ABSTRACT

Title of Thesis: MEMORY SUBSYSTEM DESIGN FOR EXPLICIT MULTITHREADING ARCHITECTURES

Degree candidate: Joseph Nuzman

Degree and year: Master of Science, 2003

Thesis directed by: Professor Uzi Vishkin
Department of Electrical and Computer Engineering

Explicit multithreading (XMT) is a parallel programming approach for exploiting on-chip parallelism. An important enabler for XMT is sufficient memory bandwidth to support parallelism. For targeted deep-submicron VLSI processes, chip designers will be faced with the widely acknowledged issues of rising interconnect RC delays and shortening clock periods. Comprehensive memory design for an XMT architecture has never before been rigorously studied. This thesis relies on an examination the implications of the XMT programming model on memory subsystem design to motivate a potential framework for on-chip memory interconnection. Many system-level issues are considered, and analytical electrical interconnect modeling is used to demonstrate the physical viability of new structures in future processes. It is estimated that a chip, built in a 2008 process with 1024 hardware execution contexts, may be capable of a sustained on-chip memory transaction throughput of 1430 GB/s.

MEMORY SUBSYSTEM DESIGN FOR EXPLICIT MULTITHREADING ARCHITECTURES

by

Joseph Nuzman

Thesis submitted to the Faculty of the Graduate School of the
University of Maryland at College Park in partial fulfillment
of the requirements for the degree of
Master of Science
2003

DEDICATION

To my mother, Susan, and father, Dwayne.

ACKNOWLEDGEMENTS

Firstly, I would like to thank Dr. Uzi Vishkin. I first began working with Dr. Vishkin as an undergrad in February 1997, and have been involved with his research, in some form or another, for over six years. It is a rare privilege to have been so deeply involved in a project as challenging, ambitious, and expansive as XMT. In addition to guiding this work, Dr. Vishkin has provided many lessons on the value of tenacity. Thanks, Uzi.

I owe a debt of gratitude to many others who have influenced my work. Shlomit Dascal helped to mentor me during my first steps in research. Efraim Berkovich helped to lay much of the groundwork for my later research. Particularly helpful was our collaboration on the first XMT simulator. Dr. Manoj Franklin was kind enough to provide direction during certain stages of my research. Dr. Jeff Davis provided helpful technical input. Aydin Balkan provided many helpful comments on this thesis.

Lastly, I'd like to acknowledge the many contributions of Dorit Naishlos. My collaboration with Dorit was by far the most productive and stimulating period of my research career. Without her example, this work would never have been what it is.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

Chapter 1

Introduction

The challenge of exploiting the large and fast-growing number of transistors available on modern chips has motivated the exploration of parallel architectures. Researchers have considered parallelism in two main categories. The first is at an ultra fine-grained level, among instruction sequences. However, the limited amount of instruction-level parallelism (ILP) present in programs, and the difficulties to uncover it [31] [59] prevent computers from fully exploiting the available on-chip resources. An alternative source of parallelism is at the level of coarse-grained multithreading, and has been traditionally expressed under the shared memory or the message-passing programming models, either directly by programmers, or with the aid of parallelizing compilers.

While some new architectures ([22] [53]) have the ability to exploit fine-grained parallelism, they report difficulties obtaining fine-grained multi-threaded codes for today's important uniprocessor programs [23]. On one hand, parallelizing compilers have been only partially successful at automatically converting serial codes. On the other hand, the programming models currently available to programmers were originally designed to target multi-chip parallel architectures, and therefore do not fit a single-chip environment.

XMT (explicit multi-threading) attempts to bridge the gap between the ultra fine-grained on-chip parallelism and the coarse-grained multi-threaded program by proposing a framework that encompasses both programming methodology and hardware design. XMT tries to increase available ILP using the rich knowledge base of parallel algorithms. Relying on parallel algorithms, rather than on a compiler, programmers express extremely fine-grained parallelism at the thread level.

Early papers on XMT have discussed in detail its fine-grained single program multiple data (SPMD) multi-threaded programming model, architectural support for concurrently executing multiple contexts on-chip, and preliminary evaluation of several parallel algorithms using hand-coded assembly programs [56] [13]. The introduction of an XMT compiler, allowed us to evaluate XMT for the first time as a complete environment ("vertical prototyping"), using a much larger benchmark suite (with longer codes) than before. The evaluation of the XMT compiler and evaluation of the programming model were published in two specialized workshops, [39] and [40] respectively. The followup [41] presented the integrated results, as well as the interplay between the programming model and the other components of XMT (compiler and architecture).

XMT as a framework developed starting from the area of parallel algorithms. While the goals are ambitious, every effort has been made to make the means as modest as possible. Those novel elements which are essential, including the spawn-join construction and the no-busy-wait ideal, are layered on top of standard computing concepts in the least disruptive manner possible.

In keeping with the thrust of the framework as a whole, the approach to XMT hardware design has been largely opportunistic: recognizing the changing constraints of system design with new technology. Wherever possible, the architectural philosophy of XMT has been to rely on proven elements. This allows XMT design to benefit from general technological progress in the industry. However, to enable the framework to work effectively, XMT has been forced to distinguish itself in two hardware architectural aspects. The first piece, the hardware prefix-sum mechanism, provides for the low-overhead coordination of many parallel threads. This was first introduced in [55] in 1997. The second piece is an effective on-chip parallel memory subsystem. This second enabling technology is presented in detail for the first time in this thesis.

We begin by briefly reviewing the XMT multi-threaded programming model, architecture, and compiler technology in Chapter 2. We then step back to consider the characteristics of memory subsystem design from an XMT perspective in Chapter 3. Chapter 4 serves to present the essential points of the proposed memory architecture. This is followed by Chapter 5, which considers issues in implementing the memory as part of the system as a whole. Chapter 6 evaluates the low-level characteristics of a practical design in deep submicron technology, including area and timing details. Finally, Chapter 7 concludes.

The reader may wish to consider that this thesis documents research largely finished by late 2000, but presented for submission at this time.

Chapter 2

Overview of the XMT Framework

Most of the programming effort involved in traditional parallel programming (domain partitioning, load balancing) is of lesser importance for exploiting on-chip parallelism, where parallelism overhead can be made low and memory bandwidth can be made high. This observation motivated the development of the XMT programming model. XMT is intended to provide a parallel programming model, which is (1) simpler to use, yet (2) efficiently exploits on-chip parallelism.

These two goals are achieved by a number of design elements; The XMT architecture attempts to take advantage of the faster on-chip communication times to provide more-uniform memory access latencies. In addition, a specialized hardware primitive (prefix-sum) exploits the high on-chip communication bandwidth to provide low overhead thread creation. These low overheads allow the efficient support of fine-grained parallelism. Fine granularity is in turn used to hide memory latencies, which, in addition to the more uniform memory accesses, supports a programming model where locality is less of an issue. The XMT hardware also supports dynamic load balancing, relieving the programmers of the task of assigning work to processors. The programming model is simplified further by letting threads always run to completion without synchronization (no busy-waits), and synchronizing accesses to shared data with a prefix-sum instruction. All these features enable efficient support for simple, algorithm-centric spawn-join semantics. The end result is a flexible programming style, which encourages the development of new algorithms, and is expected to target a wider range of applications.

2.1 XMT Programming Model

The programming model underlying the XMT framework is an arbitrary CRCW (concurrent read concurrent write) SPMD (single program multiple data) programming model. In the XMT programming model, an arbitrary number of virtual threads, initiated by a *spawn* and terminated by a *join*, share the same code [58]. The arbitrary CRCW aspect dictates that concurrent writes to the same memory location result in an arbitrary one committing. No assumption needs to be made beforehand about which will succeed. This permits each thread to progress at its own speed from its initiating *spawn* to its terminating *join*, without ever having to wait for other threads; that is, no thread busy-waits for another thread. The implied "independence of order semantics" (IOS) allows XMT to have a shared memory with a relatively weak coherence model. An advantage of using this easier to implement SPMD model is that it is also an extension of the classical PRAM model, for which a vast body of parallel algorithms is available in the literature. (Previous XMT papers related the relaxation in the synchrony of PRAM algorithms to works such as [10] and [20] on asynchronous PRAMs).

The programming model also incorporates the *prefix-sum* statement. The *prefix-sum* operates on a base variable, B, and an increment variable, R. The result of a *prefix-sum* (similar to an atomic fetch-and-increment) is that B gets the value $B + R$, while the return value is the initial value of B. The primitive is especially useful when several threads simultaneously perform a prefix-sum against a common base, because multiple prefix-sum operations can be combined by the hardware to form a multi-operand prefix-sum operation. Because each prefix-sum is atomic, each thread will receive a different return value. This way, the parallel prefix-sum command can be used for implementing efficient and scalable interthread synchronization, by arbitrating an ordering between the threads.

The XMT-C high-level language is an extension of standard C. The extensions are described individually in Appendix A. A parallel region is delineated by spawn and join statements. Synchronization is achieved through the prefix-sum and join commands. Every thread executing the parallel code is assigned a unique thread ID, designated TID. The spawn statement takes as arguments the number of threads to spawn and the ID of the first thread. Consider the following example of a small XMT-C program. Suppose we have an array of n integers, A, and wish to "compact" the array by copying all non-zero values to another array, B, in an arbitrary order. The code below spawns a thread for each element in A. If its element is non-zero, a thread performs a prefix-sum (*ps* in XMT-C) to get a unique index into B where it can place its value:

```
m = 0;
spawn(n,0);
  {
        int TID;

        if (A[TID] != 0) {
        int k;
        k = ps(&m,1);
        B[k] = A[TID];
        }
  }
join();
```

The SpawnMT model of [56] does not support nested initiation of an arbitrary-size spawn within a parallel spawn region. Such a feature, while useful, would be difficult to realize efficiently with hardware support. As an alternative, Vishkin [57] extended the programming model to support a fork operation. A thread can perform a fork operation to introduce a new virtual thread as work is discovered. Forks must be executed one at a time by a single thread, but forks from multiple threads can be performed in parallel. The fork extension allows the programmer to approach many problems in a more asynchronous and dynamic manner. In XMT-C, *fspawn* is used when forking may be necessary, in lieu of a nested spawn. Within an *fspawn*, an *xfork* indicates a fork operation.

The attempt to develop general-purpose parallel algorithms that map well to different parallel machines, has led researchers to utilize different parallel models. In [45] the QSM model is used to design and analyze general-purpose parallel algorithms, along with a suitable cost metric (which is shown to be accurate for big problem sizes). The algorithms they present are adaptations of PRAM algorithms and their mapping to other parallel models is discussed. XMT is also motivated by the attempt to provide a way to map general-purpose parallel algorithms to a real parallel machine. The XMT architecture is designed to provide such a prototype target machine for which it is easy to map PRAM algorithms, using the XMT programming model.

MIT's Cilk [19] also provides a multi-threaded programming interface and execution model, however, there are two important differences in scope. First, since Cilk is targeted at compatibility with existing symmetric multiprocessor (SMP) machines, load balancing in software was important. XMT provides hardware support to bind virtual threads to thread control units (TCUs) exactly as the TCUs become available. The low-overhead of XMT is designed to be applicable to a much broader range of applications. Second, Cilk presents a programming model that tries to match very closely standard serial programming constructs, where forking a thread takes the form of a function call. While XMT also bases its programming model on standard C, the programmer is expected to rethink the way parallelism is expressed. The wide-spawn capabilities and prefix-sum primitive are present to support the many algorithms targeted to the PRAM model. One has to keep in mind that XMT is PRAM-like programming, but not exactly PRAM programming.

2.2 The XMT Architecture

The most important distinguishing characteristics of an XMT architecture are low-overhead mechanisms for the management of parallelism. New elements not present in standard microprocessor design are introduced for the purpose of supporting the parallel programming model.

The XMT programming model allows programmers to specify an arbitrary degree of parallelism in their code. Clearly, real hardware has finite execution resources, so in general all threads cannot execute simultaneously. In an XMT machine, a TCU executes an individual virtual thread. Upon termination, the TCU performs a prefix-sum operation in order to receive a new thread ID. The TCU will then emulate the thread with that new ID. All TCUs repeat the process until all the virtual threads have been completed.

This is illustrated: (i) through a comparison with the von Neumann stored program and program counter apparatus (Figure 2.1), and (ii) through a snippet of the program of a TCU (Figure 2.2).

We begin with Figure 2.1. Its *upper part*, entitled “Von Neumann (1946–??),”

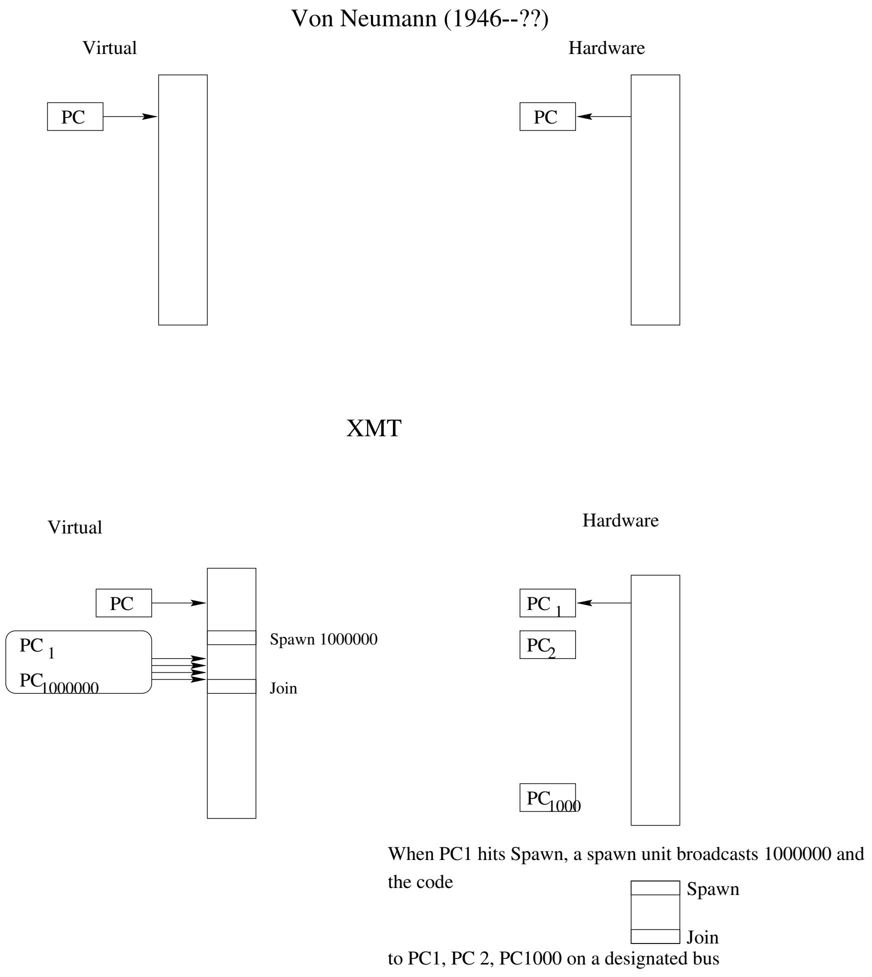


Figure 2.1: Program counter + stored program

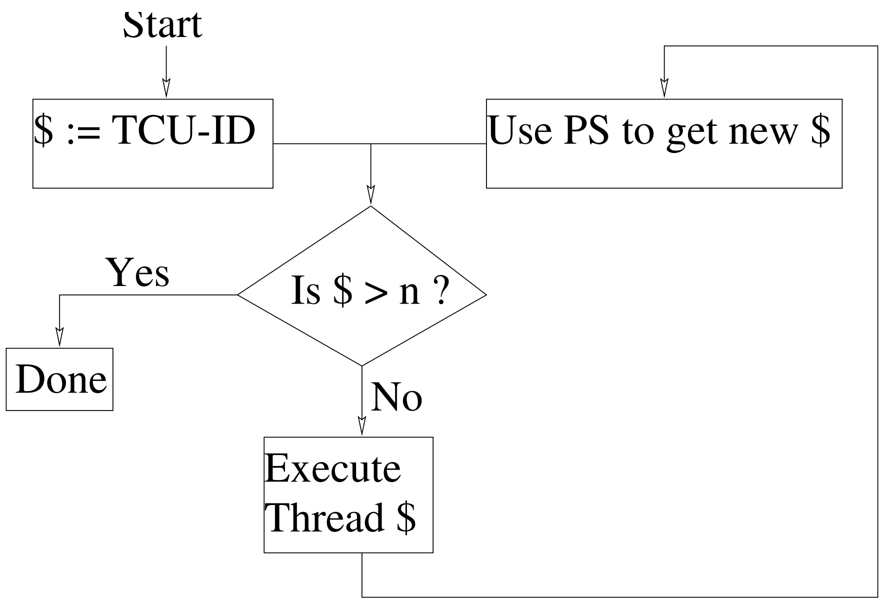


Figure 2.2: TCU program snippet

illustrates the program counter apparatus in serial machines, which has dominated general-purpose computing since 1946, with no end in sight to its reign. The *right-hand side* (of the upper part of Figure 2.1) depicts the hardware apparatus, where one command at a time is brought to the program counter. The *left-hand side* (of the upper part of Figure 2.1) demonstrates how the programmer is often educated to think about this apparatus — the virtual outlook. Here the program counter is the one to move; it moves from one location of the memory to another, perhaps like a book analogy, where the finger of a reader advances from one line of the book to another. The fact that this von Neumann apparatus has survived orders of magnitude improvements in speed since the 1940s makes it a remarkable "Darwinistic" success story. For this reason we sought to upgrade, rather than replace in a disruptive manner, this successful apparatus.

