-93-4*, Department of Computer Science, University of Turku, Turku, 1993.

[Leppänen95]
V. Leppänen and M. Penttonen, Work-Optimal Simulation of PRAM Models on Meshes, *Nordic Journal of Computing* **2**, 1 (1995), 51-69.

[McColl92]
W. F. McColl, General Purpose Parallel Computing, *Lectures on Parallel Computation Proceedings 1991 ALCOM Spring School on Parallel Computation (Editors: A. M. Gibbons and P. Spirakis)*, Cambridge University Press, Cambridge, 1992, 333-387.

[Moore96]
S. Moore, *Multithreaded Processor Design*, Kluwer Academic Publishers, Boston, 1996.

[Moura93]
C. Moura, SuperDLX - A Generic Superscalar Simulator, *ACAPS Technical Memo 64*, McGill University, Montreal, 1993.

[Nicolau84]
A. Nicolau and J. Fisher, Measuring the parallelism available for very long instruction word architectures, *IEEE Transactions on Computers* **C-33**, 11 (1984), 968-976.

[Pfister85]
G. F. Pfister, W. C. Brantley, D. A. George, S. L. Harvey, W. J. Kleinfelder, K. P. McAuliffe, E.A. Melton, V. A. Norton and J. Weiss, The IBM Research Parallel Processor Prototype (RP3): Introduction and Architecture, *Proceedings of International Conference on Parallel Processing*, 1985, 764-771.

[Prechelt93]
L. Prechelt, Measurements of MasPar MP-1216A Communication Operations, *Technical Report 01/93*, Universität Karlsruhe, Karlsruhe, 1993.

[Ranade87]
A. G. Ranade, S. N. Bhatt, S. L. Johnson, The Fluent Abstract Machine, *Technical Report Series BA87-3*, Thinking Machines Corporation, Bedford, 1987.

[Russell78]
R. Russell, The Cray-1 computer system, *Communications of the ACM* **21**, 1 (1978), 63-72.

[Schwarz80]
J. T. Schwarz, Ultracomputers, *ACM Transactions on Programming Languages and Systems* **2**, 4 (1980) 484-521.

[Thornton64]
J. Thornton, Parallel operation in the Control Data 6600, *Proceedings of the Fall Joint Computer Conference* 26, 1964, 33-40.

[Tomasulo67]
R. Tomasulo, An efficient algorithm for exploiting multiple arithmetic units, *IBM Journal of Research and Development* **11**, 1 (1967), 25-33.

[Tucker67]
S. Tucker, Microprogram control for the System/360, *IBM Systems Journal* **6**, 4 (1967), 222-241.

[Valiant90]
L. G. Valiant, A Bridging Model for Parallel Computation, *Communications of the ACM* **33**, 8 (1990), 103-111.

[Watson72]
W. Watson, The TI ASC—A highly modular and flexible super computer architecture, *Proceedings of the 1972 AFIPS Fall Joint Computer Conference*, 221-228.

[Wilkes53]
M. Wilkes, The best way to design an automatic calculating machine, *in Manchester University Computer Ibaugural Conference* 1951, Ferranti, Ltd., London, 1953.

Chapter 2

Are multiport memories physically feasible?

Martti J. Forsell

Department of Computer Science, University of Joensuu, PB 111, FIN-80101 Joensuu, Finland

Abstract

A Parallel Random Access Machine (PRAM) is a popular model of parallel computation that promises easy programmability and great parallel performance, but only if efficient shared main memories can be built. This would not be easy, because the complexity of shared memories leads to difficult technical problems. In this paper we consider the idea of true multiport memory that can be used as a building block for efficient PRAM-style shared main memory. Two possible structures for multiport memory chips are presented. We shall also give a preliminary cost-effectivity and performance analysis for memory systems using proposed multiport RAMs. Results are encouraging: At least small size multiport memories seem to be physically feasible.

Keywords: PRAM, parallelism, shared memories, VLSI

2.1 Introduction

Efficient parallel computers are hard to manufacture because of difficult technical problems. The most important is: how to arrange efficient communication between processors?

Theoretically this problem was best solved by using a shared main memory between processors [Schwarz66, Karp69, Fortune78]. In the 50’s, 60's and 70's however, the memory manufacturing technology was so underdeveloped that the idea of building true shared memories was considered impossible. Instead several other possibilities, like

Reprinted from SIGARCH Computer Architecture News, 22 4 Martti Forsell, Are multiport memories physically feasible?, 47-54, 1994 with kind permission of Association for Computing Machinery, New York, USA.

memory interleaving and memory distribution were investigated [Burnett70, Enslow74, Schwarz80].

The use of true shared main memory continued among algorithm writers, because of the simplicity of programming. Even several abstract parallel programming models using shared memory were designed. The most widely used model is a Parallel Random Access Machine [Fortune78, Leighton91, McColl92] (see Section 2.2 for a brief explanation).

During the 80's one began to use message passing processor networks to simulate the PRAM model [Schwatz80, Gottlieb83, Gajski83, Hillis85, Pfister85, Ranade87, Abolhas91], because the manufacturing of shared memories was still considered impossible. Despite of those efforts, no serious commercial products are available yet.

In this paper we return to the original idea of a shared memory of PRAM model and consider it in the light of current VLSI technology. We claim that the progress in the VLSI area during the last two decades has made it possible to consider manufacturing the main memory of a PRAM-style computer of shared semiconductor memories.

The number of ports and size of multiport memories will be quite limited, but it will be shown that even with small port count computers with multiport memories will provide good performance. Two possible structures of multiport memories will be presented. We will also estimate the complexity, cost-effectiveness and performance of proposed multiport memories.

2.2 Parallel random access machine

The *Parallel Random Access Machine* (PRAM) model is a logical extension of the Random Access Machine (RAM) model. It is a widely used abstract model of parallel computation [Fortune78, Leighton91, McColl92].

A PRAM consists of P processors each having a similar register architecture. All processors are connected to a shared memory (see Figure 2.1). All processors run the same program synchronously, but a processor may branch within the program independently of other processors. That is, each processor has its own Program Counter.

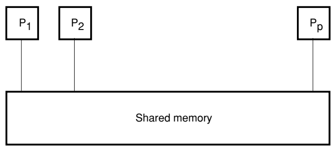


Figure 2.1 The PRAM model. $P_1,\ldots,P_P$ are processors.

The model has several different variants according to level of shared memory access. We refer to two of them in this paper (most of the other models can be treated analogously):

CRCW Common	Processors can read and write a memory cell simultaneously. In the case of writing all processors storing to the same cell must have identical data.
CRCW Priority	Processors can read and write a memory cell simultaneously. In the case of writing to the same cell the processor with the lowest index succeeds.

The Recent PRAM algorithm research is valuable because it promises that parallel processing power of multiprocessors can be exploited in very large range of applications if PRAM-style computers can be built.

PRAM processors are easy to implement, because they are similar to those in sequential machines. On the contrary, the implementation of shared memory is very difficult due to technical problems. In the following section we will consider these problems and proposed solutions.

2.3 Proposed implementations of shared memories

Except multiport memories there are at least three different methods to implement a shared memory: One can use an ordinary memory, memory interleaving or shared memory simulation by distributed memories.

2.3.1 Ordinary memory

Ordinary memories or single port memories are used to implement shared memories in small multiprocessor systems. An ordinary memory module can be connected to several processors through a time-shared bus [Enslow74] (see Figure 2.2).

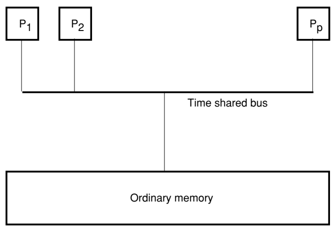


Figure 2.2 A multiprocessor system using an ordinary memory.

By using this method, we however face serious bandwidth problems: If, for example, *P* processors are simultaneously accessing the memory, this will take *P* times longer than in the case of a single processor. That is because only one memory access can be done through a single port at the same time. This problem can be partially solved by using snooping caches [Tomasevic93], but they will not help us in the case of multiple simultaneous writes.

In order to use an ordinary memory in a multiprocessor system the memory must be superfast, otherwise the performance of the system will not be much greater than the performance of a single processor system. Because the current access times of ordinary memory chips are apparently equal to processor clock cycle times, the existence of a superfast memory is considered unrealistic. Therefore, there is almost no possibility for implementing any serious parallel computer system with ordinary memories.

2.3.2 Memory interleaving

Memory interleaving is a memory system implementation method that is used for example in current supercomputers. The memory of an interleaved system is divided into several independent banks that are connected to processors through a crossbar switch [Burnett70, Enslow74] (see Figure 2.3).

Information is stored in the memory with sequential items residing in banks that are consecutive, modulo the number of memory banks. A *memory bank* is a memory block that is built as an ordinary memory.

Thus, it is not possible that several processors simultaneously access a memory within the same bank. However, several processors can access several banks simultaneously.

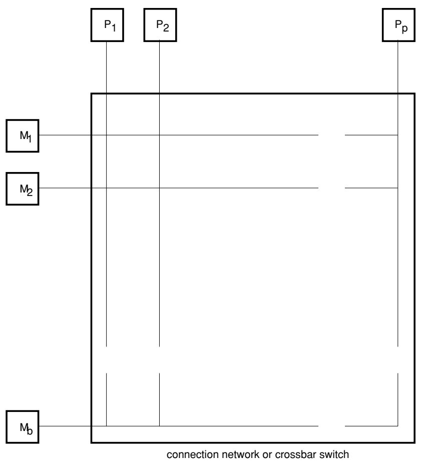


Figure 2.3 A multiprocessor system using an interleaved memory. $P_1,...,P_P$ are processors and $M_1,...,M_b$ are memory banks.

The advantage of the memory interleaving is obvious with vectorizable algorithms, because subsequent vector elements can be placed in subsequent banks and accessed simultaneously. Thus parts of vectors can be fetched in one memory cycle.

However interleaved memory systems fail to provide the ease of programming and the performance of ideal shared memories. Constantly it happens that several processors try to access simultaneously the same bank and some waiting is needed. This is very usual especially in efficient parallel PRAM algorithms, like parallel sorting and prefix calculation.

2.3.3 Simulation of a shared memory by a distributed memory

So far, the most promising method for implementing a shared memory has been a *simulation of a shared memory by a distributed memory*. In this solution an ordinary memory is distributed over a processor network (see Figure 2.4).

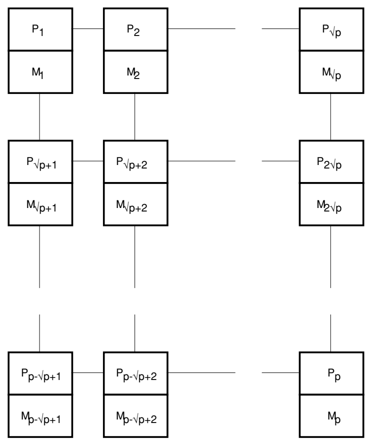


Figure 2.4 A multiprocessor system using a shared memory simulation by a distributed memory. Processors $P_1,\ldots,P_p$ with their local memories $M_1,\ldots,M_p$ are organized here as a 2-dimensional mesh.

Each network node consists of both a processor and an ordinary local memory. Processors use message passing to access memory locations that do not lie in their local memory.

According to investigations such a processor networks can be used to simulate a PRAM or some other parallel processing model [Schwatz80, Gottlieb83, Hillis85, Pfister85, Alt87, Bilardi88, Ranade87, Valiant90, Feldman92, Prechelt93]. The suggested network topologies range from hypercubes to butterflies and meshes.

It has turned out that the proposed shared memory machines have poor performance in relation to the ideal PRAM model [Ranade87, Blelloch89, Prechelt93]. That is because the simulation of a shared memory takes so much time.

Table 2.1 shows estimated average routing times of message permutations in three architectures. As can be seen, hundreds or even thousands of steps may be needed for simulating one permutation. As a result one PRAM step would execute in several hundreds of steps. Thus, we may need hundreds of processors even to get the performance of single processor.

We should, however, be careful making radical conclusions from these numbers. The performance of real parallel machines depend heavily on very many technical and architectural things, which should be mentioned explicitly when doing comparisons. The numbers indicate, however, that the performance of the investigated systems will not be sufficient for efficient PRAM simulation.

Machine	**Steps**	**Processors**
Fluent Machine [Ranade87]	3380	112k
PRAM on CM2 [Blelloch89]	600	64k
MasPar MP-1216A [Prechelt93]	4800	16k

Table 2.1 Latencies for permutation routing in memory cycles on certain parallel machines.

2.4 A natural solution for shared memories: multiport semiconductor RAM

The fourth and the most obvious way to implement shared memories is to use multiport RAMs as building blocks for a true shared memory.

A *Multiport Random Access Memory* (multiport RAM) is a memory having multiple ports that are to be used to access memory cells simultaneously and independently of each other.

In parallel computers one processor is usually connected to one port. From the processor point of view there exists a uniform shared memory connected to it. Other processors do not affect the operation of a processor; only the contents of a memory may be changed by instructions executed by other processors.

Note that the term multiport memory is used here in different meaning than in the 70's, when multiport memories were actually interleaved memories in which connection networks were closely integrated into the memory systems [Enslow74].

Small and fast multiport *Static Random Access Memories* (SRAMs) are currently commercially available, for example, from IDT. Sizes of memories vary from 8 to 128 kbit and number of ports vary from 2 to 4.

It is possible to use these memories to build *Concurrent Read Exclusive Write* (CREW) PRAM-style shared memories, but the limited size and port count badly affect the performance of such memory systems in real world applications. Also the structure of these memories does not allow simultaneous writes that is needed in CRCW PRAM-style memories.

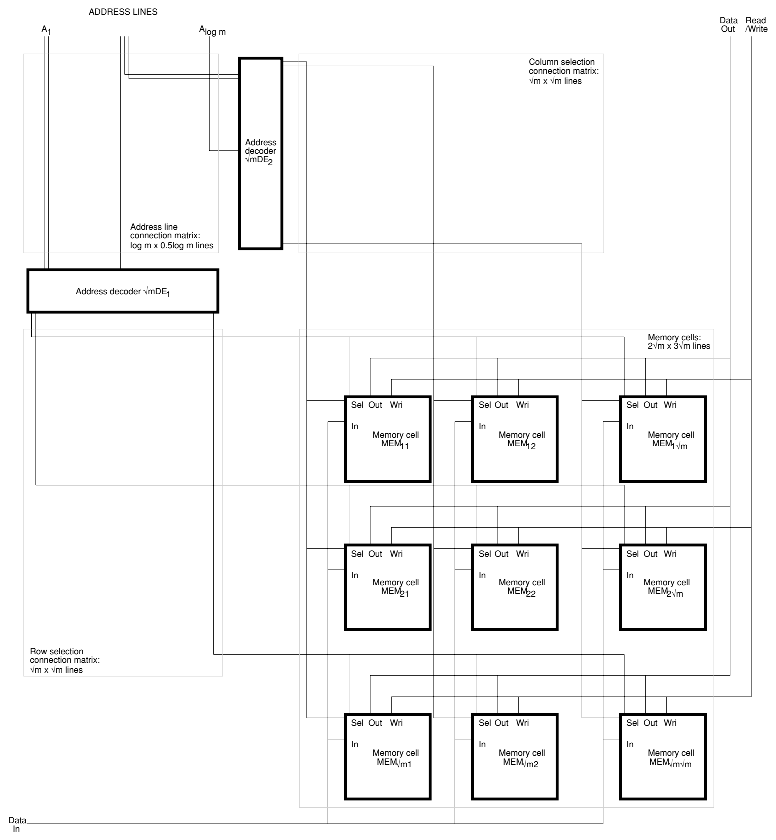

ADDRESS LINES
A1
Alog m
Data Out
Read /Write
Column selection connection matrix: √m x √m lines
Address decoder √mDE2
Address line connection matrix: log m x 0.5log m lines
Address decoder √mDE1
Memory cells: 2√m x 3√m lines
Sel Out Wri In Memory cell MEM11
Sel Out Wri In Memory cell MEM12
Sel Out Wri In Memory cell MEM1√m
Sel Out Wri In Memory cell MEM21
Sel Out Wri In Memory cell MEM22
Sel Out Wri In Memory cell MEM2√m
Row selection connection matrix: √m x √m lines
Sel Out Wri In Memory cell MEM√m1
Sel Out Wri In Memory cell MEM√m2
Sel Out Wri In Memory cell MEM√m√m
Data In

Figure 2.5 The logical structure of an ordinary single port m cell static RAM chip. Chip select and power lines are not shown. The physical lay-out of components does not reflect the actual placement of semiconductor elements and is meant only for comparing single port and multi port versions of memory chips. For the same reason the connection matrices are rounded with dotted lines and labelled with complexity numbers.

To estimate the theoretical feasibility of multiport main memories in general, two more powerful structures of multiport RAMs are presented and analyzed.

2.4.1 The structure of a multiport RAM

Let us first take a look at the logical structure of an ordinary RAM. A typical SRAM chip consists of two address decoders, memory cell array, bus interface and some wiring (see Figure 2.5).

The memory chip is connected to other devices through address lines, data lines, read/write line, chip select line and power lines. For the well known complexity reasons memory cells are arranged into the form of a square array instead of line array in current memory chips. The address must therefore be divided into two parts for decoding.

To select an appropriate memory cell, appropriate row and column select lines must be activated. Address decoders are used to select the right memory cell by activating the right selection lines according to the address information provided in address lines. The read operation (write operation) is selected by activating (deactivating) read/write line.

A memory cell consists of few gates that are connected to each others (see Figure 2.6). The cell is connected to other elements in the chip through row and column select lines, read/write line, input line and output line.

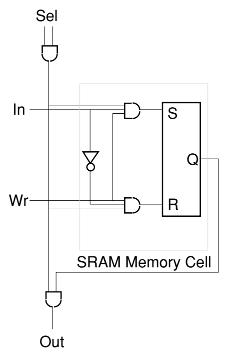


Figure 2.6 An example of a typical SRAM cell.

To transform a single port memory to a multiport memory ports are added by adding new address lines, data lines, chip select lines and decoders for each port. Inside the chip the selection circuit of memory cells is duplicated for each port and the memory cells are made multiported (see Figure 2.7).

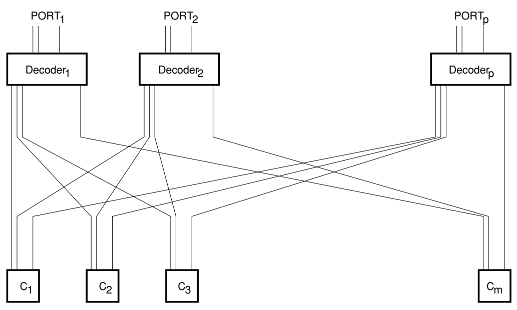


Figure 2.7 An ordinary memory is made parallel by adding new lines for each port and duplicating the cell selection circuitry for each port. $C_1,...,C_m$ are multiported cells.

A memory chip now consists of two address decoders per port, a common memory cell array and multiple wiring circuitries (see Figure 2.8). The multiport chip is connected to outside world through multiple address lines, multiple data lines, multiple read/write lines, multiple chip select lines and power lines.

The memory operations are done in the similar manner to that in the ordinary memory except now multiple ports control multiple operations in one or more cells. Address information provided in a memory port is decoded in appropriate address decoders to select appropriate memory cell by activating right row and column select lines of the cell.

The structure of a multiport memory cell is naturally more complex than the structure of an ordinary cell. We present here the logical structure of a CRCW Common cell and a CRCW Priority cell.

In CRCW Common-style memories several processors can simultaneously write the same data to a memory cell. The correct operation is obtained by ORing the input values together and then feeding the result to the original cell (see Figure 2.9). A read/write operation is determined by ORing read/write lines and cell select is determined by ORing the select lines. Finally, the output circuitry is duplicated for each port.

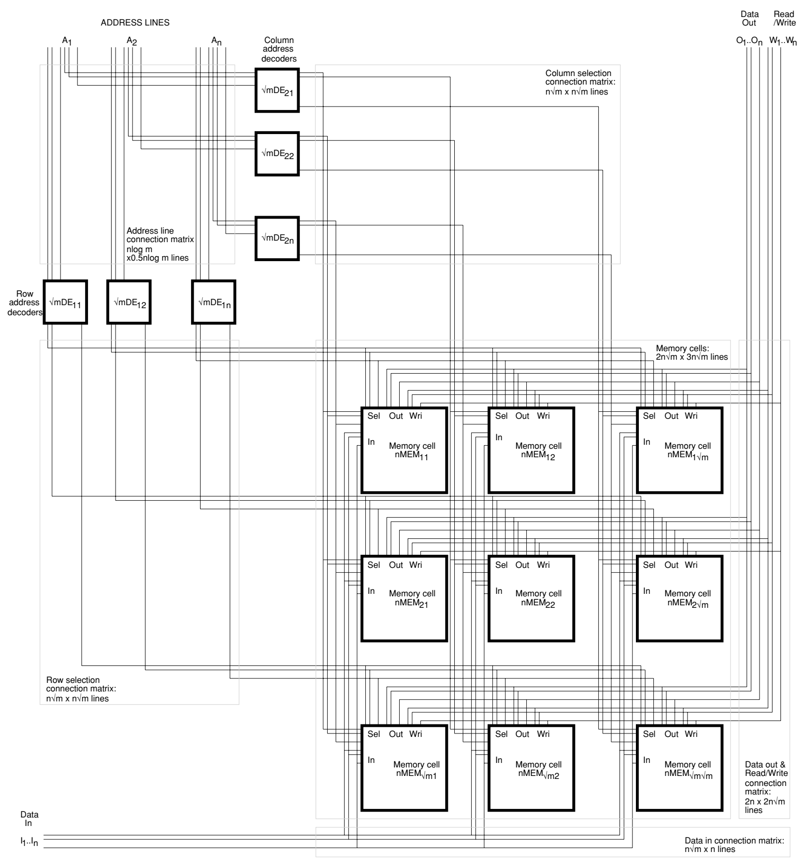

ADDRESS LINES
A_1
A_2
A_n
Column address decoders
Data Out
Read /Write
$O_1..O_n$
$W_1..W_n$
$\sqrt{m}DE_{21}$
$\sqrt{m}DE_{22}$
$\sqrt{m}DE_{2n}$
Column selection connection matrix: n√m x n√m lines
Address line connection matrix nlog m x0.5nlog m lines
Row address decoders
$\sqrt{m}DE_{11}$
$\sqrt{m}DE_{12}$
$\sqrt{m}DE_{1n}$
Memory cells: 2n√m x 3n√m lines
Sel Out Wri In Memory cell $nMEM_{11}$
Sel Out Wri In Memory cell $nMEM_{12}$
Sel Out Wri In Memory cell $nMEM_{1\sqrt{m}}$
Sel Out Wri In Memory cell $nMEM_{21}$
Sel Out Wri In Memory cell $nMEM_{22}$
Sel Out Wri In Memory cell $nMEM_{2\sqrt{m}}$
Row selection connection matrix: n√m x n√m lines
Sel Out Wri In Memory cell $nMEM_{\sqrt{m}1}$
Sel Out Wri In Memory cell $nMEM_{\sqrt{m}2}$
Sel Out Wri In Memory cell $nMEM_{\sqrt{m}\sqrt{m}}$
Data out & Read/Write connection matrix: 2n x 2n√m lines
Data In
$I_1..I_n$
Data in connection matrix: n√m x n lines

Figure 2.8 The logical structure of proposed n port m cell static RAM chip. Chip select and power lines are not shown. The physical lay-out of components does not reflect the actual placement of semiconductor elements and is meant only for comparing singleport and multiport versions of memory chips. For the same reason the connection matrices are rounded with dotted lines and labelled with complexity numbers.