The *lower part* of Figure 2.1, entitled "XMT," illustrates the new apparatus.

The *left-hand side* (of the lower part of Figure 2.1) depicts the virtual description. There is still one computer program as in the von Neumann apparatus. In the above book analogy, finger (program counter) number 1 moves from one line of the book to another, until it reaches a special command called Spawn. The Spawn command specifies a number of "threads" which can be performed in parallel. Since we are concerned at the moment with the virtual side, any number of threads may be specified. Figure 2.1 diagrams 1,000,000 threads. The virtual threads, initiated by a Spawn and terminated by a Join, share the same code. At run-time, different threads may have different lengths, based on individual control flow decisions. The programmer's understanding will be that each of the threads can progress (guided by one finger per thread) from the Spawn command to a subsequent Join command at its own speed. At the Join, the thread expires. Once all the virtual threads expire, finger number 1 continues. The main difference in the hardware description on the *right-hand side* (of the lower part of Figure 2.1), is that the number of program counters is fixed (the figure mentions 1000), and does not change as a function of the Spawn command at hand. When program counter number 1 executes a Spawn command it broadcasts the instructions of a thread (i.e., until a Join command) to the other program counters. This broadcast is performed on a broadcast network, which due to the simple topology required could be made very efficient. The program counters start by executing the first 1000 among the 1,000,000 threads, one thread each. When a program counter completes its thread, it starts executing one of the yet-to-be-executed threads. This is done until all of the 1,000,000 threads finish.

Figure 2.2 illustrates the program of a TCU. Suppose that $n = 1{,}000{,}000$ threads are to be executed as a result of a Spawn command. The figure assumes that n, and the SPMD code, were broadcast to all TCUs. TCU i starts by executing the respective virtual thread i, but only if i is not larger than n. Upon finishing the execution of a virtual thread, the TCU uses a prefix-sum computation to obtain the ID of the next virtual thread it should execute, and proceeds to execute it if that ID is not larger than n. Note that the only communication among TCUs above was through the prefix-sum computation.

This functionality is enabled by support at the instruction set level. With this architecture, all TCUs independently execute a serial program. Each accepts the standard MIPS instructions, and possesses a standard set of MIPS-type registers locally. The expanded instruction set architecture (ISA) includes a set of specialized global registers, called prefix-sum registers (PR), and a few additional instructions. New instructions are used for thread management (Appendix A may be consulted for a glossary which includes separate descriptions of these new instructions). A *spawn* instruction activates all TCUs and broadcasts a new PC at which all TCUs will start executing. The *pinc* instruction operates on the PR registers, and performs a parallel prefix-sum with value 1. A specialized global prefix-sum unit can handle multiple *pinc*'s to the same PR register in parallel. Simultaneous *pinc*'s from different TCUs are grouped, and the prefix-sum is computed and broadcast back to the TCUs. A prefix-sum functional unit with 64 single-bit inputs is very similar to a 64-bit integer adder (please see Appendix D or [55] for additional details on this unit); in each cycle only the differential increments are calculated and sent back to the clusters. This process is pipelined and completes within a constant number of cycles. *pread* performs a parallel read (prefix-sum with value 0) of a (replicated) PR register, and *pset* is used (serially) to initialize a PR register. The *psm* instruction allows for communication and synchronization between threads. It performs a prefix-sum operation with an arbitrary increment to any location in memory. It is an atomic operation, but due to hardware limitations, operations to the same base address are not performed in parallel (i.e., concurrent *psm*'s will be queued). This is similar to a fetch-and-increment [18] primitive (see [2]). A *ps* command at the XMT-C level is translated to a *psm* instruction.

Additional instructions exist to support the nested forking mechanism. A new thread to be forked requires some form of initialization. This initialization can be performed by the forking thread with the aid of *psalloc* and *pscommit*. The *psalloc* instruction works like a *pinc*, but the increment to the PR register is not visible to anyone else until the forking thread performs the corresponding *pscommit*. This allows the forking thread to initialize data (usually just a pointer) before a forked thread starts. Note that like the *pinc* instruction, *psalloc*/*pscommit* from many TCUs can be performed in parallel batches. The last new instruction, *suspend*, is also used when forking may occur. An idle TCU can suspend, waiting for its assigned thread ID to become valid, without consuming any execution resources.

The fundamental hardware units of execution for the machine are these multiple TCUs, each of which contains a separate execution context. In implementation, an individual TCU basically consists of the fetch and decode stages of a simple pipelined processor. To increase resource utilization and to hide latencies, sets of TCUs can be grouped together to form a cluster, each cluster quite similar in spirit to an SMT processor. The TCUs in a cluster share a common pool of functional units, as well as memory access and prefix-sum access resources. The clusters can be replicated repeatedly on a given chip.

More detail about XMT architecture prior to the work of this thesis can be found in Appendix B.

2.3 The XMT Compiler

A strength of the XMT model is that its performance potential does not hinge on the development of sophisticated compiler analysis and transformation techniques. The results of the paper were obtained with a prototype compiler which performs only relatively straightforward transformations of the programmer's code.

Parallel execution in the XMT architecture requires handling:

1. transition to parallel mode — activate all the TCUs and set up their environment;

2. thread creation and termination — emulate the virtual threads on each TCU, obtain a thread ID for each, and verify that it is a valid ID (i.e., less than the spawn size);

3. transition back to serial mode — detect when all threads have terminated, and resume serial execution.

In first presentations of XMT, these tasks were handled entirely by hardware automatons. Later work introduced a scheme whereby the preceding tasks are orchestrated by compiler. This choice pays off in performance and flexibility. For example, the compiler is free to schedule certain operations to have a per-TCU cost rather than a per-thread cost. Additionally, the more general hardware allows for various extensions, such as different forking schemes, and can easily support parallelization models other than XMT.

Original Xmt-C program

```
main() {
      spawn(num_threads, offset);
      {
            int TID;

            **THREAD-CODE**
      }
      join();
}
```

Transformed to

```
main() {
      spawn_setup(num_threads, offset);
      main_0_spawn();
}

main_0_spawn () {
      int TID, maxtid, offset;
      spawn_init(&max_tid, &offset);
      TID = TCUID + offset;
      while (TID < max_tid) {

            **THREAD-CODE**

            TID = get_new_tid();
      };
      tcu_halt_suspend();
}
```

Figure 2.3: XMT code shape

The prototype XMT compiler consists of two phases:

1. The front end ("Xpass") — a source-to-source translator based on SUIF [60]. This phase converts the XMT code with its parallel constructs into regular C code with specialized assembly templates for run-time threading support.

2. The back end (gcc) — builds an executable file from the C code produced by Xpass. As we based our simulator implementation on the SimpleScalar ISA, we used the version of gcc from the SimpleScalar 2.0 package (gcc 2.6.3).

The general scheme used by Xpass is based on transforming parallel codes into parallel procedures. The compiler transforms the parallel region (the code in the spawn-join block) into the body of the procedure. When the procedure is called, the processing units are awakened, and each starts to execute the procedure body, which emulates the threads on each TCU. Figure 2.3 presents a high level example of the transformations performed by our compiler.

Producing this structure involves two tasks:

1. Outlining. Detect all parallel regions (spawn-join blocks) and create a function definition for each (a "spawn-function"). Replace the spawn-join block with a call to the spawn-function.

2. Spawn-function transformation. Add TCU initialization code and thread emulation constructs to the spawn-function. These constructs include wrapping the body of the spawn-join block with a loop to emulate the threads, and inserting assembly templates.

Chapter 3

A Discussion of Memory from an XMT Perspective

Computer designers are recognizing that memory system performance is increasingly becoming critical to efficient execution [61]. At first glance, memory issues might appear to be the Achilles heel of the XMT framework. Parallel algorithms commonly require more memory operations than their serial counterparts. Preliminary studies have found XMT programs to be no exception [4].

However, the XMT model of computation provides an opportunity to break through the memory-bound limitations of serial computation and make more efficient use of memory. We will consider how to revisit memory subsystem design for this very different computational model, within the constraints of future chip technology.

Before proposing any specific solutions, we will first lay the groundwork by explicating the unique challenges and opportunities facing the XMT memory system designer. In this chapter, we will describe the primary design goal that will guide design, we will review existing technology as it might apply to an XMT system, we will define the constraints placed on the designer by the XMT consistency model, we will highlight the different assumptions relevant to the task of instruction fetching, and finally we will discuss the requirements of hardware prefix-sum operation support.

3.1 Statement of the Design Goal

It is important to define exactly what we wish to achieve with a memory design. A general-purpose computer system must be capable of handling efficiently a wide variety of cases, including different: patterns of memory access, granularity of data structures, and structures of parallelism.

To support the range of programs that may fall under the PRAM-like programming methodology of XMT, we must be particularly ambitious. Difficult programs may exhibit:

- Data access from many execution contexts. In the near future, we expect to support up to 1024 simultaneous thread contexts on a single-chip XMT machine.

- Very fine-grained threads. XMT's PRAM-style algorithmics often encourage minimal length threads. Additionally, these small threads may operate on just a single piece of data. Together, this implies that in many cases there may be little to gain from locality of reference (either spatial or temporal) within a single thread.

- Irregular, unpredictable access patterns. XMT attempts to broadly support all types of applications, including those with irregular modes of access, such as those operating on pointer based data structures. Furthermore, XMT hardware is designed to operate in a rather asynchronous fashion, rather than in a scheduled, lock-step fashion. As a result, we cannot rely on orderly or predictable access patterns of access from different threads.

In this paper, the following primary principle is followed: design a system capable of handling the most difficult common case, as described by the points above. Later, we will offer suggestions to improve performance further when more forgiving types of programs or workloads offer opportunities for optimization, or for handling less common cases, such as codes providing less concurrent parallelism than a machine is designed for.

3.2 Overview of Existing Technologies

To understand the necessity for an entirely new design effort, it is important to recognize that existing memory designs are inappropriate for an XMT system.

Nearly all popular microprocessors are designed to execute serial programs. In serial computing, parallelism is limited to that which can be extracted from a single thread of execution. The key to efficient computation is to complete a memory request as soon as possible after it is issued. Not surprisingly, memory architecture design for serial machines has tended to emphasize reduction of latency as the primary design goal. Two important characteristics evident in the access patters of many serial programs are spatial locality and temporal locality. On-chip caches have been the primary means of leveraging these two properties to reduce latency of access.

With the fine-grained parallelism for which XMT is designed, a single thread may not exhibit the properties of spatial and temporal locality of reference to the same degree. However, the simultaneous existence of many threads can mean that latency of memory access within individual threads does not have to throttle performance. If many simultaneous memory accesses can be supported, than progress can be made while some threads are waiting for memory results. While successful at reducing average access latency for many serial programs, traditional memory hierarchies do not provide the necessary bandwidth capacity to support an XMT system, nor do they support the level of overlap necessary to effectively hide latencies.

Memory subsystems developed for massively parallel machines also are a poor fit for an XMT environment. Multi-chip implementations impose the constraints of very high latency and limited bandwidth between nodes, compared to a single-chip implementation. Such systems have proven effective only for very coarse-grained sharing. Vector architectures, (eg. [44]), demand lock-step timing and highly regular data access. We must look elsewhere for broader application support.

More recently, chip multiprocessors (CMPs) have begun to appear [22]. These systems have so far been essentially single chip implementations of traditional symmetric multiprocessors (SMP). Such systems use CPU-local caches, perhaps backed by a shared cache. The designs have been tuned to running multiple independent programs (no-sharing), or to very coarse-grained threading. Like those in SMP systems, the local caches use directory or update-based coherence schemes, and are connected by a bus or low-order crossbar. These traditional interconnects and coherence schemes would not scale to the number of processor resources targeted for an XMT system, nor would they be appropriate for the fine-grained sharing common in PRAM-style algorithms.

Current research designs will not provide satisfactory solutions from an XMT perspective. MIT's RAW project also aims to harness on-chip parallelism, but takes a very different philosophical approach to computer system design. Processors are paired with small, low-latency local memories into a unit called a tile. Tiles are placed in an array fashion. The connection network is a grid topology, where each tile can talk with its four neighbors. DRAMs can be connected at the edges of the array of tiles. The RAW architecture exposes to software the details of data placement, data movement, and switch scheduling. On RAW, the programmer (or a sophisticated compiler) orchestrates movement of data from DRAM to local RAM, and from tile to tile. The simple programmer model of a uniform, global memory space does not map well to this architecture.

Another approach is to target the memory system for streaming applications. The Imagine project takes this approach, optimizing for compiler-scheduled streaming media applications. Such an architecture does not apply well to non-regular applications, especially where parallelism is not plentiful.

3.3 XMT Consistency Model

It is useful to carefully consider the shared memory consistency requirements of the XMT programming model. By making explicit what can be expected by the programmer, such considerations provide important constraints on memory system design, and also guide the compiler in performance-critical tasks like register allocation. In this section we present a practical discussion of memory consistency as it affects XMT system design.

3.3.1 Thread-Local View

XMT programming uses serial semantics as a starting point. When in serial mode, XMT follows serial ordered memory requests. Similarly, within a thread, serial consistency should be respected. This means that an individual thread should see its updates in the order they are issued.

3.3.2 Standard (Non-Fork) Spawn Regions

One of the most important aspects of the XMT programming model is the Independence-of-Order-Semantics (IOS). Recall that a spawn-join block declares a number of virtual threads. By convention, IOS implies that these virtual threads may execute in any order, in serial or in parallel. Because of the lack of order constraints, the implication is that memory consistency between different threads need not be maintained. The spawn-join construct demands only that memory be consistent at the start of a spawn operation (ie. that all spawn-region threads have access to updates performed by the serial-mode thread), and that memory again be consistent at the conclusion of a join operation (ie. the resumed serial-mode thread has access to all updates performed by the spawn-region threads).

Note that this does not imply that all virtual threads must start with memory in the pristine state that existed when the spawn began. Updates from arbitrary other threads may have changed the state of memory before any particular thread begins executing.

Note also that updates from a particular thread to *different* locations are not necessarily visible to other threads in the order issued. An XMT programmer may not reason about ordering between threads without utilizing a prefix-sum operation.

When spawn and join are the only mechanisms used for synchronization, memory consistency can either be maintained or be ignored for the duration of the spawn region. This property can give significant flexibility to a hardware designer. While an XMT program may specify very short virtual threads, in hardware a single TCU may emulate many virtual threads one after another. This means that, even for a very fine-grained spawn, the region of relaxed consistency may be quite significant.

The picture becomes more complex when the prefix-sum operation is considered. Recall that the XMT programming model draws much of its power from the use of prefix-sum operations within a spawn-region to provide no-busy-wait synchronization and dynamic ordering. While no static ordering is implied, interaction by way of prefix-sum allows for threads to determine an ordering of certain operations at run-time. This ordering may have implications for memory consistency assumptions.

Berkovich [4] terms a prefix-sum operation that forces inter-thread memory operations to be ordered a *gatekeeper prefix-sum.* Consider the following code fragment in which 2 threads compute the sum of a 4 element array A. The sum should be placed in the first element of A.

```
gatekeeper = 0;
spawn(2, 0);    /* spawn two threads, starting with ID 0 */
{
  int count;

  A[TID*2] = A[TID*2] + A[TID*2 + 1];    /* sum two elements */
  count = ps(&gatekeeper, 1);
  if (count == 1) {    /* 2nd of two to hit the gatekeeper ? */
    A[0] = A[0] + A[2];    /* compute total = sum 2 partials */
  }
}
join();
```

Each thread computes the sum of two unique elements of A, and overwrites an element with this value. After writing its first sum, each thread performs a prefix-sum with increment 1 to the gate-keeper variable. The first to arrive gets value 0 and is done. The second to arrive gets value 1. This second thread now knows that the first thread has already performed the prefix-sum, and hence must have already written the partial sum to the array. Therefore, the second thread can read the value written by the other thread, and compute the total sum correctly. The gatekeeper prefix-sum operation provides the synchronization which allows the thread to assume memory consistency at that point.

Note that usage of prefix-sum in a spawn block does not necessarily imply that such an ordering of memory operations must be enforced. Take the common example of array compaction. Here, we copy the non-zero elements of array A to array B.

```
b_size = 0;
spawn(a_size, 0);
{
  int index;

  if (A[TID] != 0) {
    index = ps(&b_size, 1);
    B[index] = A[TID];
  }
}
join();
```

The prefix-sum is used to provide each thread that wishes to write to array B with a unique index. Despite this ordering (implied by the assignment of indexes), there is no implied dependencies between the memory operations of different threads.

If a gatekeeper prefix-sum is one around which we must enforce memory consistency, it is useful to be able to determine what is or isn't a gatekeeper. Berkovich suggests the following compile-time method for identifying gatekeeper prefix-sums. A prefix-sum is treated as a gatekeeper prefix-sum if the thread has a control dependency on the result of the prefix-sum. Note that the first example above (the toy summation) fits this rule, while the second example (array compaction) does not. The intuition is that if the threads don't modify their behavior as a result of the run-time ordering, then they aren't subject to anything but the static assumptions.

While this rule was never intended as a sufficient condition for identifying the gatekeeper property, it is important to recognize that it is not a necessary condition either. Consider the following reimplementation of the first example, where the control dependence has been transformed into a data dependence. After the first ps operation, local variable *count* contains either the value 0 or the value 1. After the arithmetic inversion on the next line, it possesses the value 0 or -1. Since (for a two's-complement system) -1 is represented in binary as all 1's, we can use count as a bit-wise and-mask to conditionally zero the partial sum value. The second (non-conditional) ps is therefore a noop for the zero-value case. This second ps does not return a result, and only serves to allow both threads to perform an addition without interfering with each other.

```
gatekeeper = 0;
spawn(2, 0);    /* spawn two threads, starting with ID 0 */
{
  int count, masked;
  A[TID*2] = A[TID*2] + A[TID*2 + 1];    /* sum two elements */
  count = ps(&gatekeeper, 1);
  count = -count;    /* -1 is all 1's in two's complement */
  masked = A[2] & count;    /* mask value to zero for 1st thread */
  ps(&A[0], masked);    /* noop for 1st thread, sum for 2nd */
}
join();
```

While this example is entirely contrived, the programmer should be able to expect it to work correctly.

Gatekeeper prefix-sums appear to be most useful in a class of algorithms which we will describe as asynchronous. Examples of this class of algorithm are described in [57]. A reduced-synchrony algorithm often has a more synchronous alternative. The balanced tree paradigm of [57], can be implemented with a single spawn-join region that uses gatekeeper prefix-sum operations to synchronize without busy-waiting. Alternately, a spawn-join region can be executed for each level of the tree, with no memory ordering within each spawn, only between spawns. We term this style of algorithm *level synchronous*, since the spawn-joins serve as a synchronization barrier between each level of the tree. While the level-synchronous algorithm may be easier to support from a memory designer's perspective, the asynchronous class of algorithms are useful and must be supported efficiently as well.

From the hardware designer's perspective, we will leave the identification of gatekeeper operations as a task for compiler to determine. This means the hardware should provide support for both gatekeeper and non-gatekeeper variants of the prefix-sum instructions. From the compiler's perspective, it is always safe to treat user-specified prefix-sums as gatekeepers. It may be useful to provide a programmer-visible non-gatekeeper option until compiler technology exists to provide useful discrimination.

We make the following observation, which has strong implications in practice. In a standard spawn-region, the prefix-sum operation used by the threading runtime to generate new thread IDs is *not* a gatekeeper. This very common case is not programmer-visible, and does not require compiler techniques to identify.

3.3.3 Forking Spawn Regions

Within a region that may allow forking of threads (fspawn-join), additional considerations apply. When a thread performs an xfork() or the more useful xfork_and_init(), it implies an order between the fork operation and the start of the new thread that is instantiated. Therefore all updates performed by the forking thread up to the point of forking should be visible to the new thread.

The practical importance of this property is that xfork is used when new work is discovered or created. Beyond the memory consistency requirements implied by the xfork, we often must provide the new thread with some sort of pointer to the new work. The issues related to this are discussed in section 5.5.

3.4 Instruction Memory

The memory design requirements for supporting instruction fetch are entirely different then those considered for (data) load/store support. There are three key points that differentiate instruction memory in an XMT system.

1. Instruction memory access is read-only.

   As is typically done in standard computer design, we can consider the text segment of a program to remain unchanged throughout the life of the program. This property has a non-trivial implication for a parallel machine, since all fetch units will always have a coherent view of the instruction memory space, and memory consistency is a non-issue.

2. A thread's instruction access patterns exhibit significant locality and predictability of reference.

The properties of temporal and spatial locality present in serial code have enabled the effective use of instruction caches in microprocessors. These properties are no less evident within an XMT thread. In fact, the manner in which multiple virtual threads are emulated on a single TCU, reinforces these characteristics.

3. The SPMD programming model suggests parallel threads will usually be accessing a common range of instruction locations.

   We note that supporting instruction fetch bandwidth for an XMT system is a substantially different problem than supporting data access, and should not be significantly more demanding overall than for serial computing.

3.5 Prefix-Sum Support

Support for the operation of the prefix-sum primitive is another task of the XMT system designer. As described in section 2.2, the XMT ISA supports two distinct kinds of prefix-sum operations. Recall that with the parallel prefix-sum-to-register operation, the increment value is limited to single bit, and the choice of base variable is limited to a small set of special registers. On the other hand, the prefix-sum-to-memory instruction allows an arbitrary value increment, and the base variable can be any location in memory.

The prefix-sum-to-register operation is integral to XMT's dynamic thread scheduling. It also is useful in other contexts such as a parallel implementation of the partition step of the quicksort algorithm [40]. In these contexts, it is important for a wide-parallel XMT system to perform many such operations in parallel. The calling restrictions of the prefix-sum-to-register operation are designed to allow a special gadget (see Appendix D) to provide this capability. Implementation details will be discussed in section 5.5.

Implementation of the prefix-sum-to-memory operation likely requires making tradeoffs in performance. To guide these design choices, we make some observations about how this primitive is applied. The key observation is that in many applications, psm may be used in parallel by many different threads, but the number of collisions (that is, simultaneous operations on the same base), is often very limited.

A prime example of this phenomenon is seen in algorithms that follow the balanced tree paradigm of [57]. Here, each gatekeeper is the base variable of exactly two prefix-sum operations. There may be very many prefix-sum operations executed by different threads in parallel, but no more than two will be relative to the same base.

Other examples tend to exhibit similar behavior. Often, in problems involving directed acyclic graphs (DAG) [4][40], a counter exists at each node. Threads perform a prefix-sum operation to this counter to indicate a visit from a particular incoming edge. In this way, a thread can determine if it is the last to visit a particular node. As threads traverse the DAG simultaneously, many threads may perform prefix-sums in parallel. However if nodes have a limited in-degree, the number of collisions at any particular base will be limited.

As yet another example, consider the following hypothetical binning (bin-sort like) operation. A large array of data elements exist. Each data element will be assigned to one of many possible bins based on its value. A counter exists for each bin. A thread is spawned for each array element, and each thread increments (by means of a prefix-sum) the counter of the bin to which its element belongs. If the input array values are sufficiently distributed, we expect no particular counter to have to support many simultaneous increments.

Certainly, an array could be encountered that exhibits bad behavior from this perspective. An array of elements that all belong to the same bin is an example. If we did not know ahead of time that this would be the structure, we could not know to use a single counter with our parallel prefix-sum-to-register primitive. With the prefix-sum-to-memory primitive, the expected collisions would occur.

Similarly, a DAG traversal algorithm could be applied to an edge-heavy graph with few nodes and very large in-degrees. With such a pathological input, we again may encounter large numbers of collisions.

To support these kind of situations, the ideal prefix-sum-to-memory implementation would support many simultaneous operations to common or different bases. However, it would appear that support only for different bases in parallel would cover the majority of common cases.

Chapter 4

New Memory Subsystem Architecture

This chapter introduces a new memory system design for an XMT system. We present a non-standard on-chip memory organization, and an interconnection network tuned for deep-submicron VLSI implementation. For clarity, this chapter focuses only on the data memory subsystem, and only describes the essential elements of the design. This scheme forms the backbone of an effective memory design. Chapter 5 will follow up with implementation details, optimizations, and other aspects of the design.

Certain mechanisms described in this chapter are covered under patent application [43].

4.1 Memory Structure

The copious transistor budgets enabled by future chip technology will allow for enough on-chip RAM to hold rather large working sets. Additionally, connectivity within a chip has the potential to provide as much communications bandwidth as needed. The PRAM-like model presented by XMT demands that the memory subsystem support the simultaneous access by many threads to arbitrary locations in a shared memory space. It is clear that a large, monolithic cache could never support the demands of such a system. Also, since memory speed drops as size and number of ports increases, it pays to use smaller independent modules.

The goal is to parallelize the memory. In a system with many independent memory modules, many requests to different modules can be serviced at the same time. We will have many processing elements generating memory requests, and many independent memory modules to service the requests. By instantiating enough small cache modules that can all be accessed simultaneously, one can allow for sufficient bandwidth to satisfy many threads at once.

However, to match the shared memory orientation of the programming model, the memory subsystem must present a coherent global memory space. For a system with a small degree of memory parallelism, many solutions could be effective. But maintaining coherence among potentially hundreds of modules appears to be too costly to implement effectively.

Any sort of snooping scheme would incur a huge bandwidth cost. It would require that all writes be broadcast in some manner to all cache modules that might house that address. The total write bandwidth then is throttled by the bandwidth of the broadcast mechanism. This occurs in the scheme from Appendix B, where writes are throttled by the capacity of the ring interconnect.

With a directory-based cache coherence scheme, the centralized directory would quickly become the limiting factor in data access. Also, with fine-grained data access supported by XMT, directory management of small data lines could be prohibitively expensive.

To sidestep the coherence issues, the memory address space may be partitioned among the memory modules, such that a given address can reside at only

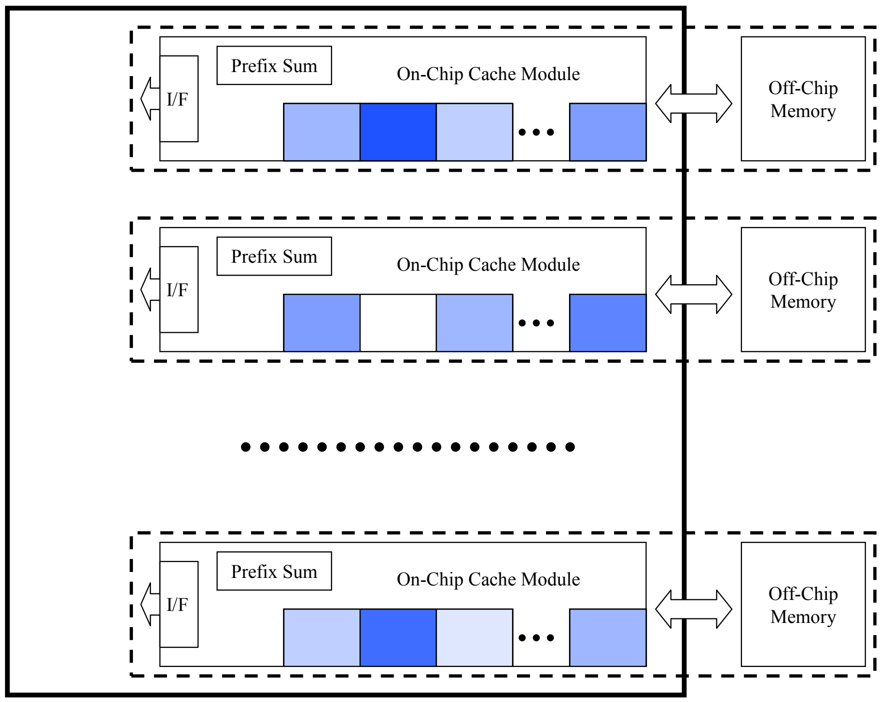


Figure 4.1: Parallel memory modules

one location (Figure 4.1). By precluding duplicate values from being located at different modules, we avoid the problem of multiple update or of maintaining a directory of value locations. By sacrificing mobility of address locations, we are forced to deal with two related issues.

- There is a mapping from memory addresses to specific memory modules. Each memory module has limited capacity to service requests. Therefore when different memory addresses are accessed simultaneously by different threads, we'd wish them to map to different memory modules. One solution might be to rely on software to map data structures to memory modules to avoid collisions. This solution is inadequate for the breadth of applications that might be encountered. Irregular, unpredictable applications can not rely on programmers or sophisticated compilers to distribute memory accesses.

  Borrowing a concept from parallel computing, we support randomized hashing to distribute addresses among the memory modules [37]. By using a random or pseudo-random hash to map locations word-by-word, we can probabilistically bound the number of collisions, without relying on any property of data access. Hashing strategies will be discussed in Chapter 5.

- A hashing scheme does not solve one class of collisions: when many threads simultaneously attempt to access the *same address*, contention will occur. This situation will be considered a special case, and approaches to handling it will be presented in Section 5.3.

The on-chip memory modules can be considered to be caches containing a subset of the data held in off-chip memory. One can make effective use of off-chip memory by extending the address partitioning to independent off-chip memory modules that serve the on-chip caches. By supporting many outstanding external requests, off-chip bandwidth can be fully utilized, even for non-streaming types of applications.

The modular memory design can be naturally extended to support multiple simultaneous prefix-sum-to-memory (psm) operations. psm operations can map to memory modules in the same way as a read or write. By incorporating a single adder into each memory module, the atomic addition can be implemented serially within each module. Such an implementation supports parallel psm to addresses at different modules, but multiple psm operations to the same address must be queued.