In CRCW Priority-style memories several processors can simultaneously write different data to the same memory cell. The processor with the lowest index succeeds. The correct operation is obtained by selecting the first inactive read/write line among all inactive read/write lines (see Figure 2.10). Write data is selected by a parallel priority encoder which activates all the read/write lines except the first inactive line.

A read/write operation is determined by ORing these lines together. The right input is selected among all the inputs by ANDing corresponding input line and output of parallel priority encoder. Obtained input lines can then be ORed together to form the input of the original cell. Finally, the output circuitry is duplicated for each port.

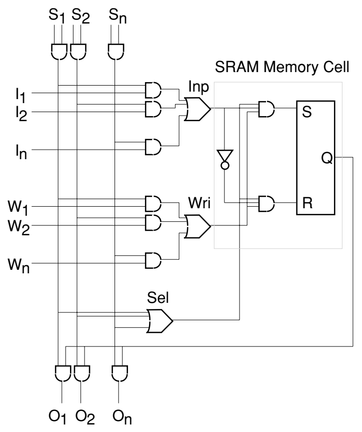


Figure 2.9 An example of a CRCW Common SRAM cell.

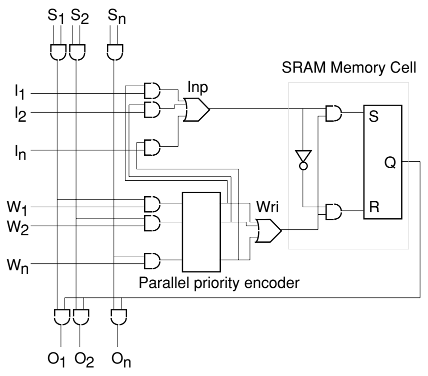


Figure 2.10 An example of a CRCW Priority SRAM cell.

The word length of a memory chip can be increased by duplicating data-in lines, data-out lines and memory cells for each bit of memory word. No changes are needed for cell selection circuitry, because a memory word is selected simultaneously. This simplifies the memory cell array relatively.

2.4.2 The cost-effectiveness of a multiport RAM

The relative cost-effectiveness of the multiport RAM is estimated by calculating wiring and component count complexity factors for ordinary and multiport memories. It is shown that multiport semiconductor RAMs look adequate in the sense of cost-effectiveness.

Suppose we are comparing a m location w-bit word SRAM and a P port m location w-bit word SRAM. Knowing that a log n-line to n-line parallel decoder has $n\log n + \log n$ components and a $2\log n \times 2n$ line matrix (reflecting the area taken by wiring) we get: Ignoring memory cells a P-port SRAM chip has P times more components ($0.5P(1+\sqrt{m})\log m$ compared to $0.5(1+\sqrt{m})\log m$) and P times more lines than a SRAM chip of the same size ($P\log m \times 2\sqrt{m}$ compared to $\log m \times 2\sqrt{m}$).

Both P-port and single port chips have the same count of memory cells, but in the multiport case the cells are more complex: Table 2.2 shows the complexity of memory cells in units of cell level gate inputs (reflecting the number of transistors in cells). The complexity of parallel N line priority encoder found in a CRCW Priority cell is assumed to be $N+2N\log N$ inputs rather than N^2+2N-1 inputs of unoptimized parallel priority encoder.

Type of memory	**Cell level input count**
Ordinary SRAM	$13w + 2$
CRCW Common	$5Pw + 6P + 11w$
CRCW Priority	$5Pw + 5P + 2P\log P + 9w$

Table 2.2 The complexity of RAMs (in units of cell level gate inputs)

Table 2.3 shows the complexity of memory cell array wiring in units of lines.

Type of memory	**Cell array lines**
Ordinary SRAM	$(w+1)\sqrt{m} \times (w+2)\sqrt{m}$
CRCW Common	$P(w+1)\sqrt{m} \times P(w+2)\sqrt{m}$
CRCW Priority	$P(w+1)\sqrt{m} \times P(w+2)\sqrt{m}$

Table 2.3 The complexity of RAMs (in units of connection matrix lines)

Simple calculations show now that P-port CRCW Common-style RAM requires P times more components and slightly less than P^2 times more silicon area in relation to single port RAM of same size. Assuming $P \leq 1024$, P-port CRCW Priority-style RAM requires $2P$ times more components and slightly less than P^2 times more silicon area.

From the complexity numbers we conclude that CRCW Common-style and CRCW Priority-style memories are slightly less than P^2 times less cost effective than ordinary memories of same size.

It is mainly the increase of wiring rather than the increase in the number of components that explodes the complexity of multiport RAMs when adding more ports. This makes the use of complex and powerful CRCW PRAM-style memories desirable over simpler and less powerful CREW PRAM-style memories currently available.

The size and port count of multiport memory chips in real multiprocessors will be quite limited, maybe within range of 4 kbit to 4 Mbit and 4 to 32 ports, because of the complexity of wiring. These estimations are based on the state of current VLSI technology.

2.4.3 The performance of the multiport RAM system

The performance calculations are favorable for PRAMs using multiport RAMs even with small count of processors. We will estimate the relative memory cycle time of a multiport RAM and the performance of multiprocessor using multiport RAMs.

If we do not count memory cells, the chip level delays are at least 3 gate delays for both ordinary RAMs and the proposed multiport RAMs. (One unit for chip selection and two units for address decoding.) The cell level write operation delays are shown in Table 4. The cell level read operation delay is one for both single port and multiport versions.

Type of memory	**Cell level**	**Chip level**
Ordinary SRAM	4	7
CRCW Common	6	9
CRCW Priority	7+log P	10+log P

Table 2.4 The length of cell level write operation datapaths and total chip level datapaths of RAM's (in units of gate delays)

The access times of multiport RAMs can likely be made small enough to fit in one clock cycle of a typical processor, because the total datapath of multiport RAM is only slightly longer than in ordinary RAM. If the port count is high, however, the CRCW Priority chips may be a little slower. Thus, the performance of a parallel machine using proposed chips is likely to achieve the performance of the ideal PRAM machine.

Relatively short cycle times of proposed memories have a dramatic effect on the memory system performance: A 16 processor PRAM-style computer using proposed chips (referred as 16-PRAM) would be as fast as a 10 000 processor network assuming the simulation of one PRAM step requires approximately 600 memory cycles on network (see Table 2.1).

In the case of sequential algorithms the difference is even greater: A 16-PRAM would run sequential algorithms even 600 times faster than 10 000 processor network. That is because 16-PRAM can run sequential algorithms as a fast as sequential machine, but global memory access takes so much time in machines with distributed memory.

On the other hand, there exist a lot of algorithms that do not require extensive communication and can therefore be executed efficiently on distributed machines. In these cases a PRAM style computer with a shared memory and small number of processors provides performance comparable to distributed machine with the same number of processors.

2.4.4 An example of a multiport RAM system within the limits of current technology

To give a rough idea of the complexity of multiport memories and the possibilities of the current technology, suppose for example, that we want to manufacture a 16 port 16kb x 1 CRCW Common-style static RAMs with access time of 20 ns. That should not be unrealistic because such chips are equally complex as currently available single port 4096kb x 1 static RAMs with access time of 15 ns. In fact, 7 to 128 decoders of a 16-port RAM are 50 times less complex than the 11 to 2048 decoders of a single port RAM.

The pin count of 16 port chips would be 290 instead of 28 in single port case. (The pin count of general w-bit word P port m location memory is $P(w + 2 + \log m) + 2$ pins assuming w bi-directional data lines.)

To build a true 16 port 1 MB shared main memory one needs 512 such 16 port 16kb x 1 CRCW Common-style chips. To reduce the complexity of the motherboard of parallel computers maybe SIMM-style packages could be used. The SIMM count of 1 MB memory would be 64 assuming one SIMM can hold 8 chips. The 1 MB 16 port memory bus would be at least 848 lines wide assuming byte addressing and 32-bit data bus. (The general P port M location W-bit word memory system bus is $P(W + 1 + \log M)$ lines wide assuming word addressing.)

2.5 Summary

We have proposed two possibilities to implement true PRAM-style shared main memories. The logical structure of building blocks, multiport RAMs is shown to be a simple extension of ordinary RAMs.

We have compared the relative cost-effectiveness of ordinary memories and multiport memories. The results are only adequate for multiport memories. P-port CRCW Common-style memories and CRCW Priority-style memories are slightly less than P^2 times less cost-effective than ordinary memories.

We have also compared the performance of some known message passing systems and the proposed multiport memory system. The results are favorable for multiport memories. Computers with multiport memories may be even hundreds of times faster than computers with message passing and distributed memory assuming the number of processors is the same.

The complexity of multiport memories will prevent the manufacturing of large PRAMs with multiport memory technology, but even with small port count computers with multiport memories will provide performance comparable to distributed memory systems with remarkably greater processor count.

Proposed chips will also provide interesting possibilities to increase the performance of distributed memory machines by replacing the sequential processors of the machine by small PRAMs using multiport memory technology.

The current research in the areas of wafer scale integration, multichip module production and optical communication may also introduce interesting new possibilities to exploit parallel processing technologies beyond the limits of current technology.

Our results concerning multiport memories are quite theoretical and there is lot of practical work to be done before we can take advantage of actual products. Anyway the challenge to build multiport memories is given. It is up to chip manufacturers to do appropriate development decisions.

References

[Abolhas91]
F. Abolhassan, J. Keller, W. J. Paul, On the Cost-effectiveness and Realization of the Theoretical PRAM Model, FB 14 Informatik, *SFB-Report 09/1991*, Universität des Saarlandes, Saarbrucken, 1991.

[Alt87]
H. Alt, T. Hagerup, K. Mehlhorn, F. P. Preparata, Deterministic simulation of idealized parallel computers on more realistic ones, *SIAM J. Comput.* **16**, 5 (1987), 808-835.

[Bilardi88]
G. Bilardi and K. T. Herley, Deterministic simulations of PRAMs on bounded degree networks, *Proceedings of 26th Annual Allerton Conference on Communication, Control and Computation* (1988).

[Blelloch89]
Guy E. Blelloch, Scans as Primitive Parallel Operations, *IEEE Transactions on Computers* **38**, 11 (1989), 1526-1538.

[Burnett70]
G. J. Burnett and E. G. Coffman, A Study of Interleaved Memory Systems, *AFIPS Conference Proceedings SJCC* 36, 1970, 467-474.

[Enslow74]
P. H. Enslow, *Multiprocessors and parallel processing*, John Wiley&Sons, New York, 1974.

[Feldman92]
Y. Feldman, E. Shapiro, Spatial Machines: A More Realistic Approach to Parallel Computation, *Communications of the ACM* **35**, 10 (1992), 61-73.

[Fortune78]
S. Fortune and J. Wyllie, Parallelism in Random Access Machines, *Proceedings of 10th ACM STOC*, Association for Computing Machinery, New York, 1978, 114-118.

[Gajski83]
D. Gajski, D. Kuck, D. Lawrie and A. Sameh, CEDAR-A Large Scale Multiprocessor, *Proceedings of International Conference on Parallel Processing*, 1983, 524-529.

[Gottlieb83]
A. Gottlieb, R. Grishman, C. P. Kruskal, K. P. McAuliffe, L. Rudolph and M. Snir, The NYU Ultracomputer - Designing a MIMD, shared-memory parallel machine, *IEEE Transactions on Computers* **C-32**, (1983), 175-189.

[Hillis85]
W. D. Hillis, *The Connection Machine*, The MIT Press, Cambridge, 1985.

[Karp69]
R. M. Karp and R. E. Miller, Parallel Program Schemata, *Journal of Computer and System Sciences* **3**, 2 (1969), 147-195.

[Leighton91]
T. F. Leighton, *Introduction to Parallel Algorithms and Architectures: Arrays, Trees, Hypercubes*, Morgan Kaufmann, San Mateo, 1992.

[McColl92]
W. F. McColl, General Purpose Parallel Computing, *Lectures on Parallel Computation Proceedings 1991 ALCOM Spring School on Parallel Computation* (Editors: A. M. Gibbons and P. Spirakis), Cambridge University Press, Cambridge, 1992, 333-387.

[Pfister85]
G. F. Pfister, W. C. Brantley, D. A. George, S. L. Harvey, W. J. Kleinfelder, K. P. McAuliffe, E.A. Melton, V. A. Norton and J. Weiss, The IBM Research Parallel Processor Prototype (RP3): Introduction and Architecture, *Proceedings of International Conference on Parallel Processing*, 1985, 764-771.

[Prechelt93]
L. Prechelt, Measurements of MasPar MP-1216A Communication Operations, *Technical Report 01/93*, Universität Karlsruhe, Karlsruhe, 1993.

[Ranade87]
A. G. Ranade, S. N. Bhatt, S. L. Johnson, The Fluent Abstract Machine, *Technical Report Series BA87-3*, Thinking Machines Corporation, Bedford, 1987.

[Schwarz66]
J. T. Schwarz, Large Parallel Computers, *Journal of the ACM* **13**, 1 (1966), 25-32.

[Schwarz80]
J. T. Schwarz, Ultracomputers, *ACM Transactions on Programming Languages and Systems* **2**, 4 (1980), 484-521.

[Tomasevic93]
M. Tomasevic and V. Milutinovic, A Survey of Maintenance of Cache Coherence in Shared Memory Multiprocessors, *Proceedings of the Hawaii International Conference on System Sciences*, Koloa, January 5-8, 1993.

[Valiant90]
L. G. Valiant, A Bridging Model for Parallel Computation, *Communications of the ACM* **33**, 8 (1990), 103-111.

Chapter 3

Efficient two-level mesh based simulations of PRAMs

Martti Forsell
Department of Computer Science, University of Joensuu, PB 111, FIN-80101 Joensuu, Finland

Ville Leppänen
Department of Computer Science, University of Turku, Lemminkäisenk 14, FIN-20520 Turku, Finland

Martti Penttonen
Department of Computer Science, University of Joensuu, PB 111, FIN-80101 Joensuu, Finland

Abstract

We consider time-processor optimal simulations of PRAM models on coated block meshes. A coated block mesh consists of π-processor blocks and $\pi \times \pi$ or $\sqrt{\pi} \times \sqrt{\pi} \times \sqrt{\pi}$ router blocks. The router blocks form a 2-dimensional or a 3-dimensional regular mesh, and the processor&memory blocks are located on the surface of the block mesh. As a generalization of the coated mesh, the 2-dimensional and 3-dimensional coated block meshes simulate EREW, CREW, and CRCW PRAM models time-processor optimally with moderate simulation cost. Using proper amount of parallel slackness, the cost can be decreased clearly below 2 routing steps per simulated PRAM processor. The coated block mesh is actually an instance of a more general two-level construction technique, which uses a seemingly inefficient but scalable solution on top of a non-scalable but efficient solution. Within blocks (chips) brute force techniques are applied, whereas the mesh structure on top makes the whole construction modular, simple, and scalable. The parameter π provides a method to balance the construction with respect to the two techniques.

Keywords: PRAM, shared memory machine, simulation, time-processor optimal, mesh, interconnection network

3.1 Introduction

The standard approach to PRAM (defined in Section 3.2.1) implementation is to map the shared memory into a collection of P distributed memory modules, to attach a memory module to each real processor, and to map the N virtual processors of the PRAM to the P real processors. Each real processor is assumed to simulate at most $\lceil N/P \rceil$ *virtual processors* (threads). The operations on shared memory are translated to messages, which are routed from the sources to the targets via a routing mechanism. The simulation itself is done stepwise in order to make easier the preservation of atomic consistency.

Diameter $ø$ of a routing machinery usually determines a lower bound for the latency of messages. For many popular networks, $ø = \Omega(\log P)$. If $N=O(P)$, the latency often determines the PRAM simulation cost. *Parallel slackness* [Valiant90] ($N>P$) can be used to balance the cost of local computation and global communication, and thus to improve the efficiency of PRAM simulations. In Valiant's XPRAM framework [Valiant90], the simulation is done in supersteps of length L, where $L = \Omega(N/P)$. During a superstep, the real processors asynchronously do local computations, and communicate with each other in order to simulate the current step of the virtual processors. A superstep ends to a barrier synchronization, which is used to check if all the processors have managed to simulate their virtual processors, or if another superstep is needed. Balancing is done by choosing N, L, and the routing machinery so that, all the communication requests of the N virtual processors can be delivered in time $O(N/P)$ (with high probability). The simulation is said to be *time-processor optimal*, if this is achieved.

3.1.1 Previous results

Many time-processor optimal PRAM simulation results have been proved for completely connected graphs ((i)-type) [Diezfelbinger93, Goldberg94, Meyerauf92, Valiant90]. In [Goldberg94], Goldberg, Matias, and Rao provide so far the best (asymptotically smallest latency) time-processor optimal $P \log \log P$-processor EREW PRAM simulation.

If the $ø$ is not $O(1)$, then to hide the delay we must either assume that each node can deal with $\Omega(ø)$ routing requests at each step ((ii)-type), or that the routing machinery is separated from the processor&memory pairs and is considerably larger than P ((iii)-type). In the latter case, the nodes only need to make simple routing decisions, and therefore it might be acceptable to ignore their hardware complexity and work.

In [Valiant90], Valiant proves a (ii)-type EREW result for a P-node binary hypercube. He also gives a (iii)-type EREW result for a P-input butterfly. Similar results are also established for most of the CRCW PRAMs in two ways: By extending the routing machinery with combining [Ranade91], or by requiring that $N=P^{1+\varepsilon}$ for some $\varepsilon>0$ [Valiant90]. In [Leppänen95b], we prove (iii)-type results for EREW and CRCW PRAMs by taking the routing machinery to be a 2D mesh or a 3D mesh. Both for EREW and CRCW simulations, time-processor optimality is achieved, when $N=\Omega(P^{d/d-1})$. General leveled network results [Leighton94] imply that (iii)-type results can also be shown for several other networks.

3.1.2 Some critique

Because of routing machinery degree, the completely connected graphs are not very realistic with current technology. Also Valiant's hypercube result is based on some quite unrealistic assumptions. The degree log P makes it difficult to expand the hypercube physically. The lengths of connections increase, when the system is scaled up. Moreover, it is somewhat unfair to assume that a processor can deal with all of its log P input and output links in unit time.

Type (iii) results also have a price to pay: The routing machinery is large in terms of the number of nodes. However, the hardware complexity ratio of a processor&memory pair and a routing machinery node might be significant, e.g., of one or two orders of magnitude. Due to non-constant connection length, the unit time assumption and scalability are problematic in the butterfly [Ranade91, Abolhassan93] but not in our mesh based solution. In the 3D case, the routing machinery size is only $\sqrt{P}/6^{1.5}$ times P [Leppänen95b]. If $P=10^4$, this ratio is ≈ 7. For the butterfly, the same ratio log $P+1\approx 15$.

The proposed simulations differ also with respect to the required parallel slackness. To achieve constant simulation cost per PRAM processor, slackness level $\Omega(\o)$ is required. The cost decreases as the slackness level increases. Supporting large number of threads per each processor is possible (Tera [Alverson90]: 128 threads per processor; SB-PRAM: 32×32 threads), but the thread requirement sets restrictions for successful programming, since achieving a processor utilization level requires certain amount of threads per processor. However, the actual "cost" of some slackness level is unclear.

3.1.3 Contributions

We propose an improvement to the coated mesh [Leppänen95b] in form of a coated block mesh. We group a bunch of routing machinery nodes to a router block by using a sparse matrix type solution within each router block. Consequently, the ø and slackness requirement decrease. Also the ratio of the number of elementary routing machinery nodes and processors improves by factor of $\pi^{d/d-1}$ in the d-dimensional case. The processors we divide to groups of π processors, and integrate one shared memory module to each group. Each processor&memory block is considered as a small PRAM.

The idea in both the router and processor&memory blocks is that in small scale one can use special constructions (brute force) without introducing any serious restrictions on the overall scalability. Brute force solutions can be very efficient, if applied within chips. By integrating somewhat costly small scale solutions to an elegant large scale solution, we hope to produce an efficient and scalable overall architecture solution. Large scale solutions tend to be slow in small scale whereas fast special solutions are often non-scalable. (Similar design principle is recently applied in constructing practical expanders [Brewer94].) By properly choosing π, we hope to find solutions, where the cost of communication, memory access, and thread management primitives are properly balanced.