Recall that support for many simultaneous accesses to a single global register (pinc) is implemented using a separate interconnect which incorporates parallel prefix-sum hardware. Details will be provided in section 5.5.

4.2 Interconnect Design

An XMT system inherently implies many execution units, and we propose to implement many independent memory modules. A substantial challenge for an XMT design is to provide connectivity between the many execution units and the many memory modules on-chip.

Due to the flat memory model supported, memory requests from a given thread can travel to any memory location on the chip. A latency cost for such memory accesses cannot be avoided. While the capacity for sending signals increases with each technology shrink, the latency for propagating signals down a long wire is increasing. Fortunately, the parallelism and independence of order characteristic of XMT threading allows for such latency to be tolerated. While many threads stall waiting for memory requests to propagate through the network, forward progress can be made by other threads, provided enough thread contexts are instantiated simultaneously. Our solution will therefore focus on providing maximum useful bandwidth to the memory subsystem.

4.2.1 Crossbar Interconnect

Investigating the Synchronous Crossbar

The interconnection network must allow as many simultaneous connections between processing elements and memory elements as possible. It is reasonable to restrict the capacity of the endpoints to a single connection at a time. However, at any given time, a processing element may need to connect to any single memory element, and a memory element may need to connect to any single processing element.

The classic solution for such an interconnection network is the crossbar switch. While it has the potential to provide the requisite connectivity, the crossbar has yet to be proven as a large-scale on-chip interconnect. In particular, the area cost of a crossbar is known to be $O(n^2)$, where n is the number of ports. However, our investigations indicate that a full crossbar topology will be a feasible structure for the on-chip interconnect for XMT systems based on near-future VLSI technology. Please refer to Appendix C for a report of this work.

To use such a topology in a system, there must be some method for scheduling the switch settings so ports can communicate without stomping on each other. Such a method should be scalable, decentralized, and efficient. Appendix C also describes solutions for scheduling a synchronous crossbar switch.

Issues

The above-mentioned work demonstrates the feasibility (from an area cost standpoint) of a crossbar structure. However, several factors make the synchronous crossbar a less than ideal solution. The first issue is that orchestrating unpredictable access among the various (processing elements and memory modules) ports adds unnecessary overhead. While Appendix C introduces several methods to handle this scheduling problem, all solutions require some sort of global coordination to set a particular configuration. Ideally, messages would freely access their destinations, with contention dealt with locally only where it occurs.

Other limitations of the synchronous crossbar approach are dictated by deep-submicron VLSI realities. It is well documented that wire delay will become an increasingly important factor in future chip design. As process dimensions shrink, transistor delay goes down while wire resistance goes up. To reach high operating frequencies requires a design to use short, low-capacitance wires. A crossbar design with high connectivity is a large structure. Modulating signals on a long wire across the structure at speeds near that at which the active devices are capable becomes infeasible.

Furthermore, clock distribution across a large chip also becomes more difficult. At high frequency, stable clock distribution is an increasingly difficult design problem. The ITRS [47] projects a growing disparity between clock speeds that can be achieved locally and those that can be achieved across a chip. Additionally, high performance clocking trees with large drivers consume a large amount of power. In contemporary Alpha processor designs, the clock tree consumes 50% of the processor power [38].

4.2.2 Novel Solution

Certain characteristics of XMT provide an opportunity for a novel technological solution. The XMT high-level programming language is a multi-threaded model that attempts to mimic no-busy-wait finite state machines; that is where no (software) thread ever needs to suspend its progress, while waiting for another thread. Execution then consists of a plurality of (software) threads that follow Independence of Order Semantics (IOS); that is, different speeds of execution among threads including some different orders of executions of reads and writes to a memory location can be tolerated. Thus, a trust of XMT is that such threads need to synchronize relatively infrequently, which provides an opportunity for

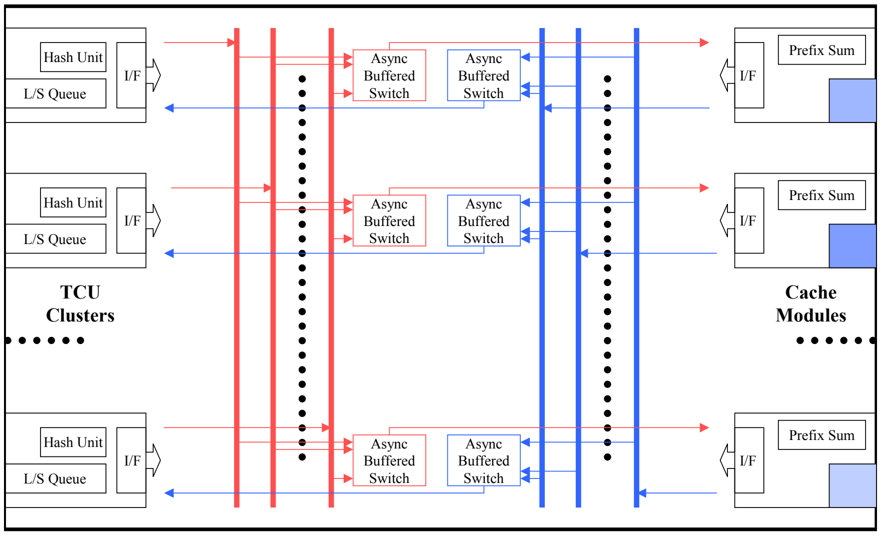


Figure 4.2: Memory system interconnect

reduced-synchrony hardware implementation.

We propose a decentralized routing scheme (Figure 4.2). The basic topology of a mux-tree crossbar can be maintained, with routing at each destination performed locally by a "tree of gatekeepers". This tree of gatekeepers is formed from locally synchronous switch points that select from two inputs depending on which arrives first. The gatekeepers structure mimics the multiplexor tree of a synchronous crossbar, but allows requests to proceed locally as they are able, rather than using a global schedule.

Datapath Layout

As an illustration of the change in datapath, we provide two figures. Both diagrams illustrate the datapath layout of an interconnect between four ports A, B, C, D. Each port has a 2-bit input port (green) going into the interconnect and a 2-bit output port (blue) coming out of the interconnect. The red boxes are

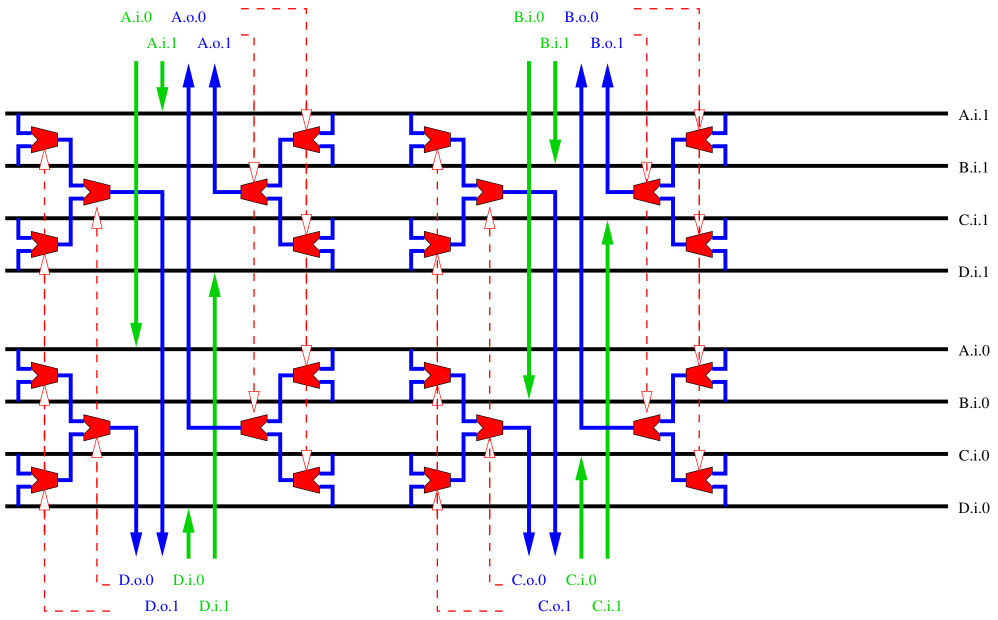


Figure 4.3: Traditional layout

multiplexors (muxs) where a decision is made to pass one input or the other.

Figure 4.3 represents the more conventional design. The layout can be described as a series of bit slices stacked on top of one another. In Figure 4.4, the 2 bits from a given port are “bundled” together and follow the same global route throughout. The dashed lines connect pairs of muxs that are coupled as a single logical 2-bit mux (mux-pair).

By abandoning scheduled switching in favor of a more dynamic scheme, we introduce a new requirement. A data bundle written into the interconnect potentially can go to any of the destinations (4 possibilities in the example). If the data is initially labeled with the intended destination, a leaf from each mux-tree can interpret the label to determine if it should enter that particular mux-tree. The label can be discarded at this point.

Similarly, when data is received at an output port, the source of the data

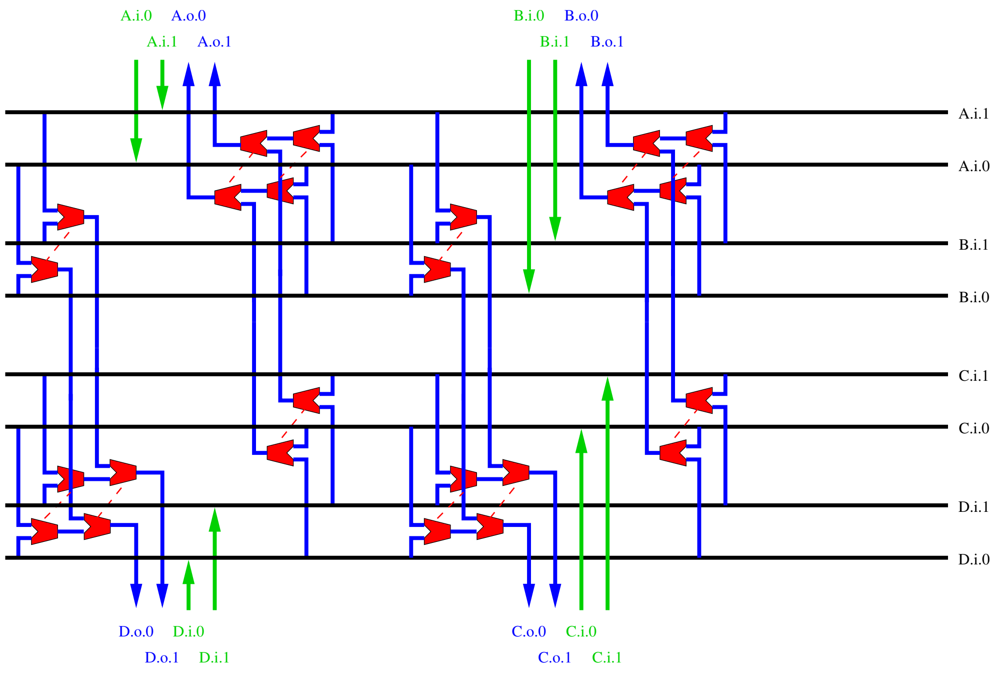


Figure 4.4: New layout

must be determined. Each mux-pair can encode the local decision made and pass this data on toward the root of the tree. The receiver can determine the sender from this information.

Reduced Synchrony Switching

This new configuration enables a scheme where switching decisions are made at the switch points. Messages should be allowed to progress through a switchpoint as soon as they arrive. To achieve this goal we turn to asynchronous control techniques. Each node of the mux-tree buffers a single data bundle. At each mux-pair, two data bundles compete to progress towards the root of the tree. An arbiter primitive [51] guards access to the buffer associated with the mux. The first of the two data bundles to arrive will win and be latched into the data buffer.

If a second bundle arrives before the first clears the mux, then the second will be granted next access. This switching automaton could be designed to incorporate alternation between inputs when both are continually attempting access. When applied globally, an alternation policy can prevent starvation.

Pipelining

For this design to be effective, it is important that many simultaneous requests be supported by pipelining throughout the interconnection network. Asynchronous design can also be applied to pipelining the wires in the form of micropipelining [51]. Long wires can be highly pipelined to form independently latched, self-timed short wire stages. Each bundle of wires (2-bit bundles in the example) requires 2 control wires to handshake between stages. Data can proceed from one stage to the next when the next stage has cleared.

4.2.3 Summary

Such a design can potentially overcome the limitations of the synchronous crossbar described earlier. The locally determined switching avoids global communication and coordination, while enabling efficient utilization of connectivity. The high degree of pipelining enables high bandwidth by allowing many data bundles in flight at once. Meanwhile, long wires are avoided to speed stage-to-stage transfers and to reduce driver size.

Furthermore, an asynchronous global interconnect allows for all processing elements to be independently clocked. This eliminates the need for a global clock tree and allows for processing clusters to be clocked as fast as can be achieved locally, without regard to global considerations.

In this chapter, we have presented a novel and potentially successful solution to the design problem. However, there is plenty of room for implementing some of these ideas in different ways. The idea of localized switching decisions can also be applied to a less asynchronous design. Instead of having an entirely unclocked interconnection network, it would be possible to partition the network into independently clocked domains. For example, each binary mux tree could be operated in a self-synchronous manner. Or, each mux tree could be reworked to operate locally in a reduced synchrony manner. The pipelines that feed the trees could be fully asynchronous, or divided into self-synchronous subdomains. A reduced synchrony design might have performance advantages over a fully asynchronous design, as the control and decision making logic is likely to be simplified. Further work is needed to determine which alternatives will perform the best.

Chapter 5

System Implementation

In this chapter we will discuss the proposed memory subsystem strategy in the context of an entire practical system. We will first discuss the structure and operation of the processing clusters, and then that of the memory modules. We will then propose potential techniques for reducing latency of access in certain cases. Finally, we will fill in the remaining holes by describing the implementation of instruction fetch memory and of parallel register operation.

This chapter will try to describe the practical design space, while Chapter 6 will focus on suggesting specific choices for a design starting point.

5.1 Processing Cluster Architecture

In previous expositions of XMT architecture, the concept of a cluster of TCUs was based on sharing of execution hardware. In this scheme, each cluster of TCUs was similar to an SMT processor, where multiple concurrent threads make efficient use of shared functional units.

When considering a large-scale XMT system as a whole, we realize that efficient use of execution resources may be a less important goal than efficient use of memory resources. While per-unit functional unit cost remains constant when scaling to higher hardware parallelism, the per-unit cost of global communication rapidly goes up.

From this perspective, a cluster should be based around a single interconnect port. The execution resources in a cluster can be considered to be generating memory requests so as to keep the memory network busy. When global communication can cost several cycles, we need to queue memory requests from many threads to keep the pipelined network busy.

By choosing the degree of clustering sufficiently high, the memory subsystem can be used efficiently during periods of plentiful parallelism. However, if the degree of clustering is so high that the TCUs within a cluster significantly impede each other, than load balancing issues are possible at the tail of a spawn, should one cluster be overtaxed and another idle. Clearly, a balance should be struck.

Of course, SMT style functional unit sharing can be implemented within such a memory-oriented cluster. The choice of SMT style clustering degree can be independent of the choice of memory-oriented clustering degree.

A processing cluster can consist of several mostly-independent processor pipelines. However, the load/store unit for all TCUs will be a shared structure. The purpose of this interface unit is to queue up requests from the TCUs, send them onto the network, and to return results to the correct TCUs.

The load/store unit must handle three kinds of memory requests. For all requests, the unit must determine the destination memory module of the address, and then send the address of the request to that memory module.

- In the case of a read operation, the request need only wait for the resulting data to arrive.

- In the case of a write operation, the updated value must also be sent in addition to the address. Although the write request does not require a result, the request will remain active until a response tag is received from the memory module. Without this tag, the TCU cannot know for sure when a write operation has been completed. While in most cases the TCU is free to proceed without knowing the status of a write operation, certain global operations may need to be delayed until pending writes are known to be visible to everyone. See Section 3.3 for a discussion of gateway prefix-sums.