In Section 3.3, we show that the coated block mesh can efficiently simulate PRAM model. In asymptotic sense the same results can be derived from certain leveled network results [Leighton94]. However, the theme of this paper is to characterize the cost more precisely, and to observe that in practice it is very small. Our EREW simulation experiments indicate that the cost can be decreased below 2 in the 3D case, and to 1.1 … 1.3 in the 2D case. The latter means that 75% to 90% of the raw processing power can be given to arbitrary PRAM computations. In our results we assume that each PRAM processor makes an external memory reference at each step. This is hardly the case in practice, and thus achieving smaller simulation cost is possible.

3.2 Definitions

3.2.1 PRAM

The PRAM model consists of N processors, P_0, P_1, …, P_{N-1}, and a shared memory M of size m. Each P_i has some local memory and registers. A step of P_i consists of a local operation, reading a shared memory location, or writing to a shared memory location. The phases of a step as well as steps are executed synchronously (system wide). A read instruction returns the value of memory location in question before the current step. In the EREW model concurrent reading and writing of any particular shared memory cell is forbidden, but each shared memory cell may be read and written once during a step. In the Arbitrary CRCW model concurrent read and write operations are allowed. If two or more processors write to a memory location in a given step, one of the values is selected arbitrarily to become the new value.

3.2.2 Coated block mesh

Our *coated block mesh* (CBM) is a generalization of the coated mesh (CM) [Leppänen95b], which consists of a 2D mesh or a 3D mesh based routing machinery that is coated with processor&memory pairs (see Figure 3.1). In the CBM, π processors and their memory modules are integrated to a processor&memory block (pm-block). Respectively, a bunch of routing machinery nodes are integrated to a router block. The coated block mesh is then composed of blocks as the coated mesh is composed of router nodes and processor&memory pairs. In a way, the routing task is partially moved from the routing machinery to the pm-blocks.

Formally, a dD coated block mesh $CBM(P, \pi, d, q)$ consists of P/π pm-blocks A_0, A_1, … $A_{P/\pi-1}$ of π processors and one shared memory module, and $(P/(\pi 2d))^{d/d-1}$ router blocks $B_{i1,\ i2,\ \ldots,\ id}$, where $0 \le i_j < (P/(\pi 2d))^{d/d-1}$ and $1 \le j \le d$. Each router block has π bidirectional links on each of its $2d$ surfaces, and is capable of receiving and sending one message per link in one time unit. The router blocks $B_{i1,\ i2,\ \ldots,\ id}$ are connected as a regular dD mesh.

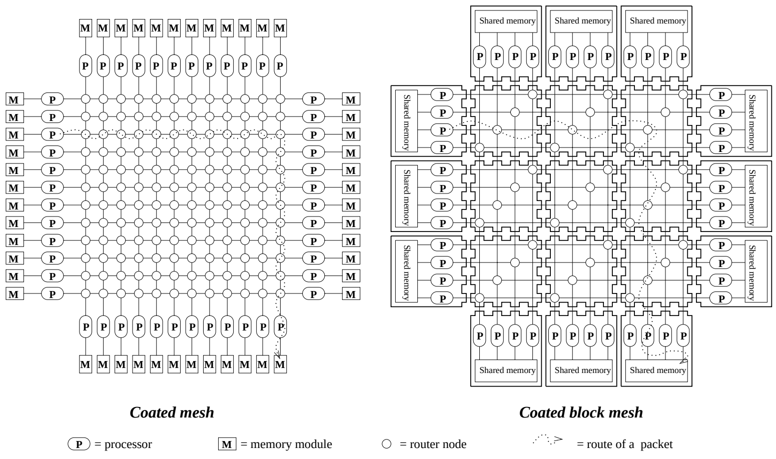


Figure 3.1 Coated mesh versus coated block mesh (π=4) in 2-dimensional case.

3.2.2.1 Router block

We assume that the internal structure of a router block is a d-dimensional $\sqrt[d-1]{\pi}$-sided sparse matrix with π router nodes. Each directed link has a buffer of size q packets. A packet entering $B_{i_1, i_2, \ldots, i_d}$ via link k on surface j, can leave the block only via link k on some surface l. The function of router nodes is defined by the PRAM simulation algorithm.

Assume that the nodes of a router block and their I/O-entries are located in a d-dimensional $\sqrt[d-1]{\pi}+2$-sided space. Then, the I/O-entries are on positions $(x_1, x_2, \ldots, x_d)$, where exactly one of $x_1, x_2, \ldots, x_d$ is either 0 or $\sqrt[d-1]{\pi}+1$.

The I/O-entries having the same coordinate x_i equal 0 (or $\sqrt[d-1]{\pi}+1$) form surface S_{2i} (respectively S_{2i+1}). The indexing of I/O-entries on surface S_j is a mapping f_j: $\{1,\ldots,\sqrt[d-1]{\pi}\}^{d-1} \rightarrow \{0,\ldots,\pi-1\}$. For constructibility we want the connections within router blocks to be axial. Therefore we define the indexing on surfaces S_{2i} and S_{2i+1} to be mirror images of each other, i.e., $f_{2i}(a_1, a_2, \ldots, a_{d-1}) = f_{2i+1}(a_1, a_2, \ldots, a_{d-1})$ for each $0 \le i \le d$. To complete the definition of internal structure of a router block, we also assign an indexing g to the internal nodes. Partial injective mapping g is a mapping from $\{1,\ldots,\sqrt[d-1]{\pi}\}^d$ to $\{0,\ldots,\pi-1\}$. There is one requirement concerning g: Each router node x and its projective image on any of the $2d$ surfaces must have the same index.

Consequently, a projective mapping of g^{-1} with respect to any surface covers the whole surface. By defining g and one f_j, we at the same time define rest of the f_j's. Clearly, the definition of f_0 is irrelevant as long as it is bijective. We use $g_2^{-1}(k) = \{k,k\}$ in the 2D case, and $g_3^{-1}(\sqrt{\pi} \times k+l) = \{k+1, l+1, ((k+l) \bmod \sqrt{\pi}) + 1\}$ in the 3D case ($0 \leq k, l < \sqrt{\pi}$).

3.2.2.2 Processor&memory block

The pm-blocks "coat" the whole surface of router blocks based routing machinery. The degree of each processor (pm-block) is 1 (π). We assume that each processor can receive and send a packet in one time unit. For simplicity, we take the memory consistency model of the shared memory modules to be the same as the whole CBM is simulating. We do not define, how the pm-blocks are implemented, but assume that fulfilling any proper memory request takes unit time. If π is a small constant, possible implementations are, e.g., a π-port memory [Forsell94], and a complete graph based π-processor DMM.

3.3 Simulation of PRAM models cbm-simulation

A somewhat standard approach [Ranade91,Valiant90] to simulate the PRAM models is to assume that the shared memory of the PRAM is mapped with a randomly chosen hash function $h \in H$ into the distributed memory modules; in our case, into the local memory of pm-blocks. Then, the simulation is done by translating the memory references to packets, which are routed via the routing machinery to targets. Replies are generated to the read requests, and routed back to senders of the read requests. Finally, the system is completed with a synchronization mechanism, which guarantees that the simulation of consecutive steps maintains the atomic memory consistency of the PRAM.

The idea in shared memory hashing is to initially arrange the shared memory according to a randomly chosen hash function h, and to rearrange the memory by choosing a new hash function $h_{\text{new}} \in H$, if h turns out to be bad for the memory access patterns produced by the programs executed on the PRAM.

Although each pm-block has the same memory consistency model as the simulated PRAM model, we still must ensure that the write requests will not influence the read requests on the current step. We can think of several approaches to solve the problem. Here we assume that the arriving requests are executed on fly, but simultaneously old memory values are temporarily preserved. Such a construction could be based on delayed write, where each memory cell is duplicated, and the delayed write instructions take effect after the synchronization point between separate PRAM steps is reached.

3.3.1 Simulation of EREW PRAM cbm-EREW

The following algorithm is derived from the EREW simulation algorithm presented in [Leppänen95b] for the CM.

Algorithm 3.3.1 (EREW simulation on 2D CBM)

P_i simulates PRAM processors $P_{iN/P}, \ldots, P_{(i+1)N/P-1}$.

1. Translate memory references to packets, and inject them uniformly to the routing machinery.
2. Route the requests greedily to their target, and execute the memory accesses.
3. Synchronize processors.
4. Build replies, and inject them uniformly to the routing machinery.
5. Route the replies back greedily.
6. Synchronize processors, and update virtual processor registers.

In [Leppänen95b] we proved that using the above algorithm and polynomial hash functions, a 3D (2D) P-processor CM can simulate $P^{3/2}$-processor (P^2-processor) EREW PRAM time-processor optimally with probability $1\text{-}P^{-\alpha \log \log P}$ for any $\alpha>1$. Because the coated block mesh basically consists of π separate coated meshes (with P/π processors in each), the simulation succeeds with high probability in each of them simultaneously, if $\pi = O(P/\pi)$. By “high probability”, we mean a probability of the form $1\text{-}P^{\gamma}$ for some constant $\gamma>1$. Clearly, the coated mesh result can be extended to Theorem 3.3.2.

Theorem 3.3.2

A $CBM(P,\pi,d,q)$ can simulate an $N=P\times(P/2d\pi)^{1/d-1}$-processor EREW PRAM time-processor optimally using at most

$36 + o(1)$, if $d=2$
$78 + o(1)$, if $d=3$

routing steps per simulated PRAM processor with high probability, if $\pi=O(\sqrt{P})$.

3.3.2 Simulation of CRCW PRAMs cbm-CRCW

In [Leppänen95b] we proved that 2D and 3D coated meshes can simulate the Arbitrary CRCW time-processor optimally (with high probability), if $N = \Omega(P^{d/d-1})$. Extending the simulation result to the CBM is not as straightforward as in the EREW case, since CRCW simulation is based on sorting, and implementing sorting on the CBM is not trivial. However, it is enough to apply sorting within each of the π logical coated meshes separately to maintain the memory consistency. We state Theorem 3.3.3 without proof. Proving it requires similar assumptions as in the EREW case. For details of the CRCW simulation algorithm, see the details the simulation on the coated mesh [Leppänen95].

Theorem 3.3.3
A $CBM(P,\pi,d,q)$ can simulate an $N=P\times(P/2d\pi)^{1/d-1}$-processor Arbitrary CRCW PRAM time-processor optimally using at most

$66 + o(1)$, if $d=2$
$153 + o(1)$, if $d=3$

routing steps per simulated PRAM processor with high probability, if $\pi=O(\sqrt{P})$.

3.4 Improvements

A straightforward method to improve simulation cost is to increase the parallel slackness factor of simulation, i.e., the load N/P of each physical processor. Next, we briefly discuss another method to improve the simulation cost.

Separating consecutive steps by naive synchronization attempts is expensive, and we would like to apply a more flexible synchronization method. Such a technique is *synchronization wave*. A synchronization wave is sent by the processors (sources) to the memory modules (destinations) and vice versa. This technique has been successfully used for PRAM simulation on the butterfly [Ranade91] and on the mesh of trees [Leppänen95a]. The idea is that when a source has sent all its packets on their way, it sends a synchronization packet. Synchronization packets from various sources push on the actual packets, and spread to all possible paths, where the actual packets could go. When a node receives a synchronization packet from one of its inputs, it waits, until it has received a synchronization packet from all of its inputs, then it forwards the synchronization wave to all of its outputs. The synchronization wave may not bypass any actual packets and vice versa. When a synchronization wave sweeps over a leveled network based routing machinery [Leighton94]—or more generally a directed acyclic graph (DAG), all nodes and processors receive exactly one synchronization packet via each input link and send exactly one via each output link.

The CBM is not directly a leveled network, but it can be seen as one, or more precisely, each CM can be seen as a virtual leveled network. Embedding a leveled network to a mesh is described in [Leighton94]. Their technique is improved in [Leppänen96b], and applied to coated meshes by embedding a DAG to a CM with duplicated connections. The idea is that each routing machinery node has d unidirectional “injection” and d unidirectional “removal” edges (one for each axis), and d bidirectional “normal” edges. A packet is routed by first choosing randomly an intermediate target from the closer half of the pile of nodes axial to the source processor. After arriving to the first intermediate target, the packet advances along the normal edges (greedy XY-routing) and moves to the second randomly chosen intermediate target at the closer half of the pile of nodes axial to the target processor. Finally, the packet is routed to the target by using the removal edges. The above routing process in a natural way organizes all unidirectional connections of a CM and a CBM (with duplicated connections) to a DAG.

Continuous synchronization wave based EREW simulation can now be derived by setting one DAG from processors to memory modules (requests) and another from memory modules to processors (replies), and letting the synchronization wave reflect from one DAG to the other (see [Leppänen96b]). We denote such a coated block mesh by CBM$^+$.

3.5 Experimental results cbm-experiments

We run about 400 experiments simulating altogether approximately 20000 steps of an N-processor EREW PRAM on 2D and 3D CBM$^+$s. We assume that each unidirectional connection has a buffer of size q packets, only the head of each queue resides at the receiving end, and a routing machinery node can only move packets from the heads of incoming queues to the tails of outgoing queues (if there is room). Each processor can receive and send at most 1 packet per routing step.

In practice, we expect a randomly chosen hash function to turn typical EREW memory reference patterns to almost random ones. However, being short of typical EREW reference patterns, we did choose the destinations of memory references randomly (using Unix function *random*). The only additional assumption about the computation is that the number of PRAM processors remains the same throughout the computation. More details about experiments and results can be found in [Leppänen96a].

3.5.1 Continuous EREW simulation

As a reference point for measuring the time to simulate a single PRAM step, we use the moment when the last "part" of a synchronization wave leaves the processors. The continuous simulation means that each processor processes the requests and replies (and the synchronization wave) in the order they arrive, and injects replies and new requests back to the routing machinery as soon as possible. Next, we study the dependence of simulation cost $c(\lambda, ø, \pi, d, q)$ on the diameter $ø$ of the routing machinery, the load $N/P = \lambda ø$ per processor, π, and q on dD CBM$^+$s. The simulation cost is calculated as an average time over 30 to 50 simulated EREW steps divided by the load.

3.5.1.1 2-dimensional case cbm-EREW-mvt-sync-2d

We run our 2D simulation experiments on $CBM^+(4\pi\times10,\ \pi,\ 2,q)$ and $CBM^+(4\pi\times50,\ \pi,\ 2,q)$, where the diameter $ø$ is 40 and 200, respectively. The influence of buffer size q on the simulation cost on 10×10 CBM$^+$ is shown in Figure 3.2. Even, when q=2 the simulation cost decreases below 2. However, with buffers of size 16 or 32 one can achieve cost 1.1 … 1.2. The cost 1 is the best one can achieve. The curves of Figure 3.2 are obtained by using π=1. To our surprise, the same curves for π=4, 8, 16, and 32 are identical, or off by cost 0.1. Thus by applying π=32 (instead of 1), one can decrease the diameter $ø$ to 1/32'th and consequently also the required load to achieve certain cost!

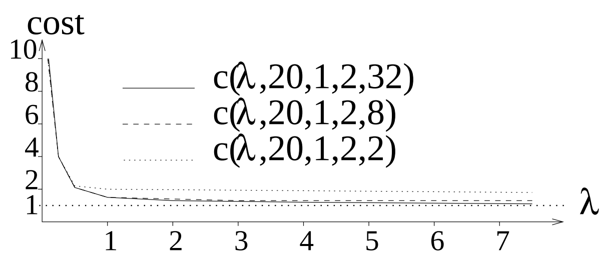


Figure 3.2 EREW simulation cost on 2D CBM$^+$.

The curves for 50×50 CBM$^+$ were practically identical for those of Figure 3.2 (occasionally off by 0.1). We conclude that the simulation cost depends strongly on λ and q, but not on ø. Small π-values seem to have minor effect on the simulation cost, and thus the advantage obtained via smaller diameter seems to translate linearly to smaller parallel slackness advantage. Clearly, the cost function begins to show asymptotic behavior, when λ=1 … 2.

3.5.1.2 3-dimensional case

Our results concerning simulation of EREW on 3D CBM$^+$ are very similar to those for 2D CBM$^+$ (see [Leppänen96a] for details). Again, the simulation cost mainly depends on q and λ. Small π-values and ø seem to have practically no effect on $c(\lambda,ø,\pi,3,q)$. Now the advantage provided by π does not imply linearly smaller parallel slackness level, but only $\approx\sqrt{\pi}$ times smaller level. The curves show asymptotic behavior when $\lambda>1$, and at best cost 1.68 was achieved (with $\lambda=5$, $q=32$). This is clearly worse than in the 2D case. The reason is that in the 3D CBM$^+$, the routing machinery bandwidth is simply too small [Leppänen96b] for all PRAM processors to make a shared memory reference at every step. Insufficient bandwidth makes the cost approach 1.5 in the 3D case.

3.6 Conclusions

We have proposed the coated block mesh as a platform for time-processor optimal implementation of PRAM models. The CBM is a generalization of the CM with respect to the integration degree π of processing and memory elements. The coated block mesh itself has many desirable properties.

- As a mesh based construction, it has a regular and feasible layout.
- The whole structure is modularly extendible. E.g., the hypercube lacks this property. The modularity also suggest fairly easy physical maintenance.
- The CBM is truly scalable in the sense that x-folding the physical system (x times more processors) provides x times more computational power, if the processors can support approximately $x^{1/d-1}$ times more threads per processor.
- The connections between logical neighbors (processors/nodes) are of small constant length.

When claiming our PRAM implementations time-processor optimal, we ignore the work and hardware complexity of the router blocks. If d=3, then the ratio of router and processor&memory blocks is $\sqrt{P}/6^{1.5}\times\sqrt{\pi}$. If $\pi = 4$ and $P = 86400$, then this ratio is only 10. On the other hand, the processors, and especially the memory modules, are apparently more expensive in terms of hardware than the routing machinery nodes.

The CBM is an application of more general design method: It applies a theoretically elegant and scalable but rather inefficient solution on top of a non-scalable but efficient small scale solution. The blocks represent non-scalable but efficient solutions whereas the mesh structure represents the elegant part. The parameter π provides a method to balance the cost of elementary operations: memory access, thread management, and communication with neighboring elements.

Our experiments show that by using the synchronization wave+DAG approach, the cost can be pushed below 2 in the 3D case, and very close to 1 in the 2D case. These results are based on the assumption that on each step each PRAM processor makes an external reference on every step! By processing threads that make local references during the “idle” cycles, further performance improvements can be expected. The larger the π-value the smaller the diameter and the smaller the required parallel slackness level to achieve a certain simulation cost. Experiments indicate that load $\approx\varnothing$ is enough to achieve good performance in the 2D and 3D cases. To our surprise, even relatively large π-values (32) seem to have no influence on $c(\lambda, \varnothing, \pi, d, q)$. Thus, the smaller diameter advantage of the CBM over the CM translates directly to smaller parallel slackness requirement advantage.

3.6.1 Topics for further research

The two-level approach itself seems healthy, but can one apply it more efficiently in another context—i.e., would it be better to use other large scale and small scale methods. One could also study several variants of our coated block mesh. Instead of one π-degree memory module, we could have ρ π-degree or π/ρ-degree memory modules per pm-block. We could also extend the degree of processors from 1 to ρ. That would make the delivery of packets harder but perhaps improve the constructibility of memory modules. If the router blocks are very big, we could consider locating τ pm-blocks on the surface of each instead of one. Supporting $\tau=\rho$ processor&memory blocks would be rather straightforward from the routing point of view.

References

[Abolhassan93]
F. Abolhassan, R. Drefenstedt, J. Keller, W Paul and D. Scheerer, On the Physical Design of PRAMs, *The Computer Journal* **36**, 8 (1993), 756-762.

[Alverson90]
R. Alverson, D. Callahan, D. Cummings, B. Koblenz, A. Porterfield and B. Smith, The Tera Computer System, *Computer Architecture News* **18**, 3 (1990),1-6.

[Brewer94]
E. Brewer, F. Chong and F. Leighton, Scalable Expanders: Exploiting Hierarchical Random Wiring, *Proceedings of the 26th Symposium on Theory of Computing*, 1994, 144-152.

[Dietzfelbinger93]
M. Dietzfelbinger and F. Meyer auf der Heide, Simple, Efficient Shared Memory Simulations, *Proceedings of the 5th Symposium on Parallel Algorithms and Architectures*, 1993, 110-119.

[Forsell94]
M. Forsell, Are Multiport Memories Physically Feasible?, *Computer Architecture News* **22**, 5 (1994), 3-10.

[Goldberg94]
L. Goldberg, Y. Matias and S. Rao, An Optical Simulation of Shared Memory, *Proceedings of the 6th Symposium on Parallel Algorithms and Architectures*, 1994, 257-267.

[Leighton94]
F. Leighton, B. Maggs, A. Ranade and S. Rao, Randomized Routing and Sorting on Fixed-Connection Networks, *Journal of Algorithms* **17**, 1 (1994), 157-205.

[Leppänen96a]
V. Leppänen, Experimental PRAM on Coated Block Mesh Simulation Results, *In preparation*, 1996.

[Leppänen95a]
V. Leppänen, On Implementing EREW Time-Processor Optimally on Mesh of Trees, *Journal of Universal Computer Science***1**, 1 (1995), 23-34.

[Leppänen96b]
V. Leppänen, Performance of Work-Optimal PRAM Simulation Algorithms on Coated Meshes, *to appear in The Computer Journal*, 1996.

[Leppänen95b]
V. Leppänen and M. Penttonen, Work-Optimal Simulation of PRAM Models on Meshes, *Nordic Journal on Computing* **2**, 1 (1995), 51-69.

[Mayerauf92]
F. Meyer auf def Heide, Hashing Strategies for Simulating Shared Memory on Distributed Memory Machines, *Parallel Architectures and Their Efficient Use*, LNCS 678, 1992, 20-29.

[Ranade91]
A. Ranade, How to Emulate Shared Memory, *Journal of Computer and System Sciences* **42**, (1991), 307--326.