- For a prefix-sum-to-memory operation, an increment value must be sent and a result must be waited on.

The concept of hashing comes to play at the load/store unit of the processing cluster. It would be possible to randomize every address such that it maps to random location in a random memory module. However, as there are a limited number of memory modules, we need not randomize every bit of an address. Say we have 2^n memory modules. If we use m bits of physical address to access a location in memory, we can partition an address into the upper $m-n$ bits and the lower n bits. The upper bits are sent to a particular memory module, and indicate the location within that memory module. The n lower bits indicate the memory module to which the request will be sent. However, this memory module index is randomly permuted for each physical location. In this way, addresses are evenly distributed in "stripes" to the memory modules. Note that this scheme preserves all the beneficial properties of full randomization, while also maintaining some degree of spatial locality within a memory module. Traditional concepts such as cache lines and paged memory can still be effective at the memory module level.

A natural place to implement the hashing of the lower bits is within the translation lookaside buffer (TLB) at the load-store module. Allowing the randomization seed to be modified per memory page (in the same manner as security bits), would open the door to various extensions, such as programmer visible (and selectable) hashing or variable granularity hashing.

5.2 Cache Module Design

In a manner similar to how we defined a processing cluster, we can distinguish a cache module as having a single port to the interconnection network. Pipelined requests enter the module one at a time, and results exit the module one at a time.

The module represents some portion of the physical address space of the processor. All requests for a specific address go through a single memory module, so the memory module serves as the single point of coordination between multiple threads for that address. Within a single cache module, we deal only with physical memory, so no concept of hashing is necessary here.

There is quite a bit of flexibility in how the memory can be organized. Likely, the module consists of an on-chip cache backed by external memory. Both the on-chip and off-chip portions might consist of multiple levels in a hierarchy of memory.

It may be worthwhile to implement some or all of the on-chip cache using DRAM technology. Since a sizeable latency penalty will already be incurred when traversing the chip, the slower access speed of on-chip DRAM memory may not hurt much. Rather, enabling larger working sets with the denser DRAM technology may significantly reduce off-chip memory access, which should have a much higher access penalty. If the memory cycle time for such technology is too slow to match the rate of the memory module, it may be useful to organize the RAM into sub-banks within the module. Alternatively, the number of independent memory modules could be increased relative to the number of processing clusters, to address a mismatch in throughput. Either organization would pay a cost of increased access latency and logic complexity, in order to fit the throughput goals of the system.

One can expect that the number of simultaneous requests presented to a memory module can be much greater than the number seen by the memory of a standard superscalar processor. Because of this, and because we already expect to pay a latency hit, it may be worthwhile to design significant intelligence into the request handling logic in a module. This should include the ability to reorder and combine requests. Since the on-chip cache modules are relatively small and independent, single cycle access for cache hits may be feasible. In such a case memory module design may be simplified, as support for overlapping requests would be more important for longer latency off-chip accesses.

To minimize latency of write operations, all write requests should be buffered, and an acknowledgement tag immediately sent back to the processing cluster.

The psm operation is implemented with an on-board adder. From the perspective of the issuer of a psm request, the adder doesn't add any latency to the operation, since the psm returns the *original* value. The adder is used only on the path to buffering for writeback to memory within the module.

Each on-chip cache module can have an independent off-chip memory assigned to it. The available off-chip I/O pins can be allotted to the memory modules, and transfers can be multiplexed on the assigned pins. System costs may dictate that fewer independent memories exist off chip, in which case on-chip memory modules would share off-chip resources. Clearly, off-chip bandwidth is limited by the number of I/O pins. Although the memory parallelism of an XMT design extends off chip, optimal performance can only be achieved for applications that match well to the amount of on-chip memory.

5.3 Optimizations

This section presents several possible optimizations that may improve performance of the memory subsystem in certain situations. Methods for lowering latency where possible include local caching protocols and speculative cache trust. Methods for solving the single-address problem are introduced.

5.3.1 Lowering Latency

As mentioned earlier, the primary thrust of the memory architecture is to provide sufficient parallel capacity to hide the latency of shared memory access. This led to a design with independent shared cache modules connected to processing clusters by a switching network. In the target case with abundant fine-grained parallelism this goal is achieved. However, in some cases it may be worthwhile to attempt to reduce the latency of memory access by a thread.

The most obvious way to reduce latency is by augmenting the system with cluster-local L1 caches. If a load can be satisfied with data stored locally at a cluster, the entire cost of two-way traversal across the interconnection network is avoided. However, by introducing local caches we also introduce the issue that was so neatly sidestepped by the original design: that of maintaining cache coherence.

There are many possible ways of shoehorning a global coherence policy into the system. Any such attempt must be careful to avoid certain pitfalls:

- Centralization must be avoided. Any coherence mechanism that requires some sort of centralized management structure would likely kill the advantages of distributed on-chip computing.

- Parallel update can not be allowed to choke the system. If multiple copies of a value are allowed to exist, and that value must be updated, the cost of this parallel update can not be allowed to ruin the performance of the system. Early XMT designs utilized a ring topology to propagate all updates to all caches, but this channel became the bottleneck.

A potential solution might be to implement some sort of distributed directory scheme. To take advantage of the address space partitioning present in the memory system design, each memory module could independently manage coherence for addresses that reside at that module. The module could issue leases to a processor cluster to cache certain values. It would then update or revoke a lease as necessary.

While it may be worthwhile to investigate such a solution, efficient implementation could prove difficult. The overhead and complexity for maintaining a directory-based scheme for fine-grained sharing could be prohibitive. Therefore, the rest of this section concentrates on caching schemes that can be implemented locally at a processing cluster, without resorting to external structures.

The primary idea is that in certain cases, a cluster can know that values cached locally may be trusted for a certain amount of time. If it is not known that a value in the cache may be trusted, an updated value must be fetched from a memory module. The question is how best to determine which values may be trusted.

Techniques for making this determination can be categorized as compiler-directed or hardware-directed. A compiler may be able to identify two kinds of cacheable data. Firstly, global values that are known not to be updated by any thread can be cached as long as this condition holds. Determining read-only values may be very simple or very complex, depending on how general the analysis is. Second, thread local values may be cached. Determining thread-local stack data is relatively trivial for a compiler.

The current XMT architecture already supports a very limited mechanism for the compiler to cache read-only globals in the form of the parallel PR registers. The prototype compiler already uses the PR registers to distribute global values intrinsic to the threading mechanism. A compiler could be extended to explicitly cache additional read-only globals by placing them in available PR registers, up to the (very limited) capacity of the machine.

This idea could be extending by adding to the architecture small, thread-local scratchpad memories. Global read-only values could be placed in these memories rather than in the PR registers. Furthermore, since the scratchpad memories would be locally writable, they could be used to hold thread local data.

An alternate method, could incorporate compiler identification of cacheable values, without introducing a new compiler-managed memory space. This could be achieved by augmenting the instruction set with load-cached instructions. This instruction would explicitly trust any cached values for the target address. While the additional instructions could be used to hint which addresses are worthwhile to cache, the actual cache policy would be maintained by hardware.

Another approach is to use run-time methods to take advantage of the IOS semantics, rather than relying on the compiler to identify safely cacheable variables. Section 3.3 discussed the memory consistency results of these semantics. In particular, values need not be kept consistent among different threads in between particular synchronization points. These points, such as a spawn or a gateway prefix-sum, are indicated by the assembly code. Here, we will refer to such a point as a memory barrier.

A memory barrier can serve two purposes. First, a TCU executing a memory barrier stalls until it knows that all the writes it has issued are visible to all TCUs. Second, the TCU knows that it may trust any cached value that has been written since the last memory barrier was executed. This implies that a TCU could utilize a local cache for reused thread local values. Also, values could potentially be shared by successive threads executing on the same physical TCU.

This idea of cacheable value horizons could be extended to a cluster-shared cache scheme. Consider that all TCUs in a particular cluster share a common synchronous clock. Because of this, the cache structure could be augmented to include a clock tag. Every cached value resulting from a read or write would be given a timestamp. Every TCU would also keep a timestamp indicating the last memory barrier encountered by that TCU. For every cache access, the standard cache tag check for validity would be augmented by a check to ensure that the cache value's timestamp is more recent than the TCUs memory barrier timestamp. This mechanism would allow for the reuse of data among threads executing on different TCUs within the same cluster.

The final latency-lowering strategy mentioned here takes a somewhat different tack. It is motivated by the intuition that threads (relative to other necessary loads and stores) rarely need to communicate values to other threads in the same spawn. Most of the time, any value held in a local cache is likely to be valid. Rather than to always rely on the ability to detect this condition beforehand, one may wish to temporarily trust the local value. When a memory access occurs, a request can be sent out to the memory system. But in the mean time, the requesting TCU may be allowed to proceed, treating a locally cached value as if it were known to be valid. A correction need only be made if another thread has indeed modified the value. In this case, any speculative progress made should be rolled back. Had the speculation not been performed, the TCU would have been stalled during this time anyway.

The speculative strategy may be the broadest solution, as reads which really involve communication between threads are automatically distinguished at run-time. While it doesn't in any way reduce the amount of cross-chip memory traffic, the memory system has been designed with the bandwidth to handle the worst case. There is a cost at each TCU for shadow registers and rollback logic. However, so long as global operations (such as spawns, prefix-sums, externally visible writes, or faults) are disallowed, rollback of this simple speculation should be very simple.

The best solution is likely to be some combination of the techniques presented here. Tradeoffs can be made between hardware and software complexity, and between simplicity and generality.

5.3.2 Single Address Problem

Recall that there is one class of inter-thread memory access collisions that is not solved by randomized hashing: accesses from many threads to the exact same address must go to the same memory module. This situation could create unbalanced contention at a particular module, hurting overall performance.

An important observation is that most of techniques proposed for reducing latency of memory requests also serve to alleviate the single address problem. If a memory request can be satisfied locally without resorting to a shared memory module, contention for that address is reduced. (An exception is the speculative memory strategy, which on its own does not reduce global memory requests.)

The most successful solutions can be applied when commonly-accessed data can be identified at compile time. The current prototype compiler already copies shared stack data to TCU-specific locations at the start of a spawn. This technique limits contention for these values to the beginning of the spawn, during the initial copying procedure. Successive threads executing on a TCU will not conflict with any threads on other TCUs.

This could be improved if scratchpad memories or explicit caching is available. In this case, identified values could be broadcast to all TCUs at the start of a spawn, perhaps using the same mechanism as instruction broadcast.

In cases where candidate data is not identified beforehand, the localized caching schemes may prove very useful. Traffic to the same address from a given cluster may be drastically reduced in this way.

A memory module could be able to detect cases of multiple accesses to a common address. Certainly such values could be cached close to the network port for quick access (low latency within the memory module). It could also be possible to implement some kind of run-time initiated broadcast of the value. Such a mechanism could come too late, though, as many requests from different clusters could already be queued in the interconnect by the time the condition was detected.

5.4 Instruction Memory Subsystem

These characteristics enumerated in section 3.4 enable a simple solution. The locality of access suggests placing instruction caches near the thread control units. Having many of these first-level caches provides high bandwidth to the TCUs. Since all accesses are read-only, cache coherence is a non-issue. Due to SPMD, these caches will largely require the same data. Therefore, we can have a single second level cache. Requested data can be broadcast from L2 cache to all the L1 caches simultaneously.

5.5 Parallel Register Implementation

5.5.1 ISA Specifications

The XMT instruction set architecture (ISA) includes a concept of parallel registers. In the current ISA, these are special purpose global registers which are only operated on by XMT instructions (those that extend the standard instruction set). In assembly language, the names are formed by prepending the prefix “PR” to an identifying number. The current experimental simulator supports 16 registers (PR0 to PR15). An efficient XMT system can be supported with fewer PR registers, and the final ISA may specify fewer to ease hardware support.

There are three kinds of operations that can be performed on the parallel registers (we ignore, for the moment fork issues).

1. parallel PR register read. Threads can perform parallel reads at any time.

This essentially copies a PR register to a local register.

2. parallel PR register increment. Threads can perform parallel increment at any time. This is the parallel prefix-sum with increment value 1 as described in section 3.5.

3. PR register set. This operation initializes a parallel register value. This operation is not supported in parallel. In current XMT prototypes, this operation is only performed by the serial-mode thread. Since only TCU 0 executes the serial thread, we can simplify hardware by limiting support for PR register setting to only this TCU.

5.5.2 Hardware Implementation

The fundamental idea for the parallel increment support was introduced in [55]. See Appendix D for the relevant description.

In this subsection, we suggest an implementation of this general mechanism, where the functionality is split into a centralized increment engine, and separate identical result generators.

Parallel Read

Each TCU will have its own duplicate copy of the PR registers. With this replication strategy, a parallel read operation is no more difficult (or costly) than a move between local registers. By restricting how these registers are updated (parallel increment and serial set), all copies are kept coherent. In this way, the local copies emulate global registers.

Parallel Increment

The parallel increment operation requires a centralized resource. Every TCU that wishes to perform a parallel increment, sends a single bit signal to this unit. The central unit has no concept of the actual value of the base register. In a given cycle, it simply assigns a unique count to each TCU that requested one at that cycle. This step is the synthesis operation which enables the low-overhead synchronization. The unique counts are then sent back to each TCU. Additionally, the total sum of increments requested at that cycle is sent to all TCUs, regardless of whether or not they requested a prefix-sum increment at that cycle.

A hardware automaton sits at the TCU to receive the results. The unique count is added to the current value of the base register, and returned as the result of the parallel increment operation. Finally, the total sum from that cycle is added to the base register value, and the result becomes the new base register value.

To this point, we haven't specified how the data will get from the TCUs to the centralized computation engines. In a given cycle, the central unit can receive a single bit from up to n inputs. It then produces n outputs with values from 0 potentially up to $n - 1$. These outputs can each be encoded in $lg(n)$ bits. Additionally, a sum is computed with value ranging from 0 to n. The value 0 is possible only if no TCUs requested an increment. The value n is possible only if all TCUs requested an increment. We can therefore encode the total sum in $lg(n)$ bits as well. Each TCU can interpret the value 0 as 0, if it did not request an increment and therefore knows that no others requested either, or as n, if it did request an increment and therefore knows that all others did as well.

So, to fully support the potential throughput of the prefix-sum unit, we would need $1+2lg(n)$ wires from each TCU to each possible prefix-sum target. We make the following assumptions.

- a given thread probably does not perform a prefix-sum every cycle. There is likely to be several instructions between two parallel increment operations.
- a hardware increment engine will have a limited capacity for the number of inputs accepted at once. Due to the similarity of the logic implementation to an adder's carry chain, we expect 64 inputs may be supported with a cycle time similar to that of a 64-bit integer adder. However, we may wish to build a machine with much more that 64 TCUs.

Considering these two observations together suggests a clustered solution. Several TCUs share a single hardware automaton and a single set of communication ports for handling parallel increment. Moreover, since it likely takes several cycles to send signals to the centralized unit and back, the clustering will allow several requests from different threads in a cluster to be pipelined.

While copies of the PR registers are kept next to this shared prefix-sum interface, duplicate copies can be kept at each TCU for minimum-latency parallel reads.

Another consideration is the wire cost of all these direct connections. There are several ways to reduce this burden. One reason for the high wiring cost is that a connection must exist for all possible prefix-sum (global) register bases. If the increment engines are all co-located, then the communication buses can be shared amongst the bases, at the cost of adding wires to indicate which register is being operated on.