[Valiant90]
L.G. Valiant, General Purpose Parallel Architectures, In Algorithms and Complexity, *Handbook of Theoretical Computer Science*, 943-971, 1990.

Chapter 4

Minimal pipeline architecture —an alternative to superscalar architecture

Abstract

Pipelining is used in almost all recent processor architectures to increase the performance. It is, however, difficult to achieve the theoretical speedup of pipelining, because code dependencies cause delays in execution. Superscalar processor designers must use complex techniques like forwarding, register renaming and branch prediction to reduce the loss of performance due to this problem. In this paper we outline and evaluate abstract Minimal Pipeline Architecture (MPA) featuring cross-bar interconnect of functional units and special two level pipelining. According to our evaluation MPA is not more complex than a basic superscalar architecture using out of order execution, but offers remarkably better performance.

Keywords: computer architecture, pipelining, superscalar processor, VLIW

4.1 Introduction

Pipelining is very popular technique in increasing the performance of processors. This is because ideal pipelining gives the speedup of the number of pipeline stages with only marginal hardware effort. In practice it is, however, difficult to achieve the speedup of ideal pipelining, because data dependencies and branches cause delays in execution [Kogge81, Flynn95, Hennessy90].

Reprinted from Microprocessors and Microsystems, 20 5 Martti Forsell, Minimal pipeline architecture—an alternative to superscalar architecture, 277-284, Copyright © 1996 with kind permission of Elsevier Science - NL, Sara Burgerhartstraat 25, 1055 KV Amsterdam, The Netherlands.

Superscalar processor designers must use complex techniques like forwarding, register renaming and branch prediction to reduce the loss of performance due to this problem [Tomasulo67, Smith84, Johnson89].

This paper introduces four techniques for minimizing the performance penalty of pipelining—general forwarding, simple instruction set, uncoded instructions and fast branching. They both reduce the depth of deep pipelines and eliminate unnecessary pipelining. These techniques are applied to a basic VLIW core to outline a new abstract architecture featuring cross-bar interconnect of functional units and two level pipelining. We call the obtained architecture as Minimal Pipeline Architecture (MPA).

The paper gives also a performance, code size, functional unit utilization and complexity evaluation of two instances of MPA and basic pipelined and superscalar architectures. According to the evaluation cross-bar based MPA is not more complex than a basic superscalar architecture using out of order execution, but offers remarkably better performance. In addition to that MPA offers interesting advantages over architectures using deeper pipelines: Pipeline delays as well as register file bandwidth problems are absent in MPA. Also the utilization of functional units is better in MPA than in basic pipelined RISC machines.

4.2 Dependencies and execution models

There are two basic models for exploiting instruction level parallelism—pipelined and superscalar execution model.

In the *pipelined* model [Kogge81, Flynn95] the execution of instructions is divided into parts called pipeline stages, such that the different parts of consecutive instructions can be executed simultaneously.

In the *superscalar* model [Johnson89, Hennessy90] multiple instructions are executed in multiple functional units simultaneously. The pure superscalar execution model is, however, very rare. It is usually combined with the pipelined model such that the execution of instructions over functional units is pipelined.

The execution of instructions in both models can be divided into two clearly separate parts—instruction fetch (IF) and instruction execute

(IE)—which are always committed sequentially. Furthermore, the IE part is usually divided into three or more subparts.

Let us denote the number of these subparts increased by one (representing the effect of IF part) with S. In the pipelined execution model S clearly equals the number of pipeline stages.

4.2.1 Non-pipelined execution

An execution model is *non-pipelined* if no explicit division of instruction execution into parts is made, i.e., the latency of instruction equals the length of clock cycle.

Dependencies do not cause delays in the scalar non-pipelined execution model, because instructions are executed in order and no overlapping happens. This model is, however, inefficient—the ideal S-level pipelined execution model performs S times better than this model.

4.2.2 Pipelined execution

Let us assume that we are dealing with the S-level pipelined execution model without any specific knowledge about the IE part of the execution.

It is quite obvious that data dependencies in the code cause delays in execution if IE parts of dependent instructions overlap (see Figure 4.1).

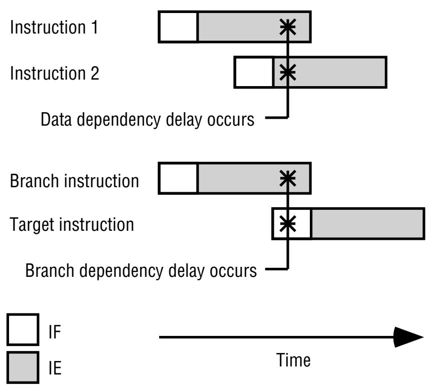


Figure 4.1 Situations where data and branch dependency delays occur.

Let l_X be the latency of the part X. Now we can say, that no data dependency delay exists if following equation holds:

$$l_{IE} \leq l_{IF} \tag{4.1}$$

Similarly, no delay is caused by a branch instruction if the IF part of the branch target instruction does not overlap with the IE part of the branch instruction:

$$l_{IE}(\text{Branch instruction}) \leq 0 \tag{4.2}$$

These two equations suggest that dependency delays can be avoided with two level pipelining where the latency of IE part of branch instructions is zero.

4.2.3 Superscalar execution

In the superscalar model data and branch dependencies cause similar delays as in the pipelined model.

Some superscalar architectures use internal buffers, which allow out of order execution of decoded instructions [Johnson89]. This leads to good exploitation of available instruction level parallelism, because execution is now limited only by actual data dependencies. Unfortunately fetching, decoding and buffering of decoded instructions take some time causing a lot of branch delays. The performance degradation due to these delays can be partially eliminated by branch prediction and speculative execution [Smith84, Johnson89].

A much simpler alternative to a superscalar processor is a *Very Long Instruction Word* (VLIW) processor [Fisher83, Nicolau84]. A VLIW processor consists of multiple functional units that operate in parallel controlled by a single sequencer. Advanced compilers are used to parallelize the program before the execution in a VLIW processor [Fisher81, Chang91].

4.3 Implementation techniques

This section describes the four key implementation techniques we used in designing MPA—general forwarding, simple instruction set, uncoded instruction format and fast branching.

4.3.1 General forwarding

The pipeline depth of four or more stages is very usual in current RISC machines. This causes a lot of data dependency delays, because the left side of equation 4.1 becomes at least three times greater than the right side.

An old hardware technique called *forwarding* eliminates the data dependency delays presuming the actual operation is executed in a single fixed stage of the pipeline [Hennessy90] (see Figure 4.2).

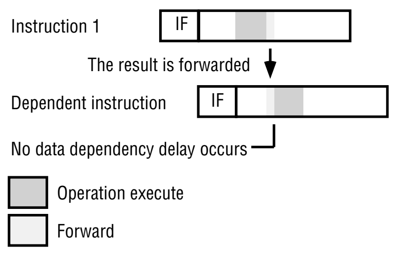


Figure 4.2 Forwarding eliminates data dependency delays.

We get a novel technique called *general forwarding* by applying the idea of forwarding to the selection of all operands: The results of functional units including the contents of registers are passed to the input multiplexers of functional units. The correct operands are selected by appropriate bits in the operation code of an instruction.

The use of general forwarding requires simultaneous access to all registers, which requires quite complex wiring with ordinary multiported register files.

Simultaneous access to all registers can be implemented by a novel register file organization called *distributed register file*, in which single registers are treated as separate functional units that are connected to each other by a cross-bar switch-like forwarding mechanism (see Figure 4.3).

In a processor using general forwarding the separate operand select stage is unnecessary, because the forwarding machinery now selects operands between the operations of consecutive stages.

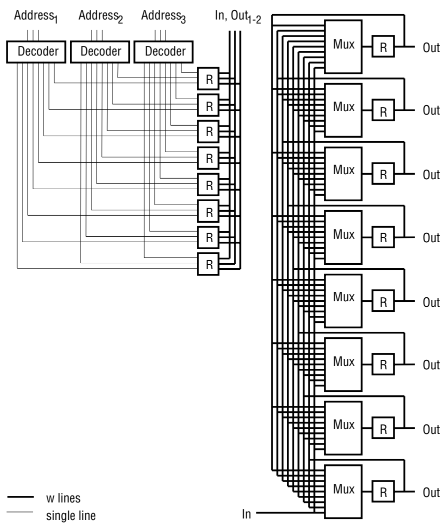


Figure 4.3 A conventional register file with two read ports and one write port (left). A distributed register file with one external input and eight outputs (right).

4.3.2 Simple instruction set

The instruction sets of basic RISC processors do not support execution in two level pipeline properly, because most instructions require sequential involvement of more than just one operational unit (see Figure 4.4).

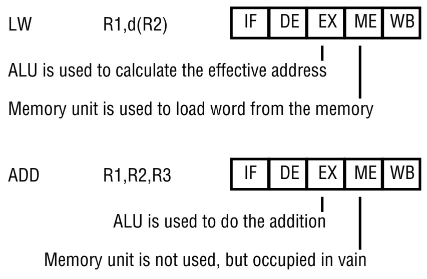

LW R1,d(R2)
IF DE EX ME WB
ALU is used to calculate the effective address
Memory unit is used to load word from the memory
ADD R1,R2,R3
IF DE EX ME WB
ALU is used to do the addition
Memory unit is not used, but occupied in vain

Figure 4.4 Some instructions occupy many functional units in DLX pipeline [Hennessy90].

The problem disappears by using a *simple instruction set*: Instructions involving the use of multiple functional units are divided into multiple instructions occupying exactly one functional unit and using always absolute or indirect addressing.

A processor using simple instruction set potentially offers better utilization of functional units than a conventional RISC processor using typical instruction set. This is because one has more accurate control of functional units with the simple instruction set. There are for example no situations where an instruction occupies a functional unit in vain (see the ADD instruction in figure 4.4).

4.3.3 Uncoded instruction format

Instruction coding is used in almost all current processors to reduce the size of program code. This does not come for free, because usually the processor must decode instructions in a separate pipeline stage before they can be executed.

This stage can be eliminated by using an *uncoded instruction format*, in which instructions are left completely uncoded: The operation code of an instruction is fetched directly into the operation register and separate bitfields in the code control directly the operation of different parts of the processor.

The negative consequence of using uncoded instruction format is that according to our tests the width of instruction word increases by a constant factor of two to ten.

4.3.4 Fast branching

Equation 4.2 reveals that the main reason for control delays is the long latency of branch instructions.

Fast branching is a technique in which general forwarding is applied also to the operation of sequencer: The result bit of the compare unit controls forwarding mechanism, which selects the next instruction memory fetch address from the next instruction address calculated by sequencer adder and absolute target address in operation code. All this happens before the next instruction fetch is started.

Fast branching eliminates branch delays presuming the operation is executed in stage two. Otherwise one or more delay slots are required between compare instructions and branch instructions (see Figure 4.5). In comparison to branch prediction fast branching offers better possibilities for code scheduling and speculative execution, because branches have no delay.

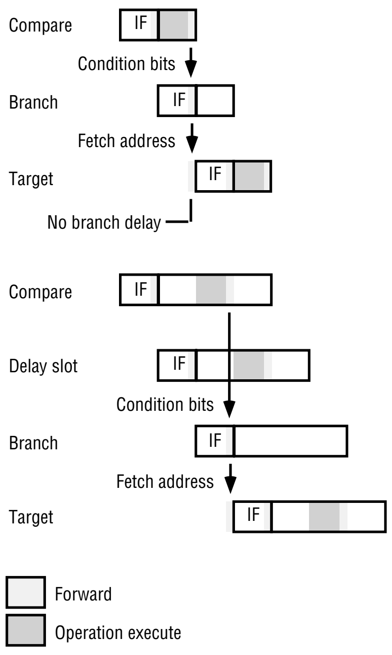


Figure 4.5 Fast branching requires the operation is executed in stage two otherwise delay slots are added.

The use of fast branching has also two disadvantages: The program loader must calculate branch target addresses while loading programs into the memory and the use of absolute addressing increases the size of the program slightly.

4.4 Architecture

In order to outline an abstract architecture with two level pipelining and fast branching, we apply the four described implementation techniques to the basic VLIW core [Fisher83]. We call the obtained architecture as *Minimal Pipeline Architecture* (MPA) [Forsell94].

The main parts of a MPA processor are *a* arithmetic and logic units (ALU_0-ALU_{a-1}), arithmetic unit for condition codes (AIC), *m* memory units (M_0-M_{m-1}), *r* registers (R_0-R_{r-1}) and sequencer (S). Parts are connected to each other by a large cross-bar switch, which acts as general forwarding mechanism (see Figure 4.6).

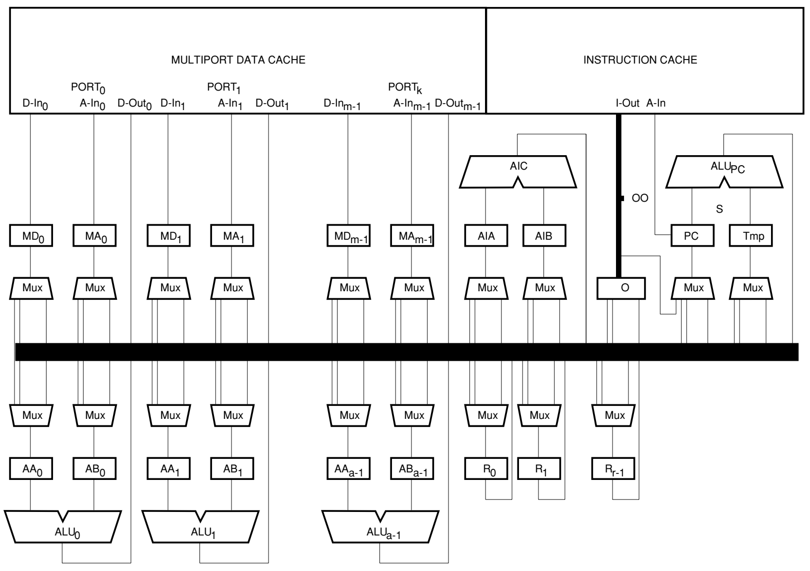


Figure 4.6 A block diagram of a MPA processor.

MPA uses VLIW-style instruction format, where single instruction consists of two word operands and several subinstructions that are assigned to appropriate functional units. The types of subinstructions are arithmetic and logical subinstructions (A), compare subinstructions (AIC), load and store subinstructions (M), write back subinstructions (R), and control subinstructions (S) (see Figure 4.7). In order to avoid separate decoding stage in MPA pipeline, instructions are uncoded and operands are always directly forwarded to appropriate functional units.

The instruction set of MPA is derived from DLX [Hennessy90] such that MPA subinstructions use only one functional unit. DLX instructions involving sequential use of many functional units are divided into separate MPA subinstructions.

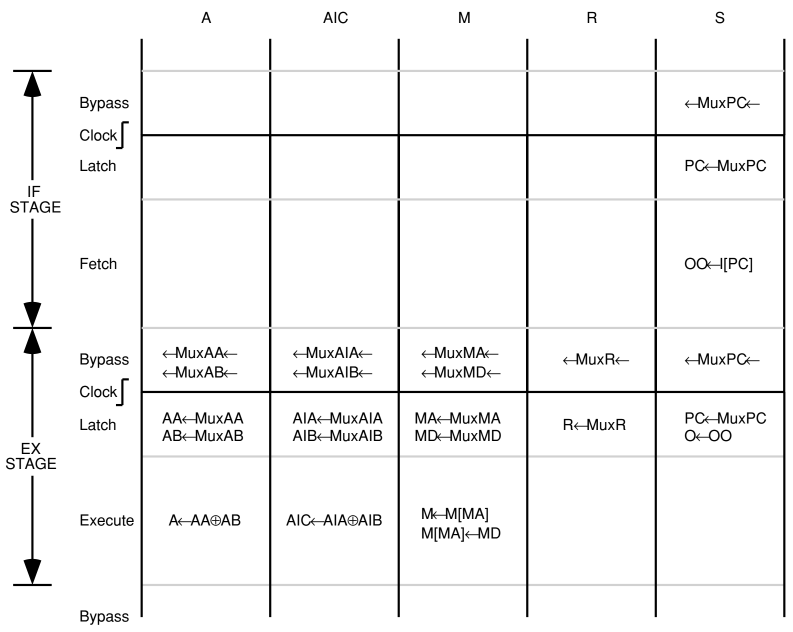


Figure 4.7 The operations for different types of subinstructions of M5 processor. The indexes are not shown.

MPA processor has a single addressing mode called *forwarded absolute addressing*. The address of a memory reference is always an absolute value, which is selected through the cross-bar from ALU operation result, memory unit result, data word operands or register contents. The more complicated addressing modes must be built from several sequential instructions: The address must first be calculated using ALU and then the result must be forwarded to memory unit.

4.4.1 MPA Pipeline

The pipeline of MPA consists of two stages: Instruction Fetch (IF) and Execute (EX). Both IF and EX stages are divided into three phases: Bypass Phase (BP), Latch Delay Phase (LD) and Execution Phase (EX) (see Figure 4.7).

The BP and LD phases of the IF stage are controlled by the previous instruction, which will select the correct value for PC according to operation code and condition codes. During the EX phase of the IF stage the processor fetches an operation code of an instruction from the instruction cache. This value on the instruction bus (OO) is used to control the beginning of instruction execution before the operation code is actually written into operation register (O) at the LD phase of the EX stage.

During the BP phase of the EX stage the operands of an instruction are selected and forwarded to operand registers of appropriate functional units. During the LD phase of the EX stage the values provided by the forwarding mechanism are written into operand registers of the functional units. The write back and control transfer operations are completed with this phase.

During the EX phase of the EX stage the functional units except the register units and sequencer commit their operations: The ALU executes an ALU subinstruction, the AIC unit executes a compare subinstruction, the memory unit executes a memory reference subinstruction. By the end of execution phase the results of the executed operations are ready for the bypassing phase of the next instruction.

4.4.2 Multicycle operations and exceptions

Multicycle operations can be implemented by two alternative techniques in MPA—dividing long operations into one cycle parts or allowing multicycle pipelines. Dividing long operations into one cycle parts unfortunately increases the complexity of the processor, because separate communication lines are needed. Allowing multicycle operations saves wiring space, but introduces unavoidable data dependencies between long instructions.

Exceptions can be implemented by saving both general and temporary register values also to exception registers during the LD phase of the EX stage and by selecting the given exception service address into PC in the case of exception. After the exception service is completed execution can be continued from the instruction which was interrupted by selecting the exception registers instead of actual registers during the BP phase of the IF stage. This technique allows also overlapping exceptions by the use of multiple sets of exception registers.

4.5 Evaluation

We evaluated the performance, code size, utilization of functional units and complexity of four processors—DLX, superDLX, M5 and M11 (see Figure 4.8). The first two were included for comparison purposes as representatives of basic pipelined and superscalar processors.

Processor	**DLX**	**sDLX**	**M5**	**M11**
Operation code length	32	320	320	512
Instruction scheduling	pipelined	superscalar	VLIW	VLIW
Pipeline stages	5	5	2	2
Functional units	4	10	4	10
Arithmetic and logic units	1	2+2+2	1	4
Compare units	1	1	1	1
Memory units	1	4	1	4
Sequencers	1	1	1	1

Figure 4.8 Evaluated processors.

DLX is an experimental load/store RISC architecture featuring basic five level pipeline [Hennessy90]. SuperDLX [Moura93] is an experimental superscalar processor derived from DLX architecture featuring out of order execution and out of order completion of instructions and branch prediction. M5 and M11 are instances of MPA containing four and ten functional units respectively [Forsell94].

The number are chosen such that DLX has approximately the same amount of processing resources as M5, and superDLX has the same amount of resources as M11.

4.5.1 Simulations

The evaluation was made by simulating the execution of five hand compiled toy benchmarks in the processors:

block	A program that moves a block of words to the other location in memory
fib	A program that calculates the fibonacci value
sieve	A program that calculates prime numbers using the sieve of Eratosthenes
sort	A program that sorts a table of words using recursive quicksort algorithm
trans	A program that returns the transpose of a matrix of words

Toy benchmarks were chosen, because we wanted to use hand compilation to make sure we were measuring the performance of architectures not compilers. The benchmarks were run assuming ideal memory hierarchy, i.e., there are no cache misses or the cycle time of main memory is smaller than the cycle time of the processor.

We measured the execution time in clock cycles, program code size in bytes and utilization of functional units. The results of simulations are shown in Figure 4.9. According to them M5 runs benchmarks 266% faster than DLX and approximately as fast as superDLX. The speedups range from 2.0 to 3.7. M11 runs benchmarks 587% faster than DLX and 222% faster than superDLX. The speedups range from 3.1 to 12.

The size measurements of the simulated benchmark programs reveal that M5 code is 5.3 times longer than DLX code and M11 code is 6.9 times longer than DLX code.

The average utilization of functional units in M5 is 67.1% that is 2.4 times better than in DLX. The average utilization of functional units in M11 is 70.0% which is 2.3 times better than in superDLX. The high functional unit utilizations of MPA are partially due to certain hand optimization techniques that decrease execution time by adding subinstructions to suitable locations in code.