We can further reduce the hardware costs if we limit the number of possible parallel increment targets in the ISA. The threading run-time requires 1-2 parallel increment bases, and experience shows a given algorithm tends to use 0-2 bases simultaneously. It still pays to have a large number of PR registers for communicating initialization data to the TCUs, so it probably pays to have many PR registers, with parallel increment supported on only a small subset.

Note that the wire cost for sending the results is higher than that for sending the requests. Clustering can be used to reduce this cost without reducing the throughput. Say n 1-bit requests can be sent from a cluster simultaneously. All the requests are treated at the central unit as if they were unclustered, but only the first result and total sum are sent back to the cluster. Once the first result is received, the interface at the cluster can assign results consecutively to each thread that entered a request. This is essentially an n increment prefix-sum calculation, operating on the original base plus the result from the central unit. As noted in Appendix D, with n sufficiently small, this can be easily performed.

PR Register Set

Supporting the PR register initialization function is relatively straightforward. We need to broadcast a register value and a register selection address from TCU 0 to all other TCUs. A pipelined broadcast tree would allow several subsequent pset instructions to be issued back to back in a write-and-forget manner. In this case, the time required for the updates to reach all TCUs must be considered when enforcing memory consistency, before a spawn instruction, for example. Alternatively, the spawn operation could be implemented over the same broadcast network, avoiding the timing issue.

5.5.3 Asynchronous Parallel Queues

In Section 2.1, the programming model concept of forking within spawns is described. The high-level constructs for this model include *fspawn()*, *xfork()*, and *xfork_and_init()*. The special assembly instructions used to implement these constructs are introduced in Section 2.2. The instructions include the *psalloc*, *pscommit*, and *suspend* instructions. Both the C-level and instruction-set-level constructs are defined in Appendix A.

We consider for a moment the nature of this forking model. With an *fspawn()* statement, the programmer can specify a thread for every piece of work that is initially ready to be performed. When a running thread discovers or enables additional work, it uses a fork operation to enqueue the work for later execution. When a TCU becomes idle after all the initial thread IDs have been claimed, it will dequeue some work to perform. What we wish to implement with the forking spawn model is a generalization of the serial work queue concept, that is, an asynchronous parallel queue.

The efficient implementation of an asynchronous parallel queue turns out to be a difficult problem. Consider the following objectives for such an implementation, ranked roughly in descending order of importance as applied to XMT-style parallelism.

1. Many threads should be able to enqueue new items (perform a fork) in parallel.

2. Many threads should be able to dequeue items (claim new thread IDs) in parallel.

3. A thread enqueuing an item should be delayed as little as possible. An

XMT thread performing a fork operation usually has other useful work to do, otherwise the fork wouldn't be necessary.

4. When there are plenty of items in the queue, don't delay a thread who wishes to dequeue. It's senseless for a TCU to sit idle when there is much pending work.

5. Even if there are very few items in the queue, dequeuing should be delayed as little as possible. We break this out as a separate point because it is a more difficult case.

If we implement the queueing with *psalloc* and *pscommit*, the hardware mechanisms should support these objectives. Conceptually, we have a single parallel register which can accept both visible and invisible increments. One hardware embodiment may be to have two hardware registers for the single architectural one. One register is the visible register. A *pread* instruction will return the value of this register. The second register is hidden. A *psalloc* operation results in the parallel increment of this register. The challenge is how to update the visible register as *pscommit* is performed.

One way to implement this is to have a counter which increments for each *psalloc* and decrements for each *pscommit*. The counter is implicitly initialized to zero during a *pset* operation to the PR register. Anytime that the counter goes to zero, we know that each *psalloc* was matched with its corresponding *pscommit*, and we can copy the value of the hidden register to the visible register.

Unfortunately, it could be common for new *psalloc*s to continually be coming in, meaning that the update never happens until all threads are idle waiting for work. This is equivalent to implementing a synchronous queue, and is a gross violation of objective 4.

We may try to enforce a threshold for a maximum difference between the visible and hidden registers. If the maximum difference is reached, any new *psalloc* is stalled until every outstanding *psalloc* has been committed. Of course this stalling goes against objective 3. If the maximum threshold is chosen to be too large, dequeuing will be delayed in conflict with objective 5.

One way to alleviate these problems is to partition *psalloc*s into groups of consecutive values. If each group is of size n, then the threads which receive the first n values from *psalloc* to a particular base are associated with the first group. Each group has its own *pscommit* counters. A *psalloc* which receives a higher value doesn't stall, but is associated with a higher group. Each *pscommit* is steered to the appropriate counter. When n threads from the first group have committed, the visible PR register can be incremented by n, regardless of uncommitted *psalloc*s in other threads. The second group would then be considered the first. The former counting resources formerly with the first group would then be available as the last group.

The are many possible tradeoffs between complexity and efficiency. Having a larger capacity for outstanding operations (group size n times the number of groups) increases implementation cost, but could prevent *psalloc*s from stalling. With a given capacity, fewer groups would be simpler to implement, but the larger group size could cause delays for dequeuing. When considering the complexity tradeoffs, it may be advisable to architecturally limit the *psalloc*/*pscommit* functionality to a single special register, rather than duplicate the functionality for every PR register.

In light of objective 3, it is useful to consider the semantics of the queueing instructions. We wish for a forking thread to be able to queue a value and continue without stalling. Recall that, in general, a thread can issue a store instruction and immediately continue, without considering whether the store has actually reached the shared cache. The thread must stall to wait for outstanding writes only on certain instructions. Unfortunately, *pscommit* is one such instruction. The queued value must be written before the prefix-sum operation is made visible. Ideally we'd like the queueing operation to be handled in the background, since the forking thread doesn't actually depend on the operation at all.

One solution is the replace the *psalloc*/store/*pscommit* sequence with a single instruction. This store-word-queued (*swq*) operation would look similar to a store-indexed style instruction where the indexing register is a PR register, but would implicitly imply that the PR register would be incremented to the next word.

```
swq  R_value, off(R_addr), PR_base
```

If we restrict to a single PR register, the last argument can be implicit. The TCU executing a *swq* would issue it to a processing cluster control unit, and then continue. Behind the scenes the control unit would perform the required steps: perform an allocation, issue the store, wait for a confirmation, commit the allocation. The TCU would only have to stall if the reservation station for the cluster control unit was full.

Chapter 6

Design Study

This chapter details a concrete design and analyzes its characteristics. A specific configuration is chosen for a nominal 0.07 micron process. Area costs of various components are calculated, and a rough floorplan is presented. The timing characteristics of the interconnect are estimated using analytical models.

6.1 Specifications

6.1.1 Design Goals

For this design we will target a 0.07 micron semiconductor process. Such a technology is expected to be introduced in 2008[47]. This time horizon for the study was chosen such that, from the time of writing, there is sufficient time for development and prototyping activities. Furthermore, it represents a point where:

- there will be sufficient on-chip resources to provide the processing and memory capabilities to fully exploit the potential of XMT,
- the deep submicron VLSI realities and lack of exploitable parallelism will

be severely hamstringing traditional designs,

- marshalling these capabilities to such a degree in an effective system will require a new architecture.

The goal of this chapter will be to demonstrate the buildability of a full-scale XMT design within the expected constraints. The design target will primarily be to perform well on fine-grained XMT-style applications. Throughout the design, specific architectural decisions must be made in a somewhat arbitrary fashion. It should be straightforward to incorporate the data from this configuration into a detailed architectural simulation. By evaluating application performance in such a simulation, one will obtain feedback which will undoubtably lead to reconsideration of design parameters using the quantitative approach, as per Hennessy-Patterson [24].

In keeping with this idea of establishing a starting point, the design will try to push the limits of hardware parallelism in processing and memories. We will try to simplify other aspects as much as possible. This includes simple single-thread hardware, and lack of sophisticated compiler technology. Optimal performance for serial programs will not be a primary concern.

6.1.2 Floorplan Layout

We can divide the processor modules into a few groups. The first group is made up of a number of independent processing clusters, where each cluster contains a number of TCU's. The second group is the many independent memory modules. The remaining groups are the centralized resources of the design. The parallel prefix-sum hardware, the memory interconnect, and the centralized instruction cache and broadcasting facility. to summarize the major connectivity requirements of these components:

- All the processing clusters must be connected to all the parallel prefix-sum units for parallel prefix-sum operations.
- All the processing clusters must be connected to the centralized instruction cache and broadcasting network.
- All the processing clusters must be connected to all the memory modules through the memory interconnect for data memory requests.

Based on these requirements, and considering that greater distances across chip will require greater latency of communication, we can decide on a general placement for these functions on chip. Figure 6.1 illustrates this general floorplan.

The parallel prefix-sum units, which provide the low-overhead synchronization for the system, will reside at the heart of the chip. They require relatively little chip area, and should provide low-latency connectivity to all the processing clusters.

Wrapped around the prefix-sum units will be the memory interconnect structure. Every message to and from a memory module must pass through this structure, so it is advantageous to have this located at the center of the chip. Additionally, by concentrating the data paths together, rather than stretching the switching structure around the chip, we minimize signal travel time. Figure 6.2 diagrams the memory network. The bipartite network is illustrated with example communication between a single processing cluster and a single memory module.

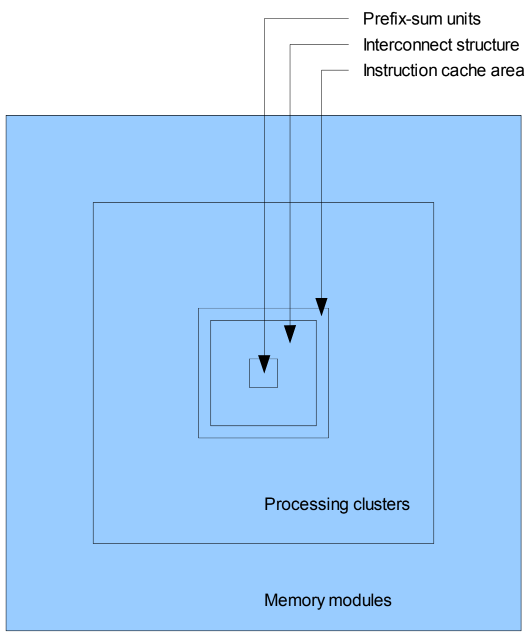


Figure 6.1: General floorplan

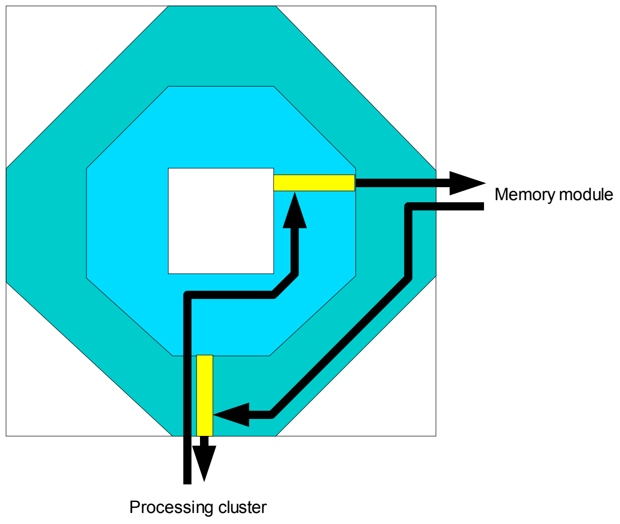


Figure 6.2: Memory network

The bulk of the chip is occupied by processing clusters and memory modules. While both require equal connectivity to the memory network, the processing clusters must additionally communicate with the prefix-sum units. The memory modules, on the other hand, must access the I/O pads, which are located around the periphery of the chip. The memory modules, then will occupy the outer area of the chip, and the processing clusters will be located within. The instruction cache and broadcasting structures service all the processing clusters, and so are placed between the processing clusters and the data memory network.

6.1.3 Specific Choices

Here we detail the specific architectural choices for the example design. These parameters will determine the chip area and signal timing characteristics which are presented in sections 6.2 and 6.3. Note that these choices were actually selected in a feedback process involving the timing and area experiments. However, here we simply present the example design, and then derive the experimental results based on that choice.

We define two basic structures for our connectivity-oriented design. A memory module is a locally synchronous module capable of handling up to one memory request per local clock cycle. A processor cluster is a locally synchronous module capable of producing up to one memory request per local clock cycle.

In keeping with our design goals, we will explore the limits of massive interthread parallelism, rather than pushing intrathread parallelism as far as possible. The Thread Control Units in this system will be fairly simple, low-issue processors. It is necessary to group several of these simple automatons together to make ample use of a shared memory network port. This system has 1024 TCU's.

Each set of 16 TCU's is grouped together around a shared memory access point to form a processing cluster. This scheme yields a total of 64 processing clusters.

Within each cluster, the TCUs are grouped into 8 simple SMT-style processors. The dual-thread design, with a simple single-issue pipeline for each thread, allows for significant reduction of hardware cost, with minimal additional complexity. Each thread has its own set of registers and fetch and decode hardware. All functional units are shared, except for the simple integer ALU's. Since approximately 50% of instructions in a typical workload are simple integer operations [48], duplicating this unit is a cheap way to increase potential instruction throughput.

Each of the 8 TCU-pairs in a cluster has its own 2 kB L1 instruction cache. While these will largely be duplicating the same code, they provide plenty of fetch bandwidth.

All TCU's in a cluster share the parallel prefix-sum interface units. They also share a single memory interface unit, which can issue one request and retire one request per cycle. Within the memory interface unit is also a shared 16 kB L1 data cache. Values in this cache can be speculatively trusted, allowing a TCU to progress (without committing) until the actual value is retrieved.

From the parallel definitions of memory module (consumer of requests) and processing cluster (producer of requests), a balanced design will satisfy memory requests with 64 independent memory modules. We consider here two variations in memory technology. One variation uses SRAM for all the memories on chip. This is traditionally the technology used for on-chip caches. For the second design, we choose to build the memory modules using DRAM, rather than SRAM, technology. DRAM, while slower than SRAM, can be much more dense.

Since DRAM combined with logic on the same wafer is a relatively immature technology, we justify this choice:

- Memory modules are not located next to processing logic. As a significant latency must be paid by every request traveling to a memory module, minimum latency within a memory module is not critical.
- This memory architecture, and XMT as a whole, leverages the bandwidth potential available on a single chip. An XMT application will not reach the same performance potential if it cannot satisfy enough memory requests on-chip. Therefore, if placing more memory on-chip can enable huge jumps in performance for more applications, it may be well worth doing.

For the SRAM-based design, we will allow to use up to the maximum chip area supported for high-performance microprocessors as estimated by the ITRS. Within the 713 mm^2 limit, each memory module can store 4 MB of data, for a system total of 256 MB on-chip memory. To illustrate another design point, we will limit the DRAM-based design to an economical 400 mm^2 chip area. Using the DRAM technology with limited chip area, each memory module is able to store 8 MB of data, for a system total of 512 MB of on-chip memory. Each memory module has an independent 20 pin bus for streaming data to and from any external memory.

The central prefix-sum module will consist of 8 independently-wired prefix-sum engines. This should be sufficient to support both the intrinsic system-threading operations, as well as user-visible parallel prefix operations.

The memory system interconnect is designed for transactions with 32 bit data. Full connectivity is provided between the processing clusters and the memory

parameter		value	
Number of TCUs		1024	
Number of processing clusters		64	
Number of memory modules		64	
SRAM design	total memory	256	MB
	max chip area	713	mm^2
DRAM design	total memory	512	MB
	max chip area	400	mm^2
Prefix-sum units		8	

Table 6.1: System Configuration

modules with 32 bit wide micropipelines. Should 64 bit transactions be required (this is the largest atomic data size supported), special support would need to be considered. One option would be to allow two data words to be chained together as a single request. Since this would require additional switching logic to give priority to chained requests, the operation of the network at every switch point could be slowed. Another option would be to hold incomplete requests at the memory module or processing cluster, until both parts of a chained request have been received. This buffering would be much simpler to implement at a processing cluster, where the maximum number of incoming requests is bounded by the number of requests sent out. Implementing this at a memory module could require significant buffering, and might result in significant delays for a chained request sent to a very busy memory module.

Table 6.1 summarizes the selected configuration.

6.2 Area Calculation

6.2.1 General Methodology

From the previous section, we have a general floorplan layout, and specific configuration parameters for the system. The next step is to estimate the chip area occupied by the various regions. There are two types of structures that must be characterized.

Wire-dominated structures, such as the memory network, occupy an area which is determined by the placement of metal interconnect wires. To determine the area occupied by these structures, we map out the layout of wiring, including grounding for signal isolation. Then, by assuming a particular wire pitch, the dimensions of the structure can be determined.

The chip area of transistor-dominated structures is primarily determined by the placement of semiconductor resources. When characterizing these structures, we estimate the number of transistors required for the device. This can be done analytically for simpler structures, and based on empirical estimates from the literature in other cases. Finally, a transistor density estimate is used to produce an estimated area cost. Logic devices, SRAM devices, and DRAM devices all tend to have very different transistor densities, so each is treated separately.

When determining the dimensions of transistor dominated structures, we also calculate the external interconnect requirements, and insure the dimensions are sufficient to support the full wiring.

Most of the figures used for area calculation come from the International Technology Roadmap for Semiconductors [47]. The values correspond to a nominal 70 micron process of 2008. The relevant figures appear in Table 6.2.

parameter	value	units
DRAM mem/area	4.33e+07	bits/mm^2
SRAM trans/area	5770000	trans/mm^2
logic trans/area	1090000	trans/mm^2
chip area	713	mm^2
signal i/o	1280	pads
on-chip local clock	6000	MHz
on-chip global clock	2500	MHz
off-chip speed	2500	MHz
wiring levels	9	levels
ground levels	3	levels
global pitch	0.00039	mm
SRAM trans/bit	6	trans/bit

Table 6.2: Technology Parameters - Area Estimation

For calculating the non-cache cost of the TCU's, we use the KSMS Estimator Version 1.0 tool [50]. This is a chip space and transistor count estimation tool directed to the Karlsruhe Simultaneous Multithreaded Simulator. This model, which has been validated against real microprocessors, is likely a conservative estimate for the simplified TCUs of the XMT system. Some of the relevant parameters used to configure the estimator appear in Table 6.3.

6.2.2 Results

The summary of area distribution appears in two tables, which only differ in the memory module configuration. The small-chip DRAM-based case appears in Table 6.4 and the larger SRAM-based case in Table 6.5. In both tables, regions are listed starting from the center of the chip, and moving outward. Cumulative area includes regions within, such that the last is the total area of the chip.

The prefix sum region was first calculated as a logic-dominated region. As an estimate of transistor cost, we used an estimate of a modern 64-bit floating-point ALU unit [48]. Even though this can be considered a conservative estimate, the area reported for the prefix-sum region was determined not by logic area, but wiring costs. The final estimate is determined by allowing for all processing clusters to be fully connected to all units. Each channel from a processing cluster to a particular prefix-sum unit requires a single wire for the increment, along with two bundles of $log_2(N)$ wires for the partial result and total sum, where N indicates the number of processing clusters in the system. The total number of wires that must enter the prefix-sum region is this value multiplied by N and again by the number of prefix-sum units. The required perimeter for concentrating this many wires determines the area for the region. The signal wiring is in a single

Name	Value	Description
Threads	2	Number of threads.
Register	32	Number of the register per thread.
Fetch Unit Count	1	Number of the fetch units.
Fetch Unit Size	1	Max num of the instr fetched per cycle.
Decode Unit Count	1	Number of the decode units.
Decode Unit Size	1	Max num of the inst decoded per cycle.
Decode Unit Predicts	1	Max number of speculative branches-
Dispatch Max Inst. Count	1	Max num of instr passed to execution units.
Dispatch Look Depth	1	Num of inst seen by dispatch unit per thread
Dispatch Queue Size	1	Size of the instr queue in the dispatch unit.
Completion Max Inst. Count	1	Max num of instr completed per cycle.
Completion Queue Size	1	Size of the instruction completion queue.
Local Load Store Unit Count	1	Number of the local load store units.
Branch Unit Count	1	Number of branche units.
Integer Unit Count	2	Number of simple integer units.
Complex Integer Unit Count	1	Number of the complex integer units.
Complex Int. Unit Pipeline Size	5	Size of the complex integer pipline

Table 6.3: KSMS Configuration - Relevant Parameters

	area (mm^2)	**cumulative area (mm^2)**	**side (mm)**
prefix sum region	1.68	1.68	1.30
interconnect region	26.32	28.00	5.29
i-cache region	8.72	36.72	6.06
broadcast region	2.46	39.18	6.26
processing region	231.64	270.82	16.46
memory region	99.26	370.08	19.24
outer rim	29.92	400.00	20.00

Table 6.4: Area Estimation Results - DRAM case

	area (mm^2)	**cumulative area (mm^2)**	**side (mm)**
prefix sum region	1.68	1.68	1.30
interconnect region	26.32	28.00	5.29
i-cache region	8.72	36.72	6.06
broadcast region	2.46	39.18	6.26
processing region	231.64	270.82	16.46
memory region	372.18	643.00	25.36
outer rim	70.00	713.00	26.70

Table 6.5: Area Estimation Results - SRAM case

global metal layer, using minimum pitch wires, and including a ground wire between every signal wire to reduce crosstalk.

The area cost of the interconnect switch is dominated by the wiring wrapped in bands around the prefix-sum region. A number of signal wires equal to the cross product of processing clusters and memory modules is required to support the fully-connected topology. This figure must be doubled to allow for switched communication in each direction: to and from memory. To tighten area used, two metal layers are dedicated to this structure. Again, global minimum pitch wires are used, and grounding/via wires are alternated with signal wires. The expansive silicon area under the interconnect wires is reserved for buffering and switching. The wiring into and out of the structure does not provide a significant constraint.

The global instruction cache area is easily calculated using SRAM models. The closely-coupled broadcast network is wired in a manner similar to the interconnect switch bands.

The area for the processing region is computed in two parts. The cache areas are calculated using the simple SRAM model of [47]. The logic resources were determined using the estimation tool mentioned in the previous section. Again, wiring was not a determining factor.

Finally, the largest chip area is dedicated to the memory modules. Values were determined using the DRAM model of [47].

6.3 Timing Estimation

6.3.1 General Methodology

Each processing cluster can be operated with an independent localized clock. With the extremely simple processing pipelines employed within, there should be no trouble reaching the ITRS projections for local clock speed. For the purposes of this study, we will assume that the processing logic operates at this clock speed.

Less simple to estimate is the timing of global communication. For this design, we have specified that communication between processing clusters and memory modules uses asynchronous micropipelines. The section will be primarily concerned with characterizing the performance of these structures.

Detailed electrical simulation for a semiconductor process which doesn't exist is not very practical. The parametric simulation models simply haven't yet been created. While it may be useful to investigate ideas using electrical simulation with current technology, we can't expect performance of a given circuit to scale predictably by any factor we might choose as the "speed" of a process. Unfortunately, when scaling deep submicron processes, the rules change for every component in a very different ways. Luckily, the global interconnect structures are quite simple, and readily yield to analytical estimation techniques. We will therefore rely on research models for deep submicron devices and interconnect.

We will primarily be concerned with two structures and their interaction, transistors and wires. In the transistor modeling, the technology parameters of Table 6.6 are used.

For evaluating transistors, the important terms are the output resistance R_{tr} and the input capacitance C_g. We will express both of these parameterized by

	value	**units**	**description**	**source**
ϵ_0	8.85e-12	F/m	permitivity of free space	
v_{sat}	80400	m/s	saturation velocity	BSIM3
L	4.5e-08	m	gate length	[47]
T_{ox}	2e-09	m	oxide thickness	[52]
V_{dd}	0.9	V	operating voltage	[52]
V_t	0.225	V	threshold voltage	[52]
e_{rox}	3.9		dielectric constant	

Table 6.6: Technology Parameters - Transistor Modeling

the ratio of gate width W to gate length L.

The expression for output resistance is derived from a updated deep-submicron model for device current in saturation, from [52].

$$R_{tr}\frac{W}{L} = \frac{0.9V_{dd}}{v_{sat}C_{ox}(V_{dd} - V_t)L}$$

C_{ox} from the above is expressed rather simply in terms of the permittivity and thickness of the oxide layer.

$$C_{ox} = \epsilon_{ox}/T_{ox}$$

For the gate capacitance, we use a standard first-order model.

$$C_g/\frac{W}{L} = L^2 C_{ox}$$

From these terms, it is straightforward to determine the inverter intrinsic delay, full- t and half- t_5 swing.

	value	**units**	**description**	**source**
ϵ_0	8.85e-12	F/m	permitivity of free space	
v_{sat}	80400	m/s	saturation velocity	BSIM3
T_w	1.09e-06	m	wire thickness	[47]
W, S	3.9e-07	m	wire width and spacing	[47]
W, S	3.9e-07	m	wire width and spacing	[47]
$R/\square$	0.016484	$\Omega/\square$	sheet resistance	[47]
T_{ins}	1.4e-06	m	interlevel dielectric thickness	[52]
e_r	1.5		dielectric constant	[47]

Table 6.7: Technology Parameters - Wire Modeling

$$t = 2.0R_{tr}\left(\frac{W}{L}\right)C_g/\left(\frac{W}{L}\right)$$

$$t_5 = \ln(\frac{1}{1-0.5})t = 0.693t$$

The relevant parameters for wire modeling appear in Table 6.7. The interconnect resistance R and capacitance C are derived as a function of wire length l.

R/l can be straightforwardly derived from sheet resistance by considering the wire pitch. C/l requires a much more sophisticated analysis. For this we use the two-ground plane model of [9] for estimating both the line-to-line and line-to-ground plane capacitances. It is assumed that ground wires alternate with signal wires to reduce crosstalk effects.

From the RC product of the above terms, a quadratic dependence on the length of the wire is apparent.

$$RC/l^2 = R/lC/l$$

By introducing optimally spaced repeaters into the wire path, we can linearize the interconnect delay. For modeling the driver/wire performance, we use the formula for end-to-end delay from [46]. This formula expresses the delay t as a quadratic function of the length l of the segment. We substitute vt for l, and then solve for velocity v (the target for optimization) in terms of t. By taking the derivative of this expression with respect to t, and equating to zero, we can solve for the optimal single-segment time. Its then a simple matter to back-substitute to calculate the optimal signal propagation speed.

6.3.2 Results

The next step is to apply the generalized modeling to the actual structures in the system. There are two important timing characteristics for a given stage in a micropipeline. The first T_{lat} is the latency for data to travel through a given stage. If a micropipeline is initially empty, the total travel time for data is T_{lat} multiplied by the number of stages. The second figure T_{cyc} is the amount of time after one packet of data enters a stage until the next packet is allowed to enter, assuming no delays further down the pipe. The potential throughput is determined by T_{cyc}.

Please bear in mind that the results presented here are intended to be relatively coarse estimates of the timing of the proposed devices. The timing estimates come from predictive analytical models, and the cycle counts are derived from these using predicted processor cycle times.

	length (m)	latency (s)	latency (cyc)	stages
mem max path	0.0194	$+1.17E-09$	7.0	12.7
mem avg path	0.0153	$+9.20E-10$	5.5	10.0
ps max path	0.0035	$+2.11E-10$	1.3	2.3
broadcast max	0.0125	$+7.53E-10$	4.5	8.2
broadcast avg	0.0063	$+3.77E-10$	2.3	4.1

Table 6.8: Timing Estimation Results

Expressions for T_{lat} and T_{cyc} can be derived by tracing the datapaths through a stage of a micropipeline.

$$T_{lat} = T_{int} + 13t$$

$$T_{cyc} = 2T_{int} + 25t$$

Where t is the inverter intrinsic delay, and T_{int} is the interconnect delay.

We wish to tune the interconnect so that the producers and consumers can interact at full speed. We therefore specify that T_{cyc} be equal to the local clock period. With T_{cyc} fixed, T_{int} is easily determined, and from this T_{lat}.

Using the floorplan calculations from section 6.2, the length of the global datapaths can be calculated. The spacing for a micropipeline stage is determined by T_{int} and the optimal propagation speed. This then gives the number of stages for a given datapath. Table 6.8 presents the timing results.

It's useful to consider these figures in light of the broader objective of this thesis. Notice that a two way traversal of the memory interconnection network could be expected to average less than 16 local clock cycles. Not coincidentally, the choice of 16 TCUs per processing cluster implies that a cluster could issue 16 requests from different threads on successive cycles, before the first thread would be required to produce another request. The first thread could have received its result and be ready to issue its next request. If we allow the processing clusters to operate the ITRS estimate for local clock rate, and consider a situation where we have an abundance of memory-intensive threads, the system described here could be capable of supporting a sustained memory transaction throughput of 1430 gigabytes per second. Other than the condition that the dataset fit in on-chip memory, this steady-state estimate makes no assumptions about spatial or temporal locality, or regular patterns of access.

As a quick point of comparison, consider a theoretical fully-synchronous cross-bar interconnect. We will assume for the moment that the issue of scheduling such a network is solved in a perfectly efficient way. We will further assume that the device can operate at the maximum global clock rate predicted by the ITRS, and will ignore any power issues with driving such a large synchronous structure. Finally, we will assume that the average latency for a memory request is also less than 16 cycles. Under these assumptions, the design could be expected to exhibit a peak performance of less than 600 gigabytes per second.

Chapter 7

Conclusion

This thesis has detailed for the first time a memory subsystem architecture with the potential to satisfy the needs of a full-scale XMT computing system. We have explored the unique needs faced in XMT memory design. An architecture, using on-chip parallel memories and reduced-synchrony interconnects, has been proposed. Many aspects of integrating such an architecture into a practical system have been considered. Modeling of the necessary structures indicates that such an architecture can be implementable, with reasonable chip area and timing characteristics, in the near future. The goal of supporting fine-grained XMT-style programs has been achieved with support for an estimated 1430 gigabytes per second of fine-grained, non-localized transactions.

In many ways, research into this direction has just begun. Opportunities for future research abound. One area of further work is more detailed electrical analysis. Alternatives for datapath switching, such as fully-asynchronous and domain-synchronous switches, should be compared. The implications of power costs should be explored. Many questions about these new structures may only be answered by building VLSI prototypes.

Detailed microarchitectural simulators need to be created to provide insight into the many architectural choices and tradeoffs presented. Cache-sizing and coherence techniques need to be empirically evaluated with real work loads. Forking implementation and variable granularity memories should be studied more carefully. Of course, for meaningful simulator results, the body of realistic XMT programs needs to continue to be grown.