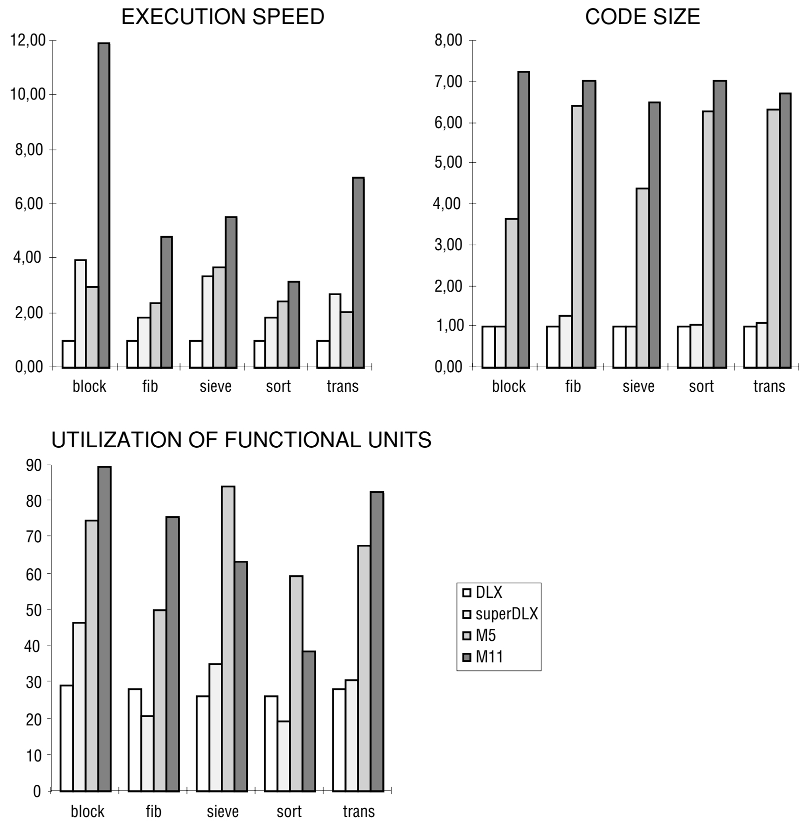
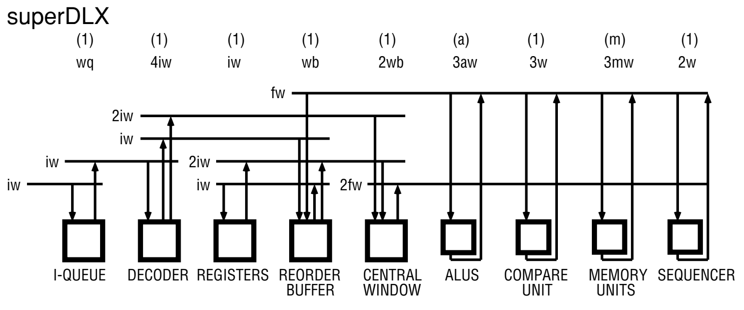


Figure 4.9 The relative performance, size of programs and utilization of functional units.

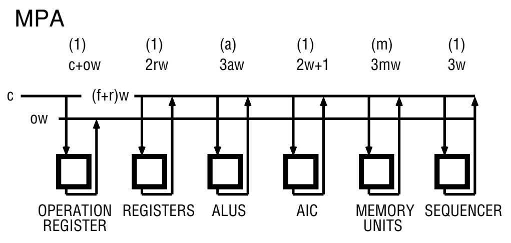
4.5.2 Complexity and clock cycle length

Let us compare the wiring area needed for a w-bit i-issue superDLX and w-bit MPA processor both having r registers and f functional units, i.e., a ALUs, one compare unit, m memory units and one sequencer. Assume that the superDLX processor has q-slot instruction queue, b-slot reorder buffer and b-slot central window. Assume also that the number of immediate operands is o and the length of instruction word is c in the MPA processor.

To simplify the comparison let us locate all elements of a processor in one dimension and assume that all elements are equally wide (see Figure 4.10). By calculating the areas needed for communication lines we get that 4-issue superDLX occupies 75% of the area needed for M5 and 10-issue superDLX occupies 106% of area needed for M11 (see Figure 4.10). A lot of additional die area is needed for an instruction queue, decoder, reorder buffer, central window, memory buffer and branch target buffer in superDLX.



a	b	f	i	m	q	w	AREA	DELAY
1	20	4	4	1	17	32	4.86K	7
4	50	10	10	4	47	32	17.9K	13



a	c	f	m	o	r	w	AREA	DELAY
1	320	4	1	2	32	32	6.46K	7
4	512	10	4	4	32	32	16.8K	13

Figure 4.10 The die area and delay calculations of communication lines. Terms like "4iw" indicate the number of lines. The width of elements are shown in parenthesis.

Due to similar functional unit construction in superDLX and in MPA only the forwarding mechanism delay may affect the clock cycle in MPA. The length of forwarding paths are the width of functional units plus three including the width of reorder buffer and central window in the case of superDLX and the width of register unit and operand register in the case of MPA (see Figure 4.10). We get that the forwarding path of MPA and superDLX are equally long and thus the clock cycles are potentially equally long.

4.6 Conclusions

We have described techniques to eliminate pipeline delays without losing the performance advantage of pipelining. We applied these techniques to a basic VLIW architecture. As a result we outlined Minimal Pipeline Architecture (MPA) using cross-bar interconnect of functional units and pipeline that has only two stages.

We evaluated two MPA processors and two reference processors, DLX and superDLX, by simulations. The results are quite favorable to MPA:

M5, a five unit implementation of MPA, performs over two and a half times better than DLX, a RISC processor, that has same number of functional units than M5. The better performance comes from better exploitation of available instruction level parallelism, which is due to over two times better utilization of functional units and absence of pipeline hazards. Unfortunately the code size expands by the factor of five.

M11, an eleven unit implementation of MPA, performs almost six times better than DLX and over two times better than superDLX, a superscalar processor featuring out of order execution and branch prediction, and the same number of functional units than M11. Furthermore, the complexity of cross-bar based M11 does not exceed the complexity of superDLX. Also in this case the code size expands by the factor of almost seven.

The difference in performance is quite surprising, because in theory superscalar and VLIW processors should perform equally well and one would have assumed that the complexity of superscalar processor were lower. We believe that the difference in performance is an evidence that the static compile time scheduling in a VLIW processor performs better than the dynamic run-time scheduling in a superscalar processor. We believe also that the high complexity of a basic superscalar processor indicates that the development of superscalar architectures has reached the point where alternative approaches like MPA are possible.

The increased code size of MPA unfortunately implies larger instruction caches and more expensive chips or more instruction cache misses and slightly decreased performance. Alternatively one may use decoded instruction cache (DINC) [Dizel87] between the main memory and processor to return the code density to the level of DLX. This would, however, require predecoding and fetch address prediction quite like in superscalar processors.

The code size reduction techniques, instruction cache analysis and feasibility issues of MPA would be interesting topics for further investigation.

References

[Chang91]
P. Chang, S. Mahlke, W. Chen, N. Warter and W. Hwu, IMPACT: An architectural Framework for Multiple-Instruction-Issue Processors, *Proceedings of the 18th Annual International Symposium on Computer Architecture*, Association for Computing Machinery, New York, 1991, 266-275.

[Dizel87]
D. Dizel, The Hardware Architecture of the CRISP Microprocessors, *Proceedings of the 14th Annual International Symposium on Computer Architecture*, Computer Society Press of the IEEE, Washington, 1987, 309-319.

[Fisher81]
J. Fisher, Trace Scheduling: A technique for global microcode compaction, *IEEE Transactions on Computers* **C-30**, (1981) 478-490.

[Fisher83]
J. Fisher, Very Long Instruction Word Architectures and ELI-512, *Proceedings of the 10th Annual International Symposium on Computer Architecture*, Computer Society Press of the IEEE, Washington, 1983, 140-150.

[Flynn95]
M. Flynn, *Computer Architecture—Pipelined and Parallel Processor Design*, Jones and Bartlett Publishers, Boston, 1995.

[Forsell94]
M. Forsell, Design and Analysis of Some Chip-level Parallel Architectures, *Licentiate thesis*, Department of Computer Science, University of Joensuu, Joensuu, 1994.

[Hennessy90]
J. L. Hennessy, D. A. Patterson, *Computer Architecture: A Quantitative Approach*, Morgan Kaufmann Publishers Inc., San Mateo, 1990.

[Johnson89]
W. M. Johnson, Super-Scalar Processor Design, *Technical Report No. CSL-TR-89-383*, Stanford University, Stanford, 1989.

[Kogge81]
P. M. Kogge, *The Architecture of Pipelined Computers*, Hemisphere Publishing Corporation, Washington, 1981.

[Moura93]
C. Moura, SuperDLX - A Generic Superscalar Simulator, *ACAPS Technical Memo 64*, McGill University, Montreal, 1993.

[Nicolau84]
A. Nicolau and J. Fisher, Measuring the parallelism available for very long instruction word architectures, *IEEE Transactions on Computers* **C-33**, 11 (1984), 968-976.

[Smith84]
A. Smith and J. Lee, Branch Prediction Strategies and Branch Target Buffer Design, *Computer* **17**, 1 (1984), 6-22.

[Tomasulo67]
R. M. Tomasulo, An efficient algorithm for exploiting multiple arithmetic units, *IBM Journal of Research and Development* **11**, 1 (1967), 25-33.

Chapter 5

MTAC—A Multithreaded VLIW Architecture for PRAM Simulation

Martti J. Forsell

Department of Computer Science, University of Joensuu, PB 111, FIN-80101 Joensuu, Finland

Abstract

The high latency of memory operations is a problem in both sequential and parallel computing. Multithreading is a technique, which can be used to eliminate the delays caused by the high latency. This happens by letting a processor to execute other processes (threads) while one process is waiting for the completion of a memory operation. In this paper we investigate the implementation of multithreading in the processor-level. As a result we outline and evaluate a MultiThreaded VLIW processor Architecture with functional unit Chaining (MTAC), which is specially designed for PRAM-style parallelism. According to our experiments MTAC offers remarkably better performance than a basic pipelined RISC architecture and chaining improves the exploitation of instruction level parallelism to a level where the achieved speedup corresponds to the number of functional units in a processor.

Keywords: PRAM, VLIW, multithreading, chaining

5.1 Introduction

Efficient parallel computers are hard to manufacture due to yet unsolved theoretical and technical problems. The most important is: how to arrange efficient communication between processors?

A theoretically elegant solution is to arrange communication by building a shared memory between processors [Schwarz66, Karp69, Fortune78]. In the 50's, 60's and 70's , however, the idea of building true shared memories was considered impossible. Instead several other possibilities, like memory interleaving, and memory distribution, were investigated [Burnett70, Enslow74, Schwarz 80.

Despite of their poor feasibility, the use of shared memory models continued among algorithm designers due to the simplicity of programming. The most widely used model is parallel random access machine (PRAM) [Fortune78, Leighton91, McColl92] (see Section 5.2 for a definition).

While building of shared memory has remained unfeasible, there has been a considerable effort to simulate an ideal shared memory machine like a PRAM with a physically distributed memory machine, which consists of processors and memory modules connected to each others through a communication network [Schwartz80, Gottlieb83, Gajski83, Hillis 85, Pfister85, Ranade87, Abolhassan93, Forsell96a].

Efficient processor architectures are not as hard to design as entire parallel computers, but there are still two major problems that reduce the performance of them: Typical thread-level parallel processor architectures suffer from low utilization of processors, because a large portion of time is wasted in waiting for memory requests to complete, and typical instruction-level parallel architectures suffer from delays caused by instruction dependencies, because typical sequential code contains a lot of dependencies.

In this paper we try to find a common solution to both of these problems by using multithreading to increase the utilization of processors and functional unit chaining to eliminate dependencies. With this in mind, we apply multithreading and chaining along with extensive superpipelining to a basic very long instruction word (VLIW) [Fisher83, Nicolau84] (see Section 5.3 for a definition) architecture [Forsell96b]. As a result we outline a MultiThreaded VLIW processor Architecture with functional unit Chaining (MTAC), which specially suits for efficient simulation of a PRAM.

MTAC features a very short clock cycle, chained operations and absence of pipeline and memory system hazards. A preliminary performance evaluation of MTAC as a processor architecture is given.

According to our experiments MTAC offers remarkably better performance than a basic pipelined RISC architecture with the same number of functional units and chaining improves the efficiency of instruction level parallelism to a level where the achieved speedup corresponds to the number of functional units in processors.

5.2 The idea of simulation

In this section we outline briefly how an *ideal shared memory machine* (SMM), like a PRAM, can be simulated by a physically distributed memory machine.

5.2.1 Parallel random access machine

The *Parallel Random Access Machine* (PRAM) model is a logical extension of the Random Access Machine (RAM) model. It is a widely used abstract model of parallel computation [Fortune78, Leighton91, McColl92]. A PRAM consists of p processors each having a similar register architecture. All processors are connected to a shared memory (see Figure 5.1). All processors run the same program synchronously, but a processor may branch within the program independently of other processors. That is, each processor has its own Program Counter. There are no memory access restrictions: If all p processors initiate a memory reference simultaneously, the memory system will complete all references in a single clock cycle.

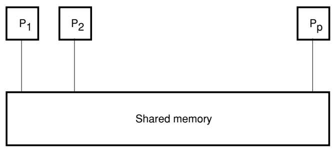


Figure 5.1 A Parallel Random Access Machine with processors $P_1,\ldots,P_p$.

The ideal shared memory of large PRAMs is very difficult to build directly due to extreme complexity and very high costs: According to our earlier investigations multiport memories are feasible, but the silicon area and cost of a p-port shared memory are p^2 times greater than the required silicon area and cost of an ordinary single port memory [Forsell94]. It is hard to imagine that multiport memories could be implemented by another structure that is simpler than that proposed in [Forsell94].

5.2.2 Distributed memory machine

A *physically distributed memory machine* (DMM) is a computer, which consists of p processors and m memory modules connected to each others through a communication network (see Figure 5.2). Processors use message passing to access memory locations.

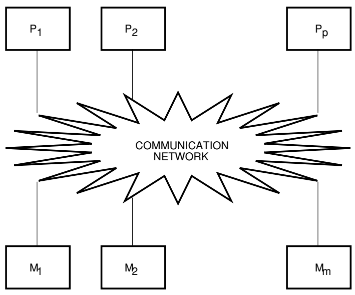


Figure 5.2 A physically distributed memory machine, where processors $P_1,\ldots,P_p$ are connected to memory modules $M_1,\ldots,M_m$ through a communication network.

Most current parallel computers are DMMs [Hillis85, Prechelt93], because a DMM is a lot easier to build than a SMM [Forsell94].

Note that the term shared memory is currently widely used in the computer industry in a different meaning: A machine has a shared memory if all processors have access to a single memory, even if access is serialized. We consider this misleading because this terminology classifies both a parallel computer using the PRAM model and a parallel computer with a time-shared bus between processors and the memory to the same shared memory class, although the PRAM computer has considerably higher performance in most applications.

5.2.3 Simulating SMM on DMM

According to earlier investigations, DMMs can be used to simulate SMMs like PRAMs [Schwartz80, Gottlieb83, Gajski83, Hillis85, Pfister85, Ranade87, Abolhassan93, Forsell96a].

The standard solution uses two special techniques—random hashing and processor overloading.

Random hashing is used to distribute memory locations of a SMM to the modules of a DMM: A memory mapping function is randomly picked up from a family of memory mapping functions, which distribute memory locations evenly over the memory modules. If the function does not work well with a particular application the function is changed to another. The main purpose of hashing is to prevent contention of messages in the communication network.

Processor overloading means simulating a number of virtual processors by a single physical processor. Processor overloading is used to hide the latency of long operations, which slow down the execution of parallel programs: While, e.g., a memory request initiated by an instruction of a virtual processor is being processed in a network, the processor executes instructions of the other virtual processors. This is possible because the instructions of virtual processors are independent of each others within a cycle of a physical processor.

The current RISC and superscalar processors with large register files and long reorder buffers and pipelines [Johnson89, Hennessy90] are not suitable for processor overloading, because they lack mechanisms for fast process switches, and the number of registers is still far too small for efficient data prefetching. The objective of this paper is to outline a processor architecture that is capable for processor overloading and extracting enough instruction level parallelism.

5.3 Multithreading, distribution and chaining

In this section we describe four main techniques—multithreading, register file distribution, VLIW scheduling and functional unit chaining—that are necessary for an efficient PRAM processor architecture.

5.3.1 Multithreading

A processor is *multithreaded* if it is able to execute multiple processes (threads) of a single program in an overlapped manner [Moore96]. The threads in a multithreaded processor are executed in fixed order or scheduled by a scheduling mechanism utilizing, e.g., a priority queue. The use of a complex scheduling mechanism usually requires that more than one instruction of a thread must be executed before next thread can be changed to execution. This is caused by the time taken by the scheduling decision [Moore96].

5.3.2 Register file distribution

An efficient realization of multithreading requires simultaneous access to a large number of registers, because the execution of multiple threads is overlapped in a multithreaded processor. This causes problems, because large multiport register files are slow and very difficult to build.

Quick simultaneous access to all registers can be implemented by a novel register file organization called *distributed register file*, in which single registers are treated as separate functional units that are connected to each other by a cross-bar like selection network where needed (see Figure 5.4) [Forsell96b].

5.3.3 VLIW scheduling

A *Very Long Instruction Word* (VLIW) processor is a processor, which executes instructions consisting of the fixed number of smaller subinstructions [Fisher83, Nicolau84]. Subinstructions are executed in multiple functional units in parallel. Subinstructions filling actual instructions are determined under compile time.

In order to achieve high speedups the programs for VLIW processors must be compiled using advanced compilation techniques breaking the basic block structure of the program [Fisher81, Chang91].

We selected the VLIW scheduling for MTAC, because the main alternative—the dynamic runtime scheduling with out of order execution and branch prediction used in superscalar processors [Johnson89, Hennessy90]—is not suitable for strictly synchronous PRAM simulation. Furthermore, there is evidence that a VLIW processor with the same number of functional units is faster than a superscalar processor with out of order execution and dynamic branch prediction, while the VLIW solution requires no more silicon area [Forsell96b].

5.3.4 Functional unit chaining

In a processor consisting of multiple functional units [Hennessy90] one can chain the units so that a unit is able to use the results of its predecessors in the chain. We call this technique *functional unit chaining* (see Figure 5.3).

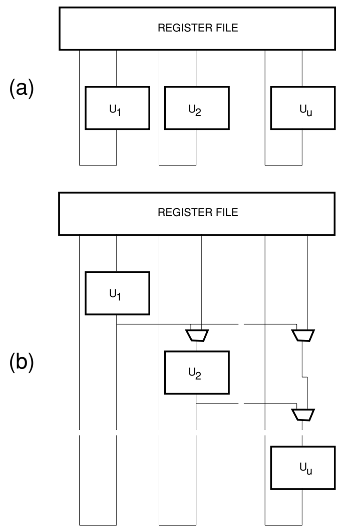


Figure 5.3 A set of functional units that operate (a) in parallel and (b) in chain. Symbols with two inputs and a single output are multiplexers.

5.3.5 Efficient parallel multithreading with chaining

Functional unit chaining allows for execution of a code fraction containing dependencies within a single clock cycle of a thread. Unfortunately, this significantly increases the length of the clock cycle.

Chaining can, however, be combined with superpipelining [Jouppi89] to retain short clock cycles. This solution generates delays due to dependencies in a sequential program code, but in the case of parallel programs, threads are independent within a cycle of a physical processor by definition.

5.4 Multithreaded architecture with chaining

In order to outline an abstract multithreaded processor architecture for a simulation of a PRAM we apply functional unit chaining, fixed order multithreading, and superpipelining to the basic VLIW core with a distributed register file [Forsell96b]. We call the obtained architecture the *MultiThreaded Architecture with Chaining* (MTAC).

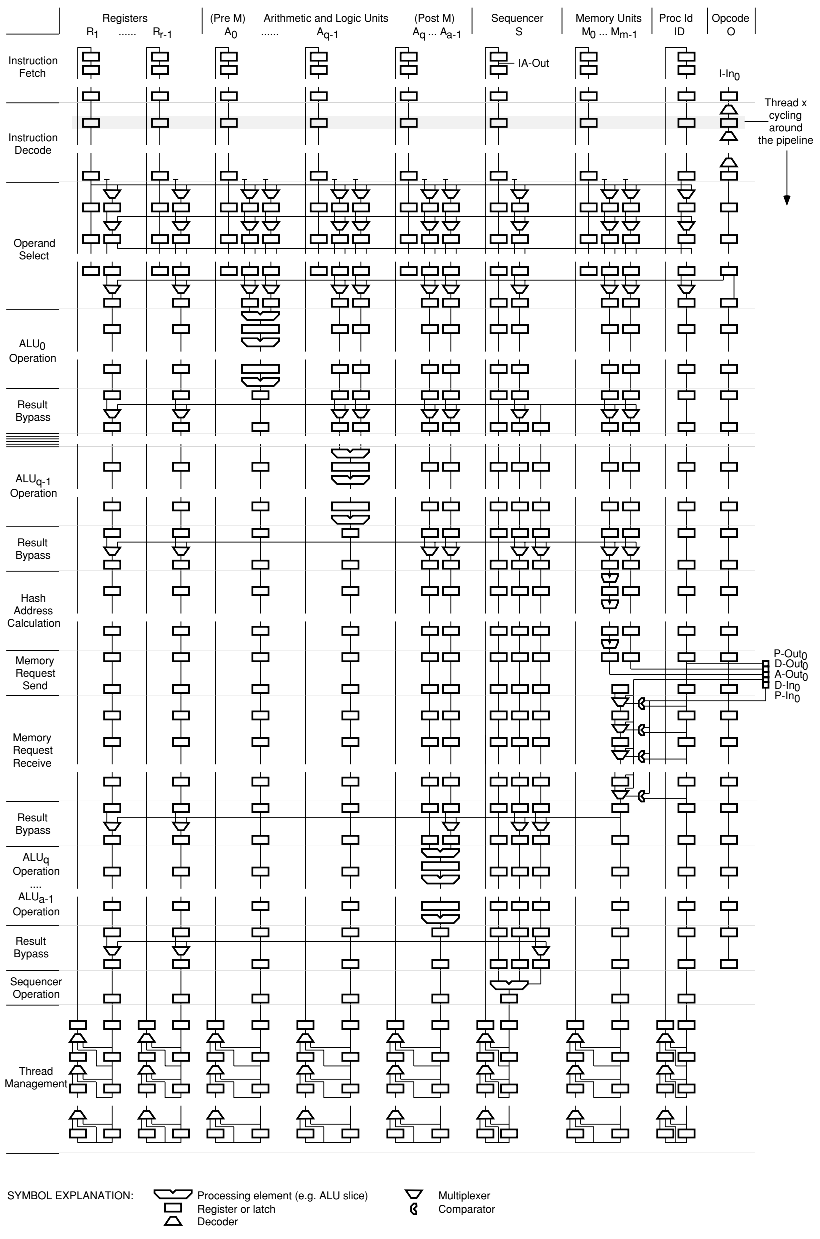


Figure 5.4 A detailed block diagram of a MTAC processor.

The main features of MTAC are

- Multithreaded operation with single instruction length threads and a zero-time switch between threads.
- Variable number of threads.
- Multiple functional units connected as a chain, so that a unit is able to use the results of its predecessors in the chain.
- A very short clock cycle due to a reqular structure without long lines that cannot be pipelined.
- VLIW-style instruction scheduling allowing for a complex coding of instructions if necessary.
- The construction of a compiler is easier than with average VLIW machines due to functional unit chaining allowing for the execution of dependent subinstructions within an instruction.
- No need for on-chip (or off-chip) data or instruction caches or branch prediction. Multithreading is used to hide memory, communication network and branch latencies.
- Completely queued operations — no memory reference message handling overhead.

The main parts of a MTAC processor are *a* arithmetic and logic units (A_0-A_{a-1}), *m* memory units (M_0-M_{m-1}), operation register (O), process identification register (ID), *r* general registers (R_0-R_{r-1}) and sequencer (S). Each part consists of a set of registers and processing elements connected to each others in a chain-like manner. Some subparts of chains are also connected to each others (see Figure 5.4).

A horizontal line of latches store contents of a single thread (see Figure 5.4). From this point of view a MTAC processor consists of $t_{\max}$ slices, which store and process threads. Any of the lowermost $t_{\max}$-$t_{\min}$+1 slices can be connected to the topmost slice so that the pipeline of MTAC can store simultaneously $t_{\min}$ to $t_{\max}$ threads. In every clock cycle threads are forwarded one step in the pipeline. Thus, the processor contains a variable number of threads which are cycling around the pipeline.

5.4.1 MTAC pipeline

The pipeline of a MTAC consists of *i* instruction fetch stages (IF_0-IF_{i-1}), *d* decode stages (DE_0-DE_{d-1}), *o* operand select stages (OS_0-OS_{o-1}), *e* execute stages (EX_0-EX_{e-1}), *h* hash calculate stages (HA_0-HA_{h-1}), *u* memory request stages (ME_0-ME_{u-1}), *b* result bypass stages (BB_0-BB_{b-1}), one sequencer stage (SE) and 1 to $t_{\max}$-$t_{\min}$+1 thread management stages (TM$_0$-TM$_{t_{\max}-t_{\min}+1}$). Symbols *i*, *d*, *o*, *e*, *h*, *u*, *b*, $t_{\min}$, and $t_{\max}$ are implementation dependent constants.

The cycle of a thread begins with the fetch of a very long operation code word from the local memory (IF stages). The latency of the memory is embedded into *i* stages. During the DE stages the operation code of the instruction is decoded. Due to very short clock cycle *d* stages are needed for decoding. OS stages take care of the operand selection and the transfer of operands to operand registers of appropriate functional units. To transfer all operands *o* stages are needed.

During the EX stages ALUs commit their operations. Due to the varying order of instructions in a typical basic block, the first *q* ALUs (A$_0$-A$_{q-1}$) are placed before memory units and the rest *a*-*q* ALUs (A$_q$-A$_{a-1}$) are placed after memory units. Totally *e* stages are needed for all execute operations.

In the middle of the EX stages physical addresses of memory references are calculated (HA stages) and all reference messages are transmitted simultaneously to the communication network (ME stages). Before the end of ME stages a thread receives the possible reply messages of its memory requests. If a thread issuing a memory read request reaches the last ME stage without receiving the reply message, the whole processor is halted until the message arrives.

The processing of a thread ends with the selection of the next PC address according to the results of compare operations (SE stage). The last 1 to $t_{\max}$-$t_{\min}$+1 stages are used for transferring the contents of threads to the beginning of the chain and allowing for the number of threads to vary from $t_{\min}$ to $t_{\max}$.

5.4.2 Other aspects

We selected not to chain memory units, because sequential memory requests within an instruction cycle of a thread in multithreaded processor cannot be done without losing memory consistency or performing full synchronization between the requests.

The order of the functional units is chosen according to typical instruction order in a basic block of code: Memory units are placed in the middle of ALUs. The sequencer is the final unit in the chain. We added also a special mechanism to benefit more from chaining: An ALU or sequencer may use the result of a compare operation committed in one of the previous ALUs to select one of its operands.

Synchronization between processors and threads is be done by a separate synchronization network between processors: A processor is equipped with a mechanism, which freezes a thread after it has executed a synch instruction by blocking its memory references and PC changes until the processor receives a synch message from a separate synchronization network.

PRAM model assumes unrealisticly that the number of processors p can be as high as an algorithm requires, whereas the number of threads is limited to tmax in a MTAC processor. This is not, however, a difficult problem, because any task T_Q consisting of Q operations and taking U time steps with enough processors can be executed with t processors in time U+(Q-U)/t [Brent74]. On the other hand, if the number of threads is set to $T < t_{\min}$, the processor creates $t_{\min}$-T null threads, so that the processor remains functional. Alternatively, we can quite easily add a support for fast context switches to MTAC, if more than $t_{\max}$ simultaneous threads are absolutely required: Add w switch stages (SW) in the beginning of the pipeline of MTAC. During these stages a thread sends its contents to the local memory and receives the contents of a stored thread from the local memory.

5.5 Performance evaluation

We evaluated the performance, the static size of code and the utilization of functional units for five processors—DLX, T5, T7, T11, T19 (see Table 5.1).

UNITS	DLX	T5	T7	T11	T19
Functional Units	4	4	6	10	18
- Arithmetic and Logic Unit (ALU)	1	1	3	6	12
- Compare Unit (CMP)	1	1	1	1	1
- Memory Unit (MU)	1	1	1	2	4
- Sequencer (SEQ)	1	1	1	1	1
Register Unit	1	1	1	1	1

Table 5.1 Evaluated processors.

DLX is an experimental load/store RISC architecture featuring basic five level pipeline [Hennessy90]. It is included for comparison purposes as a representative of basic pipelined processor. DLX provides a good reference point for an architectural comparison, although there are faster sequential processors available.

T5, T7, T11 and T19 are instances of MTAC containing four, six, ten and eighteen functional units plus one register unit, respectively. The numbers are chosen so that DLX and T5 have the same number of functional units. The final ALU was changed to a compare unit (CMP) in MTAC processors for comparison purposes, because DLX has a dedicated hardware (CMP) for calculating and solving branch target addresses.

To measure the exploited instruction level parallelism we included also T7, T11 and T19. T11 has twice the processing resources of T7 and T19 has twice the processing resources of T11. T5 was not used as a base machine for instruction level parallelism measurements, because it does have enough arithmetic power for proper comparisons.

We also evaluated the performance of six machines—D-1, D-16, D-16s, T5-1, T5-16 and T19-16 (see Table 5.2).

PROPERTIES	D-1	D-16	D-16s	T5-1	T5-16	T19-16
Type of processor	DLX	DLX	DLX	T5	T5	T19
Number of processors	1	16	16	1	16	16
Number of threads	1	512	1	512	512	512
Clock frequency (MHz)	300	300	300	600	600	600
Thread swithing time (clk)	-	70	-	0	0	0
Latency of a network (clk)	-	6	6	-	6	12
Memory module technology	IL	B	B	B	B	B
Interleaving factor	4	-	-	-	-	-
Number of memory modules	1	16	16	1	16	64
Number of banks in a module	-	32	32	64	64	64
Assumed number of bank collisions	-	3	3	3	3	3
Cycle time of a memory module (clk)	27	27	27	54	54	54
Access time of a memory module (clk)	15	15	15	30	30	30
Access time of a level 1 cache (clk)	1	-	-	-	-	-
Line length of a level 1 cache (word)	4	-	-	-	-	-
Cache miss time (clk)	16	-	-	-	-	-
Next cache line word available (clk)	4	-	-	-	-	-

Table 5.2 Evaluated machines. The numbers are assumptions based on available technology (see the discussion in Section 5.6). (IL=Interleaved, B=Banked).

D-1 is a sequential machine with a single DLX processor, a single cycle four-word line level 1 cache, and a 4-way interleaved DRAM memory system. In a case of a cache miss, we assume that, the cache line is filled so that first word is available after 16 clock cycles and the remaining are available after 4 cycle delay each. D-1 is included for comparison purposes as a representative of conventional sequential computer. D-1 provides a good reference point for a system level comparison, although there are faster sequential computers available.

D-16 and D-16s are parallel machines using sixteen DLX processors connected to each others with a communication network. D-16 uses 512 threads per processor in order to the hide message latency. D-16s uses a single thread per processor. T5-1 is a sequential machine with a single 512-thread T5 processor. T5-16, T19-16 are parallel machines using sixteen 512-thread T5 or T19 processors connected to each others with a communication network, respectively. The memory system of all parallel machines consists of 16 to 64 memory modules. A memory module is divided into 32 or 64 banks with access queues due to slow cycle time of a memory module.

5.5.1 Simulation methods

We made two kinds of experiments—processor level simulations and machine level estimations.

Processor level simulations were made by simulating a single thread of execution of seven hand compiled toy benchmarks (see Table 5.3) in the processors.

PROGRAM	NOTES
add	A program that calculates the sum of two matrices
block	A program that moves a block of words to other location in memory
fib	A non-recursive program that calculates a fibonacci value
max	A program that finds the maximum value of matrix of integers
pre	A program that calculates the presum of a matrix of words
sort	A program that sorts a table of words using recursive mergesort algorithm
trans	A program that returns the transpose of a matrix of words

Table 5.3 The benchmark programs.

Single thread simulations are based on the fact that in the most parallel applications the size of the problem, *N*, is much bigger than the number of the processors, *P*, in a parallel machine. Now, a typical way to spread the problem of size *N* to *P* processors is to divide the problem into *P* subproblems of size *N/P*. Then these *P* subproblems are solved with *P* processors using sequential algorithms. Finally, if necessary, the results of these *P* subproblems are combined with *P* processors. Usually the sequential part dominates the execution time. Now, it is possible to extract a single thread of execution from the sequencial part, and that is how these benchmarks were created. The purpose of these processor level simulations was to figure out the architectural characteristics of MTAC without the effect of a memory system.

The benchmarks, except fib, are frequently used primitives in parallel programs. They were chosen for simplicity, because we wanted to use hand compilation to make sure we were measuring the performance of architectures, not compilers. The benchmarks were run assuming processors have an ideal distributed memory system, i.e., every memory request is completed before the result is used. We assumed also that all processors are able to run at the same clock frequency and DLX is able to run multithreaded code perfectly.

Machine level estimations were made by simulating the execution of full versions of two hand compiled toy benchmarks (add and max) (see Table 5.3) in the machines and taking into account the effect of the assumed memory system. Due to randomized hashing of memory locations it is possible that two memory reference messages collide, i.e., they are trying to enter into a single memory bank simultaneously. In the case of collision, one message have to wait in an access queue until another has completed its memory access. We assumed that the maximum number of collisions is three. The purpose of these machine level estimations was to figure out the more realistic performance of a MTAC system. Due to a speculative nature of these estimations we included only two benchmarks.

We used *dlxsim* (version 1.1) [Hennessy90] for DLX programs and the *MTACSim* (version 1.1.0) simulator [Forsell97] for MTAC programs. A code example for the max benchmark is shown in Table 5.4.

5.5.2 Results

In our processor level simulations we measured the execution time in clock cycles, the static program code size in instructions and the utilization of functional units for all processors. The normalized results are shown in Figure 5.5.

```
; An example code of max benchmark for T5-1 and T5-16 machines
;
; R1        Pointer to the table
; R2        The end address of the table
; R3        Memory(Ponter) contents
; R4        Max value
; R5        Thread interleave constant (threads * word/processor)
; R6        Thread number sifted for word/processor
; R7        Save address for combining
; R8        Combining pointer
; R9        Combining interleave stepping down
; R10       Constant 4

_MAIN

    SHL0   O0,O1  WB5    A0     OP0    THREAD        OP1    2
    SHL0   R30,O0 WB6    A0     OP0    2
    LD0    A0     ADD0   O0,R6  WB1    A0     WB4    M0     WB7    A0     OP0    _TABLE
    ADD0   R1,R5  WB1    A0     WB2    O0     WB10   O1     OP0    _ENDING       OP1    4

L0
    LD0    R1     WB3    M0     SLE    M0,R4  BNEZ   O1     OP1    L1
           ADD0   R1,R5  SLT    R1,R2  BNEZ   01,O0  WB1    A0     WB4    R3     OP0    L2     OP1    L0

L1  ADD0   R1,R5  SLT    A0,R2  BNEZ   01     WB1    A0     OP1    L0

L2  SHR0   R5,O0  WB9    A0     OP0    1
    ADD0   R7,R9  ST0    R4,R7  WB8    A0

L3
    SHR0   R9,O0  LD0    R8     WB3    M0     SLE    M0,R4  BNEZ   O1     WB9    A0     OP0    1      OP1    L4
           ADD0   R7,R9  ST0    R3,R7  SGE    R9,R10 BNEZ   O1,O0  WB4    R3     WB8    A0     OP0    L5     OP1    L3
L4  ADD0   R7,R9  SGE   R9,R10  BNEZ   O1     WB8    A0     OP1    L3

L5  TRAP   O0            OP0           0

_TABLE:
   .RANDOM               4096

_ENDING:
```

Table 5.4 An Example code of the max benchmark for T5-1 and T5-16 machines. Subinstructions written on a single line belong to the same instruction. See [Forsell97] for further information.

According to the tests T5 runs benchmarks 2.7 times faster than DLX. T7, T11 and T19 perform 4.1, 8.1 and 16.6 times faster than DLX.

The exploited instruction level parallelism in MTAC seemed to scale up exceptionally well in this case of small number of functional units: T11 was 1.97 times faster than T7 and T19 was 2.05 times faster than T11.

The static code size reduces almost as one might expect by execution time measurements. According to measurements the code size in instructions of T5, T7, T11 and T19 are 0.38, 0.28, 0.22 and 0.20 times the code size of DLX, respectively.

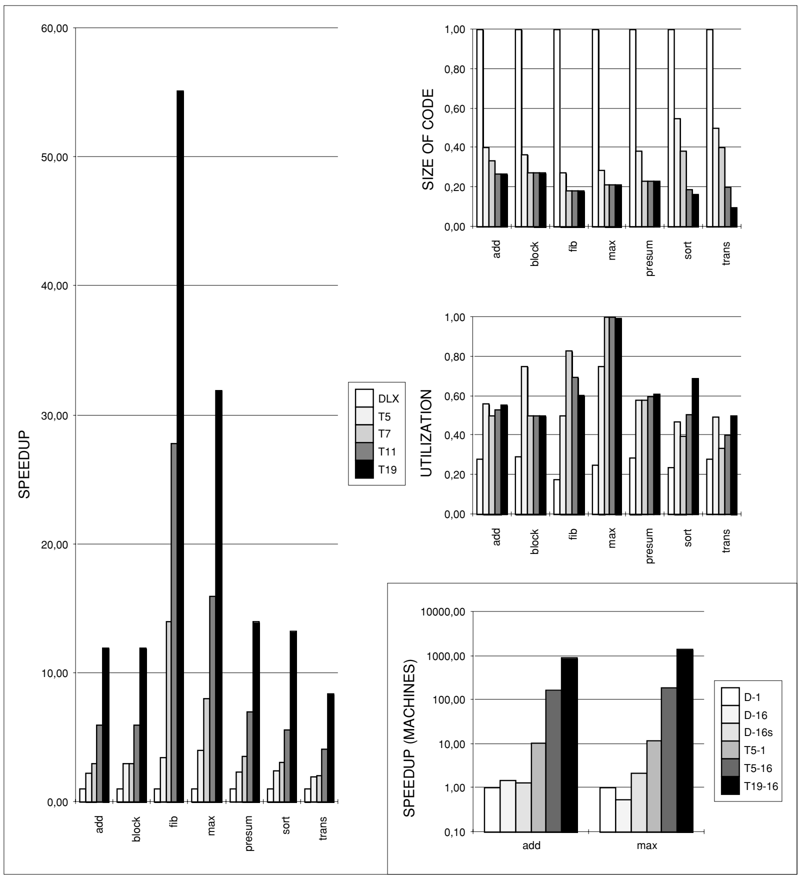


Figure 5.5 The results of experiments. The relative performances, the relative static code size and the utilization of functional units in the evaluated processors (upper left charts). The relative performances of the evaluated machines (the lower right chart).

The size of code would have been reduced more in the case of longer basic blocks, because in some of the benchmarks main blocks reduced into a single instruction that was not compressible.

The average utilization of functional units in DLX was only 25.8%. In the T5, T7, T11 and T19 average utilizations were 58.6%, 59.2%, 60.4% and 63.5%, respectively. Thus, the utilizations in MTAC processors were roughly twice than that in DLX.

In our machine level estimations we measured the execution time in milliseconds for all machines. The normalized results are shown in Figure 5.5.

According to the tests T5-1 runs add benchmark 10.5 times faster and max benchmark 12.0 times faster than D-1. T5-16 performs add 168, 109 and 126 times better than D-1, D-16 and D-16s. The numbers are 191, 354 and 88,5 for the max benchmark. T19-16 performs add 895, 583 and 671 times better than D-1, D-16 and D-16s. The numbers are 1460, 2700 and 675 for the max benchmark.

The performance of MTAC machines seemed to scale up exceptionally well in this case of small number of processors: T5-16 was 16.0 and 15.9 times faster than T5-1.

In the case of DLX systems, the execution time of benchmarks were capitalized by memory systems delays. The performance of these benchmarks would not be much higher even if the DLX processor of D-1 was replaced by a faster superscalar processor. The performance of parallel DLX machines, D-16 and D-16s, did not scale up from a single processor version, D-1. This due to inability of a DLX processor to deal with overloading.

5.6 Other projects and feasibility

Currently there are many research and development projects going on in the area of multithreading [Moore96]. Among them are *Saarbrücken Parallel Random Access Machine* (SB-PRAM) [Abolhassan93] based on Fluent Abstract Machine [Ranade87] and Berkley-RISC I [Patterson82] style multithreaded processor [Keller94], *Alewife* Project based on Sparcle processors in MIT [Agarwal93], and *TERA MTA* supercomputer from Tera Computer Corporation [Alverson90].

Unlike MTAC and TERA, SB-PRAM and Sparcle are not high speed designs: A SB-PRAM processor runs at 8 MHz and a MIT Sparcle processor runs at 33 MHz. The GaAs based TERA runs at 333 MHz and it is capable of using up to 128 threads per processor. However, a

TERA does not have the a distributed register file architecture as a MTAC and fails to provide the instruction level parallelism and degree of superpipelining of MTAC.

MTAC processors contain a larger number of components compared to basic pipelined processors like DLX. This increases the silicon die area and manufacturing costs of MTAC processors. On the other hand, one does not need to use silicon area for large on-chip caches with MTAC saving die area for other use.

In addition to that, DLX cannot compete with MTAC in the area of parallel computing, because MTAC is superior in dealing with processor overloading as seen in machine level estimations: MTAC switches threads without any cost in every clock cycle, whereas DLX uses at least 70 clock cycles to switch to another thread. In addition to that, MTAC features a multithread pipeline and it can be extensively superpipelined without the fear of instruction dependencies whereas DLX features single threaded execution with five level single thread pipelining.

We believe that the clock frequency of a MTAC processor can be made as high as the clock frequency of fastest commercial processors (600 MHz) with current technology, or even higher, because there is no need for forwarding within a single clock cycle in MTAC, the structure of MTAC is very regular, and there are no long wires that can not be pipelined. In this sense MTAC closely resembles *Counterflow Pipeline Processor Architecture* (CFPP) from Sun Microsystems Laboratories and Oregon State University, which uses regular structure, local control and local communication to achieve maximum speed [Sproull94, Janik97]. CFPP is, however, not designed for parallel processing like MTAC.

We also believe that the maximum number of threads in MTAC tmax can be made as high as 500 with current technology, or even higher, because a MTAC processor can be divided to multiple chips placed in a multichip module, if it does not fit into a single chip. Thus, the latency of a fast communication network and the approximate 90 ns cycle time of current main memory building blocks, DRAM chips, could be hided in a parallel computer using MTAC processors and fast parallel communication system like CBM outlined in [Forsell96a].

5.7 Conclusions

We have described techniques for reducing performance loss due to low utilization of functional units and delays caused by instruction dependencies in thread-level parallel and instruction-level parallel architectures. We applied these techniques along with extensive superpipelining to a basic VLIW architecture. As a result we outlined the MultiThreaded Architecture with Chaining (MTAC), which specially suits for efficient PRAM simulation.

We evaluated four MTAC processors and one reference processor by simulations. The results are favorable for MTAC:

T5—a five unit implementation of MTAC—performed 2.7 times faster than a basic pipelined processor with the same number of functional units. T7, T11 and T19— six, ten and eighteen unit implementations of MTAC—performed 4.1, 8.1 and 16.6 times faster than the basic pipelined processor, respectively.

The better performance comes from better exploitation of available instruction-level parallelism, which is due to more than two times better utilization of functional units, the absence of pipeline and memory system hazards, and the ability of MTAC to execute code blocks containing dependencies within a single clock cycle.

In addition to better performance, the absence of pipeline and memory system hazards implies that there is no need for cache memories to get full performance. Furthermore, there are no long wires in MTAC, so everything in MTAC can be extensively superpipelined allowing for a very short clock cycle.

We also evaluated three machines based on MTAC procesors and three reference machines by simulations. The effect of of assumed memory systems was estimated and taken into account. The results are even better for MTAC based machines than in the case of plain processors:

T5-1—a machine with a single T5 processor and banked memory system—performs 11 times faster than a conventional sequential machine with a basic pipelined processor and a cached memory system. T5-16 and T19-16—16-processor machines based on T5 and T19 processors—perform respectively 90 to 350 and 580 to 2700 times faster than two 16-processor machines using the basic pipelined processor.

Functional unit chaining seems to improve the exploitation of instruction level parallelism to the level where the achieved speedup corresponds to the number of functional units in a processor at least in the case of small number of units: T11, which has almost twice the processing resources of T7, performs two times faster than T7 and T19, which has almost two times the processing resources of T11, performs two times faster than T11.

Chaining also simplifies the VLIW compiler technology required to achieve the full computing power, because MTAC can execute code containing true dependencies. Unfortunately the object code compiled for one MTAC processor is not compatible with another version of MTAC processor. This is a common problem with VLIW architectures, but it is a problem with superscalar processors too. The maximum performance cannot be achieved without recompilation.

A general purpose supercomputer cannot be based solely on MTAC, because there are certain strictly sequential problems that can be executed much faster with conventional processors. Moreover, high speed realtime applications may need faster response than MTAC can serve. A possible solution could be a two unit structure with a dedicated scalar unit just like in vector processors.

References

[Abolhas93]
F. Abolhassan, R. Drefenstedt, J. Keller, W. J. Paul and D. Scheerer, On the Physical Design of PRAMs, *Computer Journal* **36**, 8 (1993), 756-762.

[Agarwal93]
A. Agarwal, J. Kubiatowicz, J. Kranz, D. Lim, D. Yeung, G. D'Souza and M. Parkin, Sparcle: An Evolutionary Processor Design for Large-Scale Multiprocessors, *IEEE Micro*, (1993), 48-61.

[Alverson90]
R. Alverson, D. Callahan, D. Cummings, B. Kolblenz, A. Porterfield and B. Smith, The Tera Computer System, *Proceedings of the International Conference on Supercomputing*, June 11-15, 1990, Amsterdam, The Netherlands, 1-6.

[Brent74]
R. Brent, The parallel evaluation of general arithmetic expressions, *Journal of the ACM* **21**, (1974), 201-206.

[Burnett70]
G. J. Burnett and E. G. Coffman, A Study of Interleaved Memory Systems, *AFIPS Conference Proceedings SJCC*, 36 (1970), 467-474.

[Chang91]
P. Chang, S. Mahlke, W. Chen, N. Warter and W. Hwu, IMPACT: An architectural framework for multiple-instruction-issue processors, *Proceedings of the 18th Annual International Conference on Computer Architecture*, Association for Computing Machinery, New York, 1991, 266-275.

[Enslow74]
P. H. Enslow, *Multiprocessors and parallel processing*, John Wiley&Sons, New York, 1974.

[Feldman92]
Y. Feldman, E. Shapiro, Spatial Machines: A More Realistic Approach to Parallel Computation, *Communications of the ACM* **35**, 10 (1992) 61-73.

[Fisher81]
J. Fisher, Trace Scheduling: A technique for global microcode compaction, *IEEE Transactions on Computers* **C-30**, (1981), 478-490.

[Fisher83]
J. Fisher, Very Long Instruction Word Architectures and ELI-512, *Proceedings of the 10th Annual International Symposium on Computer Architecture*, Computer Society Press of the IEEE, 1983, 140-150.

[Forsell94]
M. Forsell, Are multiport memories physically feasible?, *Computer Architecture News* **22**, 4 (1994), 47-54.

[Forsell96a]
M. Forsell, V. Leppänen and M. Penttonen, Efficient Two-Level Mesh based Simulation of PRAMs, *Proceedings of the International Symposium on Parallel Architectures, Algorithms and Networks*, June 12-14, 1996, Beijing, China, 29-35.

[Forsell96b]
M. Forsell, Minimal Pipeline Architecture—an Alternative to Superscalar Architecture, *Microprocessors and Microsystems* **20**, 5 (1996), 277-284.

[Forsell97]
M. Forsell, MTACSim - A simulator for MTAC, *In preparation*, Department of Computer Science, University of Joensuu, Joensuu, 1997.

[Fortune78]
S. Fortune and J. Wyllie, Parallelism in Random Access Machines, *Proceedings of 10th ACM STOC*, Accosiation for Computing Machinery, New York, 1978, 114-118.

[Gajski83]
D. Gajski, D. Kuck, D. Lawrie and A. Sameh, CEDAR-A Large Scale Multiprocessor, *Proceedings of International Conference on Parallel Processing*,1983, 524-529.

[Gottlieb83]
A. Gottlieb, R. Grishman, C. P. Kruskal, K. P. McAuliffe, L. Rudolph and M. Snir, The NYU Ultracomputer - Designing a MIMD, shared-memory parallel machine, *IEEE Transactions on Computers* **C-32**, (1983), 175-189.

[Hennessy90]
J. Hennessy and D. Patterson, *Computer Architecture: A Quantitative Approach*, Morgan Kaufmann Publishers Inc., San Mateo, 1990.

[Hillis85]
W. D. Hillis, The Connection Machine, The MIT Press, Cambridge, 1985.

[Janik97]
K. Janik, S. Lu, M. Miller, Advances of the Counterflow Pipeline Microarchitecture, *to appear in Proceedings of the Third International Symposium on High-Performance Computer Architecture*, 1997.

[Johnson89]
W. M. Johnson, Super-Scalar Processor Design, *Technical Report No. CSL-TR-89-383*, Stanford University, Stanford, 1989.

[Jouppi89]
N. Jouppi and D. Wall, Available Instruction Level Parallelism for Superscalar and Superpipelined Machines, *Proceedings of the 3rd Conference on Architectural Support for Programming Languages and Operating Systems*, 1989, Boston, USA, 272-282.

[Karp69]
R. M. Karp and R. E. Miller, Parallel Program Schemata, *Journal of Computer and System Sciences* **3**, 2 (1969), 147-195.

[Keller94]
J. Keller, W. Paul, D. Scheerer, Realization of PRAMs: Processor Design, *Proceedings of WDAG '94*, 8th International Workshop on distributed Algorithms, 1994, Terschelling, The Netherlands, 17-27.

[Leighton91]
T. F. Leighton, *Introduction to Parallel Algorithms and Architectures: Arrays, Trees, Hypercubes*, Morgan Kaufmann, San Mateo, 1992.

[McColl92]
W. F. McColl, General Purpose Parallel Computing, *Lectures on Parallel Computation Proceedings 1991 ALCOM Spring School on Parallel Computation (Editors: A. M. Gibbons and P. Spirakis)*, Cambridge University Press, Cambridge, 1992, 333-387.

[Moore96]
S. Moore, *Multithreaded Processor Design*, Kluwer Academic Publishers, Boston, 1996.

[Nicolau84]
A. Nicolau and J. Fisher, Measuring the parallelism available for very long instruction word architectures, *IEEE Transactions on Computers* **C-33**, 11 (1984), 968-976.

[Patterson82]
D. Patterson and C. Sequin, A VLSI RISC, *Computer* **15**, 9 (1982), 8-21.

[Pfister85]
G. F. Pfister, W. C. Brantley, D. A. George, S. L. Harvey, W. J. Kleinfelder, K. P. McAuliffe, E.A. Melton, V. A. Norton and J. Weiss, The IBM Research Parallel Processor Prototype (RP3): Introduction and Architecture, *Proceedings of International Conference on Parallel Processing* (1985), 764-771.

[Prechelt93]
L. Prechelt, Measurements of MasPar MP-1216A Communication Operations, *Technical Report 01/93*, Universität Karlsruhe, Karlsruhe, 1993.

[Ranade87]
A. G. Ranade, S. N. Bhatt, S. L. Johnson, The Fluent Abstract Machine, *Technical Report Series BA87-3*, Thinking Machines Corporation, Bedford, 1987.

[Schwarz66]
J. T. Schwarz, Large Parallel Computers, *Journal of the ACM* **13**, 1 (1966), 25-32.

[Schwarz80]
J. T. Schwarz, Ultracomputers, *ACM Transactions on Programming Languages and Systems* **2**, 4 (1980), 484-521.

[Sproull94]
R. Sproull, I. Sutherland and C. Molnar, Counterflow Pipeline Processor Architecture, *IEEE Design and Test of Computers* **11**, 3 (1994), 48-59.

APPENDIX A. THE DLX ARCHITECTURE

DLX is a simple load/store architecture developed for Hennessy's and Patterson's book *Computer Architecture—A Quantitative Approach* [Hennessy90] (see Figure A.1). In this appendix we describe the DLX architecture like it is described in Hennessy's and Patterson's book.

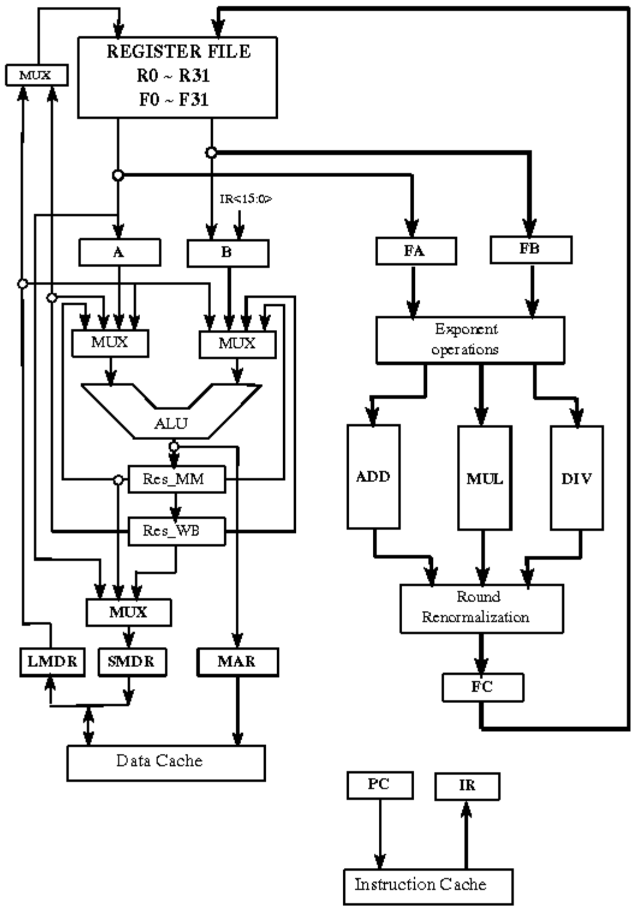


Figure A.1 The block diagram of DLX processor.

Note: DLX assembly language notation uses reverse ordering of operands: The destination operand is always on the left and the source operands are always on the right.

A.1 ARCHITECTURE

The DLX architecture has thirty two 32-bit general purpose registers. The value of R0 is always 0. Additionally, there is a set of floating-point registers, which can be used as 32 single-precision (32-bit) registers, or as even-odd pairs holding double precision values. The 64-bit floating-point registers are named F0,F2,...,F28,F30. Both single and double precision operations are provided.

Memory is byte addressable in Big Endian mode with a 32-bit address. All memory references are through loads or stores between memory and either general purpose registers or floating point registers. All memory accesses must be aligned.

All instructions are 32-bits and must be aligned. The instruction set is described later in this appendix.

A.2 OPERATIONS

There are four classes of instructions: load and stores, ALU operations, branches and jumps, and floating-point operations.

Any of the general purpose or floating point registers may be loaded or stored except that loading R0 has no effect. There is a single addressing mode, base register + 16-bit signed offset. Halfword and byte loads place the loaded object in the lower portion of the register. The upper portion of the register is filled with either the sign extension of the loaded value or zeroes, depending on the opcode.

All ALU instructions are register-register instructions. The operations include simple arithmetic and logical operations: add, subtract, AND, OR, XOR and shifts. Immediate forms of all these instructions, with a 16-bit sign extended immediate, are provided. The operation LHI (load high immediate) loads the top half of a register, while setting the lower half to zero. This allows a full 32-bit constants to be built in two instructions.

There are also compare instructions, which compare two registers ($=$, $\neq$, $<$, $>$, $\leq$, $\geq$). If the condition is true, these instructions place a 1 in the destination register (to represent true); otherwise they place the value 0.

Control is handled through a set of jumps and a set of branches. The four jump instructions are differentiated by the two ways to specify the destination address and whether or not a link is made. Two jumps use a 26-bit signed offset added to the program counter (of the instruction sequentially following the jump) to determine the destination address. The other two jumps specify a register that contains the destination address. There are two flavors of jumps: plain jump, and jump and link (used for procedure calls). The latter places the return address in R31.

All branches are conditional. The branch condition is specified by the instruction, which may test the register source for zero or non-zero. The tested register may be a data value or the result of a compare.

A.3 PIPELINE

DLX uses five stage pipelining, which divides instruction execution into five stages—Instruction Fetch (IF), Instruction Decode and Operand Fetch (DE), Execution and Effective Address Calculation (EX), Memory Reference (ME), and Write Result Back to the Register File (WB)

In the IF stage an opcode of an instruction is fetched according to address pointed by PC and PC is increased. During the DE stage opcode is decoded and operands are selected and fetched by multiplexers controlled by opcode in operation register. In addition, new PC value is computed and PC value is changed if the condition of branch instruction is true for branch instructions.

In the EX stage an ALU operation is performed for ALU instructions, or a address calculation is performed for load and store instructions.

During the ME stage the result of ALU operation is transferred to the WB stage for ALU instructions, or memory access is performed according to address calculated during previous stage for load and store instructions. Finally, the WB stage writes the result of ALU operation or memory load into a register.

DLX uses forwarding and delayed branching to avoid some pipeline hazards. A simple hardware solution is used to implement delayed branches with one delay slot: A separate adder is used to calculate the branch target address during the DE stage. Additionally, special logic devoted to testing is used to find out whether the branch is taken or not during the DE stage.

A.4 THE COMPLETE INSTRUCTION SET OF DLX

Symbols used for describing DLX instructions

Fd	Floating destination register
Fs1	Floating source register 1
Fs2	Floating source register 2
Rd	Integer destination register
Rs1	Integer source register 1
Rs2	Integer source register 2
d16	16-bit signed offset
d26	26-bit signed offset
i16	16-bit signed immediate

Load		
LB	Rd,d16(Rs1)	Load Byte
LBU	Rd,d16(Rs1)	Load Byte Unsigned
LH	Rd,d16(Rs1)	Load Halfword
LHU	Rd,d16(Rs1)	Load Halfword Unsigned
LW	Rd,d16(Rs1)	Load Word
LF	Fd,d16(Rs1)	Load Floating
LD	Fd,d16(Rs1)	Load Double
Store		
SB	d16(Rd),Rs1	Store Byte
SH	d16(Rd),Rs1	Store Halfword
SW	d16(Rd),Rs1	Store Word
SF	d16(Rd),Fs1	Store Floating
SD	d16(Rd),Fs1	Store Double
Transfer		
MOVI2S	S,Rs1	Move to Special Register
MOVS2I	Rd,S	Move from Special Register
MOVF	Fd,Fs1	Move Floating
MOVD	Fd,Fs1	Move Double
MOVFP2I	Rd,Fs1	Move word from floating to integer register
MOVI2FP	Fd,Rs1	Move word from integer to floating register
Arithmetic		
ADD	Rd,Rs1,Rs2	Add
ADDI	Rd,Rs1,i16	Add Immediate
ADDU	Rd,Rs1,Rs2	Add Unsigned

ADDUI	Rd,Rs1,i16	Add Unsigned Immediate
SUB	Rd,Rs1,Rs2	Subtract
SUBI	Rd,Rs1,i16	Subtract Immediate
SUBU	Rd,Rs1,Rs2	Subtract Unsigned
SUBUI	Rd,Rs1,i16	Subtract Unsigned Immediate
MULT	Fd,Fs1,Fs2	Multiply
MULTU	Fd,Fs1,Fs2	Multiply Unsigned
DIV	Fd,Fs1,Fs2	Divide
DIVU	Fd,Fs1,Fs2	Divide Unsigned
Logical		
AND	Rd,Rs1,Rs2	Logical And
ANDI	Rd,Rs1,i16	Logical And Immediate
OR	Rd,Rs1,Rs2	Logical Or
ORI	Rd,Rs1,i16	Logical Or Immediate
XOR	Rd,Rs1,Rs2	Logical Exclusive Or
XORI	Rd,Rs1,i16	Logical Exclusive Or Immediate
LHI	Rd,Rs1,i16	Load High part
SLL	Rd,Rs1,Rs2	Logical Shift Left
SRL	Rd,Rs1,Rs2	Logical Shift Right
SRA	Rd,Rs1,Rs2	Arithmetic Shift Right
SLLI	Rd,Rs1,i16	Logical Shift Left Immediate
SRLI	Rd,Rs1,i16	Logical Shift Right Immediate
SRAI	Rd,Rs1,i16	Arithmetic Shift Right Immediate
SLT	Rd,Rs1,Rs2	Set Less Than
SGT	Rd,Rs1,Rs2	Set Greater Than
SLE	Rd,Rs1,Rs2	Set Less or Equal
SGE	Rd,Rs1,Rs2	Set Greater or Equal
SEQ	Rd,Rs1,Rs2	Set Equal
SNE	Rd,Rs1,Rs2	Set Not Equal
SLTI	Rd,Rs1,i16	Set Less Than Immediate
SGTI	Rd,Rs1,i16	Set Greater Than Immediate
SLEI	Rd,Rs1,i16	Set Less or Equal Immediate
SGEI	Rd,Rs1,i16	Set Greater or Equal Immediate
SEQI	Rd,Rs1,i16	Set Equal Immediate
SNEI	Rd,Rs1,i16	Set Not Equal Immediate
Control		
BEQZ	Rs1,d16	Branch Equal Zero
BNEZ	Rs1,d16	Branch Not Equal Zero
BFPT	Rs1,d16	Floating Branch Equal Zero
BFPF	Rs1,d16	Floating Branch Not Equal Zero
J	d26	Jump

JR	Rs1	Jump Register
JAL	d26	Jump and Link
JALR	Rs1	Jump and Link Register
TRAP	d26	Trap
RFE	d26	Return from Exception
Floating point		
ADDF	Fd,Fs1,Fs2	Add Floating
ADDD	Fd,Fs1,Fs2	Add Double
SUBF	Fd,Fs1,Fs2	Subtract Floating
SUBD	Fd,Fs1,Fs2	Subtract Double
MULTF	Fd,Fs1,Fs2	Multiply Floating
MULTD	Fd,Fs1,Fs2	Multiply Double
DIVF	Fd,Fs1,Fs2	Divide Floating
DIVD	Fd,Fs1,Fs2	Divide Double
CVTF2D	Fd,Fs1	Convert Floating to Double
CVTF2I	Fd,Fs1	Convert Floating to Integer
CVTD2F	Fd,Fs1	Convert Double to Floating
CVTD2I	Fd,Fs1	Convert Double to Integer
CVTI2F	Fd,Fs1	Convert Integer to Floating
CVTI2D	Fd,Fs1	Convert Integer to Double
LTF	Fs1,Fs2	Set Less Than Floating
LTD	Fs1,Fs2	Set Less Than Double
GTF	Fs1,Fs2	Set Greater Than Floating
GTD	Fs1,Fs2	Set Greater Than Double
LEF	Fs1,Fs2	Set Less or Equal Floating
LED	Fs1,Fs2	Set Less or Equal Double
GEF	Fs1,Fs2	Set Greater or Equal Floating
GED	Fs1,Fs2	Set Greater or Equal Double
EQF	Fs1,Fs2	Set Equal Floating
EQD	Fs1,Fs2	Set Equal Double
NEF	Fs1,Fs2	Set Not Equal Floating
NED	Fs1,Fs2	Set Not Equal Double

References

[Hennessy90]

J. Hennessy and D. Patterson, *Computer Architecture: A Quantitative Approach*, Morgan Kaufmann Publishers Inc., 1990.

APPENDIX B. THE SUPERDLX ARCHITECTURE

SuperDLX is an abstract superscalar processor architecture, which implements multiple-out-of-order issue, multiple-out-of-order completion, register renaming and branch prediction [Moura93] (see Figure B.1). It uses the same instruction set and register structure as DLX (see Appendix A).

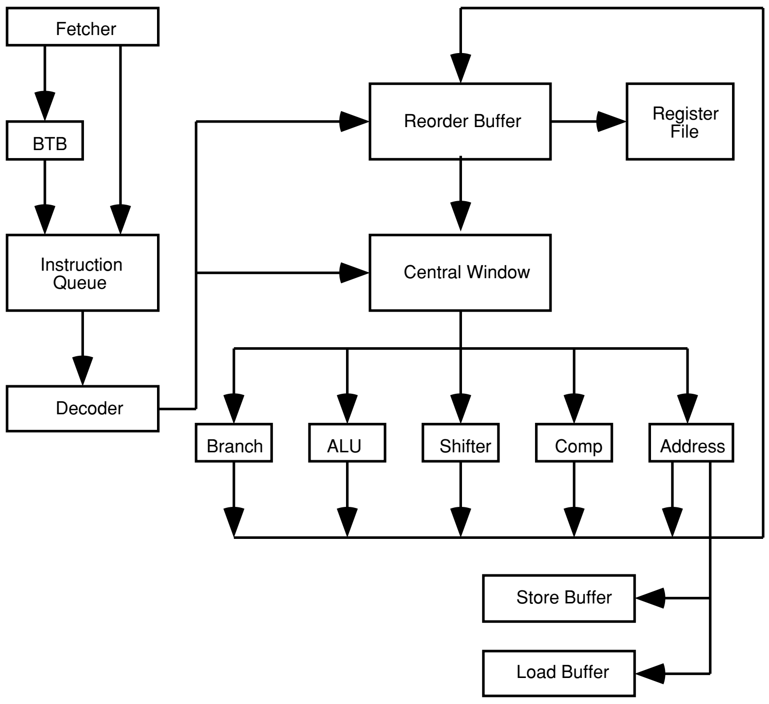


Figure B.1 The block diagram of superDLX processor. The floating point unit is not shown.

Being a scalable architecture, superDLX has parametric number of functional units. We chose an example configuration for our benchmarks to have 4 ALUs, 1 compare unit, 1 branch unit and 4 memory units. In addition, we assumed that superDLX is able to detect and eliminate empty operations in DLX code. Otherwise the performance of superDLX would be remarkably lower than shown in Chapter 4. We will use the name superDLX processor for this configuration of superDLX architecture.

SuperDLX has a five stage pipeline which is not similar to one in DLX. The pipeline stages in superDLX are: Instruction Fetch (IF), Decode (DE), Execute (EX), Write Back (WB), and Result Commit (RC).

In the IF stage instructions are fetched from the cache and placed in an instruction queue. Because instruction fetching depends on the results of branch instruction execution, 2-bit dynamic branch prediction is used. The maximum number of instruction fetched per clock cycle in superDLX processor is 10.

In the DE stage, instructions are taken from the instruction queue, decoded and dispatched to their appropriate operational unit (entries in the reorder buffer and the central instruction window are created). The maximum number of instruction decoded per clock cycle in superDLX processor is 10.

During the EX stage issue logic examines instructions in the instruction window and selects the ready ones for execution. An instruction is ready if the operands are ready and the functional unit is available. When two instructions conflict for the same functional unit the oldest one has the highest priority.

The processor is allowed to determine the effective outcome of predicted branches: If a branch prediction happens to be wrong, the instructions following the branch in reorder buffers are flushed.

During the WB stage the completed operations are identified in the reorder buffer and freed in the corresponding functional units. The completed results are validated and forwarded to the instructions that need them in the instruction windows.

In the RC stage validated results in the reorder buffer are sent to the register file. Invalidated instruction (that follow a wrong branch prediction) are discarded. The maximum number of result committed per clock cycle in superDLX processor is 10.

References

[Moura93]
C. Moura, SuperDLX - A Generic Superscalar Simulator, *ACAPS Technical Memo 64*, McGill University, Montreal, Canada, 1993.

APPENDIX C. THE MINIMAL PIPELINE ARCHITECTURE

The Minimal Pipeline Architecture (MPA) is an abstract superscalar load/store architecture using VLIW-style instruction scheduling policy and very short pipelines. In this appendix we will explain the symbols used to describe the MPA. Additionally we will give a complete list of MPA instructions. The architecture is described in Chapter 4 in detail.

The symbols used for describing MPA operations:

Ax	Result of ALU x
AAx	Input register 1 of ALU x
ABx	Input register 2 of ALU x
AIA	Input register 1 of ALU for integer condition codes
AIB	Input register 2 of ALU for integer condition codes
AIC	Result of ALU for integer condition codes
d	32-bit immediate value
I[x]	Location x of instruction memory
Mx	Result of memory x operation
Mx[y]	Location y of memory x
MAx	Address register for memory x
MDx	Data register for memory x
MuxY	Multiplexer for Y.
O	Operation register
Ox	Operand x of an instruction
OO	Result of operation code fetch
O1	Immediate operand field of operation register
PC	Program counter
RA	Return address
Rx	Contents of register x
Xx	Ax, Mx, Ox or Rx

C.1 THE COMPLETE INSTRUCTION SET OF THE MPA

Load

LDBn	Xx	Load byte from memory n address Xx
LDBUn	Xx	Load byte from memory n address Xx unsigned
LDHn	Xx	Load halfword from memory n address Xx
LDHUn	Xx	Load halfword from memory n address Xx unsigned
LDn	Xx	Load word from memory n address Xx

Store		
STBn	Xx,Xy	Store byte Xx to memory n address Xy
STHn	Xx,Xy	Store halfword Xx to memory n address Xy
STn	Xx,Xy	Store word Xx to memory n address Xy
Write Back		
WBn	Xx	Write Xx to register Rn. Old value of Rn is preserved if x is r-1.
Operand Input		
OPn	d	Input value d into operand n
Align		
ALBn	Xx,Xy	Align load byte Xx by Xy in ALU n
ALBUn	Xx,Xy	Align load byte Xx by Xy in ALU n unsigned
ALHn	Xx,Xy	Align load halfword Xx by Xy in ALU n
ALHUn	Xx,Xy	Align load halfword Xx by Xy in ALU n unsigned
ASSBn	Xx,Xy	Align store source byte Xx by Xy in ALU n
ASSHn	Xx,Xy	Align store source halfword Xx by Xy in ALU n
ASDBn	Xx,Xy	Align store destination byte Xx by Xy in ALU n
ASDHn	Xx,Xy	Align store destination halfword Xx by Xy in ALU n
Arithmetic		
ADDn	Xx,Xy	Add Xx and Xy in ALU n
SUBn	Xx,Xy	Subtract Xy from Xx in ALU n
MULn	Xx,Xy	Multiply Xx by Xy in ALU n
MULUn	Xx,Xy	Multiply Xx by Xy in ALU n unsigned
DIVn	Xx,Xy	Divide Xx by Xy in ALU n
DIVUn	Xx,Xy	Divide Xx by Xy in ALU n unsigned
MODn	Xx,Xy	Determine Xx modulo Xy in ALU n
MODUn	Xx,Xy	Determine Xx modulo Xy in ALU n unsigned
Logical		
ANDn	Xx,Xy	And of Xx and Xy in ALU n
ORn	Xx,Xy	Or of Xx and Xy in ALU n
XORn	Xx,Xy	Exclusive or of Xx and Xy in ALU n
ANDNn	Xx,Xy	And not of Xx and Xy in ALU n
ORNn	Xx,Xy	Or not of Xx and Xy in ALU n
XNORn	Xx,Xy	Exclusive nor of Xx and Xy in ALU n
SHRn	Xx,Xy	Shift right Xx by Xy in ALU n
SHLn	Xx,Xy	Shift left Xx by Xy in ALU n
SHRAn	Xx,Xy	Shift right Xx by Xy in ALU n arithmetic
NOPn		No operation in ALU n

RORn	Xx,Xy	Rotate right Xx by Xy in ALU n
ROLn	Xx,Xy	Rotate left Xx by Xy in ALU n

Compare

SEQ	Xx,Xy	Set IC if Xx equals Xy
SNE	Xx,Xy	Set IC if Xx not equals Xy
SLT	Xx,Xy	Set IC if Xx is less than Xy
SLE	Xx,Xy	Set IC if Xx is less than or equals Xy
SGT	Xx,Xy	Set IC if Xx is greater than Xy
SGE	Xx,Xy	Set IC if Xx is greater than or equals Xy
SLTU	Xx,Xy	Set IC if Xx is less than Xy unsigned
SLEU	Xx,Xy	Set IC if Xx is less than or equals Xy unsigned
SGTU	Xx,Xy	Set IC if Xx is greater than Xy unsigned
SGEU	Xx,Xy	Set IC if Xx is greater than or equals Xy unsigned

ADDCC	Xx,Xy	Add Xx and Xy and set CC
SUBCC	Xx,Xy	Subtract Xy from Xx and set CC
MULCC	Xx,Xy	Multiply Xx by Xy and set CC
MULUCC	Xx,Xy	Multiply Xx by Xy unsigned and set CC
DIVCC	Xx,Xy	Divide Xx by Xy and set CC
DIVUCC	Xx,Xy	Divide Xx by Xy unsigned and set CC
MODCC	Xx,Xy	Determine Xx modulo Xy and set CC
MODUCC	Xx,Xy	Determine Xx modulo Xy unsigned and set CC

ADDXn	Xx,Xy	Add with carry Xx and Xy in ALU n
SUBXn	Xx,Xy	Subtract with carry Xy from Xx in ALU n
ADDXCC	Xx,Xy	Add with carry Xx and Xy and set CC
SUBXCC	Xx,Xy	Subtract with carry Xy from Xx and set CC
TADDn	Xx,Xy	Tagged add Xx and Xy in ALU n
TSUBn	Xx,Xy	Tagged subtract Xy from Xx in ALU n
TADDCC	Xx,Xy	Tagged add Xx and Xy and set CC
TSUBCC	Xx,Xy	Tagged subtract Xy from Xx and set CC

ANDCC	Xx,Xy	And of Xx and Xy in ALU n and set CC
ORCC	Xx,Xy	Or of Xx and Xy in ALU n and set CC
XORCC	Xx,Xy	Exclusive or of Xx and Xy in ALU n and set CC
ANDNCC	Xx,Xy	And not of Xx and Xy in ALU n and set CC
ORNCC	Xx,Xy	Or not of Xx and Xy in ALU n and set CC
XNORCC	Xx,Xy	Exclusive nor of Xx and Xy in ALU n and set CC

Control

BEQZ	Ox	Branch to Ox if IC equals zero
BNEZ	Ox	Branch to Ox if IC not equals zero

FBT	Ox	Floating Branch to Ox if FC equals zero
FBF	Ox	Floating Branch to Ox if FC not equals zero
JMP	Xx	Jump to Xx
JMPL	Xx	Jump and link PC+1 to register RA
TRAP	Xx	Trap
BA	Ox	Branch always to Ox
BN	Ox	Branch newer to Ox
BNE	Ox	Branch on not equal to Ox (BNZ)
BE	Ox	Branch on equal to Ox (BZ)
BG	Ox	Branch on greater to Ox
BLE	Ox	Branch on less than or equal to Ox
BGE	Ox	Branch on greater than or equal to Ox
BL	Ox	Branch on less than to Ox
BGU	Ox	Branch on greater to Ox unsigned
BLEU	Ox	Branch on less than or equal to Ox unsigned
BCC	Ox	Branch on carry clear to Ox (BGEU)
BCS	Ox	Branch on carry set to Ox (BLU)
BPOS	Ox	Branch on positive to Ox
BNEG	Ox	Branch on negative to Ox
BVC	Ox	Branch on overflow clear to Ox
BVS	Ox	Branch on overflow set to Ox

Register window

SAVE	Save register window
REST	Restore register window

Floating Point

FADDn	Xx,Xy	Floating add Xx and Xy in ALU n
FSUBn	Xx,Xy	Floating subtract Xy from Xx in ALU n
FMULn	Xx,Xy	Floating multiply Xx by Xy in ALU n
FDIVn	Xx,Xy	Floating divide Xy by Xx in ALU n
FMODn	Xx,Xy	Floating Xx modulo Xy in ALU n
FSEQ	Xx,Xy	Floating set FC if Xx equals Xy
FSNE	Xx,Xy	Floating set FC if Xx not equals Xy
FSLT	Xx,Xy	Floating set FC if Xx is less than Xy
FSLE	Xx,Xy	Floating set FC if Xx is less than or equals Xy
FSGT	Xx,Xy	Floating set FC if Xx is greater than Xy
FSGE	Xx,Xy	Floating set FC if Xx is greater than or equals Xy
DADDn	Xx,Xy	Double add Xx and Xy in ALU n
DSUBn	Xx,Xy	Double subtract Xy from Xx in ALU n
DMULn	Xx,Xy	Double multiply Xx by Xy in ALU n

DDIVn	Xx,Xy	Double divide Xy by Xx in ALU n
DMODn	Xx,Xy	Double Xx modulo Xy in ALU n
DSEQ	Xx,Xy	Double set FC if Xx equals Xy
DSNE	Xx,Xy	Double set FC if Xx not equals Xy
DSLT	Xx,Xy	Double set FC if Xx is less than Xy
DSLE	Xx,Xy	Double set FC if Xx is less than or equals Xy
DSGT	Xx,Xy	Double set FC if Xx is greater than Xy
DSGE	Xx,Xy	Double set FC if Xx is greater than or equals Xy
CVTFIn	Xx	Convert floating Xx to integer in ALU n
CVTFDn	Xx	Convert floating Xx to double in ALU n
CVTDIn	Xx	Convert double Xx to integer in ALU n
CVTDFn	Xx	Convert double Xx to floating in ALU n
CVTIFn	Xx	Convert integer Xx to floating in ALU n
CVTIDn	Xx	Convert integer Xx to double in ALU n

APPENDIX D. THE MULTITHREADED ARCHITECTURE WITH CHAINING

The Multithreaded Architecture with Chaining (MTAC) is an abstract multithreaded load/store architecture using VLIW-style instruction scheduling policy and functional unit chaining. In this appendix we will explain the symbols used to describe the MTAC. Additionally we will give a complete list of MTAC instructions. The architecture is described in Chapter 5 in detail.

The symbols used for describing MPA operations:

Ax	Result of ALU x
AAx	Input register 1 of ALU x
ABx	Input register 2 of ALU x
AIA	Input register 1 of ALU for integer condition codes
AIB	Input register 2 of ALU for integer condition codes
AIC	Result of ALU for integer condition codes
d	32-bit immediate value
I[x]	Location x of instruction memory
Mx	Result of memory x operation
Mx[y]	Location y of memory x
MAx	Address register for memory x
MDx	Data register for memory x
MuxY	Multiplexer for Y.
O	Operation register
Ox	Operand x of an instruction
OO	Result of operation code fetch
O1	Immediate operand field of operation register
PC	Program counter
RA	Return address
Rx	Contents of register x
Xx	Ax, Mx, Ox or Rx

D.1 THE COMPLETE INSTRUCTION SET OF THE MTAC

Load

LDBn	Xx	Load byte from memory n address Xx
LDBUn	Xx	Load byte from memory n address Xx unsigned
LDHn	Xx	Load halfword from memory n address Xx
LDHUn	Xx	Load halfword from memory n address Xx unsigned
LDn	Xx	Load word from memory n address Xx

Store		
STBn	Xx,Xy	Store byte Xx to memory n address Xy
STHn	Xx,Xy	Store halfword Xx to memory n address Xy
STn	Xx,Xy	Store word Xx to memory n address Xy
Write Back		
WBn	Xx	Write Xx to register Rn. Old value of Rn is preserved if x is r-1.
Operand Input		
OPn	d	Input value d into operand n
Align		
ALBn	Xx,Xy	Align load byte Xx by Xy in ALU n
ALBUn	Xx,Xy	Align load byte Xx by Xy in ALU n unsigned
ALHn	Xx,Xy	Align load halfword Xx by Xy in ALU n
ALHUn	Xx,Xy	Align load halfword Xx by Xy in ALU n unsigned
ASSBn	Xx,Xy	Align store source byte Xx by Xy in ALU n
ASSHn	Xx,Xy	Align store source halfword Xx by Xy in ALU n
ASDBn	Xx,Xy	Align store destination byte Xx by Xy in ALU n
ASDHn	Xx,Xy	Align store destination halfword Xx by Xy in ALU n
Arithmetic		
ADDn	Xx,Xy	Add Xx and Xy in ALU n
SUBn	Xx,Xy	Subtract Xy from Xx in ALU n
MULn	Xx,Xy	Multiply Xx by Xy in ALU n
MULUn	Xx,Xy	Multiply Xx by Xy in ALU n unsigned
DIVn	Xx,Xy	Divide Xx by Xy in ALU n
DIVUn	Xx,Xy	Divide Xx by Xy in ALU n unsigned
MODn	Xx,Xy	Determine Xx modulo Xy in ALU n
MODUn	Xx,Xy	Determine Xx modulo Xy in ALU n unsigned
SELn	Xx,Xy	Select Xx or Xy according to the result of previous compare operation in functional unit chain
Logical		
ANDn	Xx,Xy	And of Xx and Xy in ALU n
ORn	Xx,Xy	Or of Xx and Xy in ALU n
XORn	Xx,Xy	Exclusive or of Xx and Xy in ALU n
ANDNn	Xx,Xy	And not of Xx and Xy in ALU n
ORNn	Xx,Xy	Or not of Xx and Xy in ALU n
XNORn	Xx,Xy	Exclusive nor of Xx and Xy in ALU n

SHRn	Xx,Xy	Shift right Xx by Xy in ALU n
SHLn	Xx,Xy	Shift left Xx by Xy in ALU n
SHRAn	Xx,Xy	Shift right Xx by Xy in ALU n arithmetic
NOPn		No operation in ALU n
RORn	Xx,Xy	Rotate right Xx by Xy in ALU n
ROLn	Xx,Xy	Rotate left Xx by Xy in ALU n

Compare

SEQ	Xx,Xy	Set IC if Xx equals Xy
SNE	Xx,Xy	Set IC if Xx not equals Xy
SLT	Xx,Xy	Set IC if Xx is less than Xy
SLE	Xx,Xy	Set IC if Xx is less than or equals Xy
SGT	Xx,Xy	Set IC if Xx is greater than Xy
SGE	Xx,Xy	Set IC if Xx is greater than or equals Xy
SLTU	Xx,Xy	Set IC if Xx is less than Xy unsigned
SLEU	Xx,Xy	Set IC if Xx is less than or equals Xy unsigned
SGTU	Xx,Xy	Set IC if Xx is greater than Xy unsigned
SGEU	Xx,Xy	Set IC if Xx is greater than or equals Xy unsigned

ADDCC	Xx,Xy	Add Xx and Xy and set CC
SUBCC	Xx,Xy	Subtract Xy from Xx and set CC
MULCC	Xx,Xy	Multiply Xx by Xy and set CC
MULUCC	Xx,Xy	Multiply Xx by Xy unsigned and set CC
DIVCC	Xx,Xy	Divide Xx by Xy and set CC
DIVUCC	Xx,Xy	Divide Xx by Xy unsigned and set CC
MODCC	Xx,Xy	Determine Xx modulo Xy and set CC
MODUCC	Xx,Xy	Determine Xx modulo Xy unsigned and set CC

ADDXn	Xx,Xy	Add with carry Xx and Xy in ALU n
SUBXn	Xx,Xy	Subtract with carry Xy from Xx in ALU n
ADDXCC	Xx,Xy	Add with carry Xx and Xy and set CC
SUBXCC	Xx,Xy	Subtract with carry Xy from Xx and set CC
TADDn	Xx,Xy	Tagged add Xx and Xy in ALU n
TSUBn	Xx,Xy	Tagged subtract Xy from Xx in ALU n
TADDCC	Xx,Xy	Tagged add Xx and Xy and set CC
TSUBCC	Xx,Xy	Tagged subtract Xy from Xx and set CC

ANDCC	Xx,Xy	And of Xx and Xy in ALU n and set CC
ORCC	Xx,Xy	Or of Xx and Xy in ALU n and set CC
XORCC	Xx,Xy	Exclusive or of Xx and Xy in ALU n and set CC
ANDNCC	Xx,Xy	And not of Xx and Xy in ALU n and set CC
ORNCC	Xx,Xy	Or not of Xx and Xy in ALU n and set CC
XNORCC	Xx,Xy	Exclusive nor of Xx and Xy in ALU n and set CC

Control

BEQZ	Ox	Branch to Ox if IC equals zero	
BNEZ	Ox	Branch to Ox if IC not equals zero	
FBT	Ox	Floating Branch to Ox if FC equals zero	
FBF	Ox	Floating Branch to Ox if FC not equals zero	
JMP	Xx	Jump to Xx	
JMPL	Xx	Jump and link PC+1 to register RA	
TRAP	Xx	Trap	
BA	Ox	Branch always to Ox	
BN	Ox	Branch newer to Ox	
BNE	Ox	Branch on not equal to Ox	(BNZ)
BE	Ox	Branch on equal to Ox	(BZ)
BG	Ox	Branch on greater to Ox	
BLE	Ox	Branch on less than or equal to Ox	
BGE	Ox	Branch on greater than or equal to Ox	
BL	Ox	Branch on less than to Ox	
BGU	Ox	Branch on greater to Ox unsigned	
BLEU	Ox	Branch on less than or equal to Ox unsigned	
BCC	Ox	Branch on carry clear to Ox	(BGEU)
BCS	Ox	Branch on carry set to Ox	(BLU)
BPOS	Ox	Branch on positive to Ox	
BNEG	Ox	Branch on negative to Ox	
BVC	Ox	Branch on overflow clear to Ox	
BVS	Ox	Branch on overflow set to Ox	

Register window

SAVE	Save register window
REST	Restore register window

Floating Point

FADDn	Xx,Xy	Floating add Xx and Xy in ALU n
FSUBn	Xx,Xy	Floating subtract Xy from Xx in ALU n
FMULn	Xx,Xy	Floating multiply Xx by Xy in ALU n
FDIVn	Xx,Xy	Floating divide Xy by Xx in ALU n
FMODn	Xx,Xy	Floating Xx modulo Xy in ALU n
FSEQ	Xx,Xy	Floating set FC if Xx equals Xy
FSNE	Xx,Xy	Floating set FC if Xx not equals Xy
FSLT	Xx,Xy	Floating set FC if Xx is less than Xy
FSLE	Xx,Xy	Floating set FC if Xx is less than or equals Xy
FSGT	Xx,Xy	Floating set FC if Xx is greater than Xy
FSGE	Xx,Xy	Floating set FC if Xx is greater than or equals Xy

DADDn	Xx,Xy	Double add Xx and Xy in ALU n
DSUBn	Xx,Xy	Double subtract Xy from Xx in ALU n
DMULn	Xx,Xy	Double multiply Xx by Xy in ALU n
DDIVn	Xx,Xy	Double divide Xy by Xx in ALU n
DMODn	Xx,Xy	Double Xx modulo Xy in ALU n
DSEQ	Xx,Xy	Double set FC if Xx equals Xy
DSNE	Xx,Xy	Double set FC if Xx not equals Xy
DSLT	Xx,Xy	Double set FC if Xx is less than Xy
DSLE	Xx,Xy	Double set FC if Xx is less than or equals Xy
DSGT	Xx,Xy	Double set FC if Xx is greater than Xy
DSGE	Xx,Xy	Double set FC if Xx is greater than or equals Xy
CVTFIn	Xx	Convert floating Xx to integer in ALU n
CVTFDn	Xx	Convert floating Xx to double in ALU n
CVTDIn	Xx	Convert double Xx to integer in ALU n
CVTDFn	Xx	Convert double Xx to floating in ALU n
CVTIFn	Xx	Convert integer Xx to floating in ALU n
CVTIDn	Xx	Convert integer Xx to double in ALU n