Appendix A

Glossary of New Programming Constructs

fspawn [C]

Spawn of virtual threads with the ability to fork additional threads.

```
fspawn(nthreads,offset);
{
        int TID;
}
join();
```

The *fspawn* statement begins a parallel region. Threads are created with variable `TID` initialized to values ranging from `offset` to `offset` + `nthreads` - 1. The bracketed statements between an *fspawn* and a *join* constitute the code executed by each thread. Unlike the *spawn* statement, the *fspawn* statement allows for threads to *xfork* during execution.

join [C]

See *spawn* [C] or *fspawn*.

pinc [assembly]

Parallel prefix-sum to register, with increment value of 1.

```
pinc     PRi, $r
```

The *pinc* instruction has the following semantics: `$r` $\leftarrow$ `PRi`; `PRi` $\leftarrow$ `PRi` $+ 1$. There is support in hardware for multiple operations from different threads to occur simultaneously. The threading runtime uses this construct to generate unique thread IDs.

pread [assembly]

Parallel read of a PR register (parallel prefix-sum with value 0).

```
pread    PRi, $r
```

The *pread* instruction has the following semantics: `$r` $\leftarrow$ `PRi`. There is support in hardware for multiple operations from different threads to occur simultaneously. The threading runtime uses this construct to efficiently read global values.

ps [C]

Prefix-sum function (atomic fetch-and-add).

```
int ps (int *base, int inc);
```

The *ps* statement increments (`*base`) by `inc` and returns the original value of (`*base`). The difference between this statement and the standard `++` operator is that *ps* is implemented as an atomic operation. This property allows for the use of *ps* for coordination and synchronization between threads.

psalloc [assembly]

Allocate a value from a PR register, without updating the register.

```
psalloc  PRi, $r
pscommit PRi, $r
```

The *psalloc* instruction has the following semantics: `$r` $\leftarrow$ `PRi`; `temp` $\leftarrow$ `PRi` $+ 1$. A subsequent *pscommit* instruction issued from the same TCU has the following semantics: `PRi` $\leftarrow$ `temp`. The value returned from the *psalloc* is unique (ie. not given to any other *psalloc*), however the update is not visible to any *pread* until a matching *pscommit* There is support in hardware for multiple operations from different threads to occur simultaneously. These instructions allow for efficient implementation of the *xfork_and_init* XMT C statement.

pscommit [assembly]

See *psalloc*.

pset [assembly]

Initialize a parallel register.

```
pset     PRi, $r
```

The *pset* instruction has the following semantics: `PRi` $\leftarrow$ `$r`. As the updated value must be broadcast to all TCUs, there is no support for multiple *pset* operations to occur simultaneously. The threading runtime uses this construct to initialize global values before a *spawn*.

psm [assembly]

Prefix-sum to memory, with an arbitrary increment value.

```
psm  $ri,off($rj),$rk
```

The psm instruction has the following semantics: `temp` ← Mem(`$rj`+`off`); Mem(`$rj`+`off`) ← `temp` + `$rk`; `$ri` ← `temp`. There is support in hardware for multiple operations from different threads to occur simultaneously, provided they operate on different bases. Operations to the same base will be queued. The current compiler uses this instruction to support the *ps* function.

spawn [C]

Spawn of virtual threads.

```
spawn(nthreads,offset);
{
        int TID;
}
join();
```

The *spawn* statement begins a parallel region. Threads are created with variable `TID` initialized to values ranging from `offset` to `offset` + `nthreads` - 1. The bracketed statements between an *spawn* and a *join* constitute the code executed by each thread.

spawn [assembly]

Spawn operation, all TCUs are interrupted and execute at this `pc`.

```
spawn
```

The spawn instruction has the following semantics: `temp` ← `pc` and then for all TCUs `npc` ← `temp`. The threading runtime uses this construct to initiate parallel execution of a spawn block.

suspend [assembly]

Suspend TCU until PR register reaches threshold.

```
suspend  PRi, $r
```

The *suspend* instruction simply suspends a TCU without using resources. The TCU resumes if the specified PR register exceeds the given value (`PRi` > `$r`). The threading runtime uses this construct to suspend a TCU during the execution of an *fspawn* region.

xfork [C]

Request an additional thread be created.

```
xfork ( );
```

The *xfork* statement is used within an *fspawn* block to indicate that an additional thread should be created. The effect is as if `nthreads` is incremented by one. Many different threads can perform *xfork* in parallel to dynamically adapt execution as work is discovered.

xfork_and_init [C]

Request an additional thread be created, and provide initial data.

```
xfork_and_init (int *array, int *data);
```

The *xfork_and_init* statement is used within an *fspawn* block to indicate that an additional thread should be created. The effect is as if `nthreads` is incremented by one. Also, *xfork_and_init* takes as argument the base of an array and data to be placed in it. The hardware ensures that the data is initialized before the corresponding thread is initiated. Many different threads can perform *xfork_and_init* in parallel to dynamically adapt execution as work is discovered.

Appendix B

XMT-M

This appendix is an unpublished update to [5], and describes early XMT architecture design prior to the contributions of this thesis. It is based on the work of Efraim Berkovich, Joseph Nuzman, Manoj Franklin, Bruce Jacob, and Uzi Vishkin.

This report presents XMT-M, a microarchitecture implementation of the XMT model that is possible with current technology. XMT-M offers an engineering *design point* that addresses four concerns: *buildability, programmability, performance, and scalability.* The XMT-M hardware is geared to execute multiple threads in parallel on a single chip: relying on very few new gadgets, it can execute parallel threads without busy-waits! Existing code can be run on XMT-M as a single thread without any modifications, thereby providing backward compatibility for commercial acceptance. Simulation-based studies of XMT-M demonstrate considerable improvements in performance relative to the best serial processor even for small, and therefore practical, input sizes.

B.1 Introduction

The coming years promise to be exciting ones in the area of computer architecture. Continued scaling of sub-micron technology will give us orders of magnitude increase in on-chip hardware resources. Even by conservative estimates a single chip will have a billion transistors in a few years. Exploiting parallelism in a big way is a natural way to translate this increase in transistor count to completing individual tasks faster.

Parallelism had been traditionally exploited at coarse- and fine-grained levels. Emphasizing the buildable in the short term, traditional techniques targeting coarse-grained parallelism have focused primarily on MPPs (massively parallel processors). Although MPPs provide the strongest available machines for some time-critical applications, they have had very little impact on the mainstream computer market [12]. Most computers today are uniprocessors, and even large servers have only modest numbers of processors. A recent report from the President's Information Technology Advisory Committee (PITAC) [27] has acknowledged the importance and difficulty of achieving scalable application performance on today's parallel machines. According to the report, *"there is substantive evidence that current scalable parallel architectures are not well suited for a number of important applications, especially those where the computations are highly irregular or those where huge quantities of data must be transferred from memory to support the calculation"*.

The commodity microprocessor industry has been traditionally looking to fine-grained or instruction level parallelism (ILP) for improving performance, with sophisticated microarchitectural techniques (such as pipelining, branch prediction, out-of-order execution, and superscalar execution) and sophisticated compiler optimizations, but with little help from programmers. Such hardware-centered techniques appear to have scalability problems in the sub-micron technology era, and are already appearing to run out of steam. Compiler-centered techniques also are handicapped, primarily due to the artificial dependencies introduced by serial programming.

On analyzing this scenario, it becomes apparent that the huge investment in serial software has forced programmers to hide most of the parallelism present in an application by expressing the algorithm in a serial form, and delegating it to the compiler and the hardware to re-extract (a part of) that hidden parallelism. The result has been that both hardware complexity and compiler complexity have been increasing monotonically, with a less satisfying improvement in performance! However, we are reaching a point in time when such evolutionary approaches can no longer bear much fruit, because of increasing complexity and fast approaching physical limits. According to a recent position paper by Dally and Lacy [12], *"over the past 20 years, the increased density of VLSI chips was applied to close the gap between microprocessors and high-end CPUs. Today this gap is fully closed and adding devices to uniprocessors is well beyond the point of diminishing returns".*

To get significant increases in computing power, a radically different approach may be needed. One such approach is to "set free the crippled programmers", so that they are not forced to suppress the parallelism they observe, and are instead allowed to explicitly specify the parallelism. The books [2] [11] [26] attest to the many great ideas that the parallel computing field has developed over the years, although some of the ideas were ahead of the implementation technology and are still waiting to be put to practical use. Culler and Singh, in their recent book on Parallel Computer Architecture [11], mention under the title "Potential Breakthroughs" (p. 961): *"breakthrough may come come from architecture if we can somehow design machines in a cost-effective way that makes it much less important for a programmer to worry about data locality and communication; that is, to truly design a machine that can look to the programmer like a PRAM."* The recently proposed *explicit multi-threading (XMT)* framework [56] was influenced by a hope that this can be done.

We view ILP as the main success story form of parallelism thus far, as it was adopted in a big way in the commercial world for reducing the completion time of general purpose applications. XMT aspires to *expand the ILP "parallelism bridgehead"* with the "ground forces" of algorithm-level parallelism (which is guided by a *sound theoretical foundation*), by letting programmers express both fine-grained and coarse-grained parallelism in a natural way[1].

The XMT framework also permits decentralized and scalable processors, with reduced hardware complexity. Decentralization is very important, because in the future, *wire delays will become the dominant factor in chip performance* [42]. By wire delays we mean on-chip delay of connections between gates. The Semicon-

[1]Designed for reducing data access, communication and synchronization cost for current multiprocessors, there has been the parallel programming methodology as described in Section 2.2 of [11]. There has also been a related evolutionary approach to let programmers express some of the (coarse-grained) parallelism with the use of heavy-weight forks (carried out by the operating system) and light-weight threads (using library functions), to be run on multiprocessors. However, it has not yet been demonstrated that general-purpose applications could benefit much from these techniques; two concrete but pointed examples are breadth-first-search on graphs and searching directed acyclic graphs; more generally, irregular integer applications of the kind taught in standard Computer Science *algorithms and data-structure* courses.

ductor Industry Association estimates that, within a decade, only 16% of a chip will be reachable in a clock cycle [42]. Microarchitectures will have to use decentralization techniques to tolerate long on-chip communication latencies, i.e., localize communication so as to make infrequent use of cross-chip signal propagation.

The objective of this paper is to explore a decentralized microarchitecture implementation for the XMT paradigm. The highlights of the investigated microarchitecture, called XMT-M, are: (i) decentralized processing elements that can execute multiple threads in parallel, (ii) independence of order among the concurrent threads, (iii) relaxed memory consistency model, and (iv) buildability (with current technology).

The rest of this paper is organized as follows. Section 2 provides background material on the XMT framework. Section 3 describes XMT-M, a realizable microarchitecture implementation of the XMT paradigm. Section 4 presents an experimental analysis of XMT-M's performance, conducted with a detailed simulator. In particular, it shows that even for small input sizes, the XMT processor's performance is significantly better than that of the best serial processor that has comparable hardware. Section 5 discusses related work and highlight the differences with the XMT approach. Finally, Section 6 presents the major conclusions and directions for future work.

B.2 Explicit Multi-Threading (XMT)

The XMT framework is grounded in a rather ambitious vision aimed at on-chip general-purpose parallel computing [56]; that is, presenting a competitive alternative to the state-of-the-art serial processors and their successors in the next decade. Towards that end, the broad XMT framework spans the entire spectrum from algorithms through architecture to implementation. This section provides a brief description of the XMT framework.

B.2.1 The XMT Programming Model

The programming model underlying the XMT framework is *parallel* in nature, as opposed to the serial model used in most computers. To be specific, XMT uses an *arbitrary CRCW (concurrent read concurrent write) SPMD (single program multiple data)* programming model. SPMD implies concurrent threads that execute the same code on different data; it is a more implementable extension of the classical PRAM model [26]. The XMT threads can be moderately long, providing some locality of reference[2]. Figure B.1 illustrates the XMT programming model. The (virtual) threads, initiated by the Spawn and terminated by the Join, have the same code. At run-time, different threads may have different lengths, based on the control flow paths taken through them. The arbitrary CRCW aspect of the model dictates that concurrent writes into the same memory location result in having an arbitrary one among these writes to succeed; that is, one thread writes into the memory location while the others bypass to their next instruction. This permits each thread to progress at its own speed from its initiating Spawn to its terminating Join, without ever having to wait for other threads; that is, no thread ever does a busy-wait for another thread. Inter-thread synchronization

[2]It is important to note that traditional multiprocessors exploit coarse-grain parallelism with the use of very long threads, which provide even more locality. However, programmers find it more difficult to reason about coarse-grain parallelism than fine-grain parallelism. Thus, there is a trade-off between thread length (which affects locality of reference) and programmability.

occurs only at the Joins. We say that the XMT programming model inherits the *independence of order semantics (IOS)* of the arbitrary CRCW and builds on it. A major advantage of using this SPMD model is that it is an extension of the classical PRAM model, for which a vast body of parallel algorithms are available in the literature.

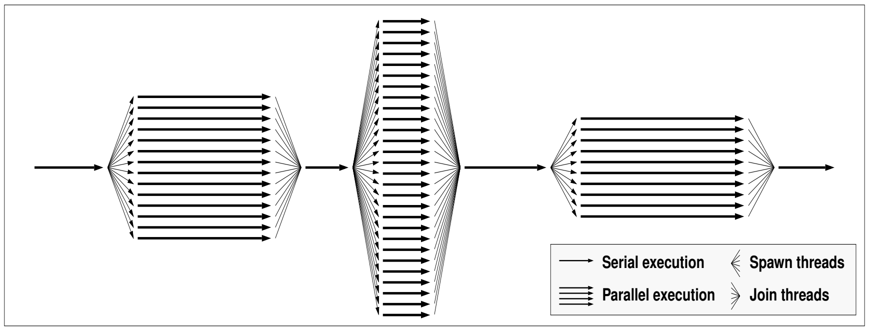


Figure B.1: Parallelism profile for the XMT model

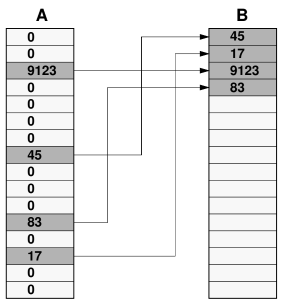
B.2.2 XMT High-Level Language Level

The XMT high-level language level includes SPMD extensions to standard serial high-level languages. The extension includes Spawn and Prefix-sum statements. The prefix-sum statement is used to synchronize between threads. Prefix-sum is similar to the fetch-and-increment used in the NYU Ultra Computer [18], and has the following semantics:

```
Prefix-sum B, R;     (i) B = B + R and (ii) R = initial value of B
```

The primitive by itself may not be very interesting, but happens to be very useful when several threads simultaneously perform a prefix-sum against a common base. In such a situation, the multiple prefix-sum operations can be combined by the hardware to form a single *multi-operand* prefix-sum operation. Because each prefix-sum is atomic, each thread will receive a different value in its local storage R (due to independence of order semantics, any order is acceptable). The prefix-sum command can be used for implementing functions such as (i) load balancing (parallel implementation of queue/stack), and (ii) inter-thread synchronization.

We shall use a simple example to clarify the XMT programming model and the use of prefix-sum. Suppose we have an array of integers, A, and wish to "compact" the array by copying all non-zero values to another array, B, in an arbitrary order (cf. left hand side of Figure B.2). The right hand side of Figure B.2 gives a XMT high-level language code to do this array compaction.



```
int base = 0;              # a global variable, shared by all threads
Spawn(n)                   # spawn threads with ID 0 through n-1
{
    int t_id;              # thread ID number: between 0 and n-1
    int e = 1;             # local variable, private to each thread
    if (A[t_id])           # check if array element is non-zero
    {
        PS(base, e);       # perform prefix-sum to get new value for e
        B[e] = A[t_id];    # copy element from array A to B (index e)
    }
}                          # implied Join
n = base;
```

Figure B.2: The array compaction problem. The non-zero values in array A are copied to array B, in an arbitrary order. The code on the right hand side gives an XMT high-level program to solve the array compaction problem.

B.2.3 The XMT Instruction Set Architecture (ISA)

For most part, the XMT ISA is the same as any "standard" ISA. Three new instructions are added to support the XMT programming model—a Spawn instruction, a Join instruction, and a Prefix-sum instruction. The Spawn instruction and Join instruction are added to enable transitions back and forth from the serial mode to the parallel mode. The prefix-sum (PS) instruction is introduced as a primitive for coordinating parallel threads (and other uses). It is important to note that an existing (serial) code forms legal single-thread code for an XMT processor; this phenomenon helps to provide object code compatibility for existing code.

Register Model

The XMT ISA specifies two types of registers—global and local. The global registers are visible to all threads of a spawn-join pair and the serial thread. The local registers are specific to each virtual thread, and are visible only to the relevant thread. To provide compatibility with existing binaries, assemblers, compiler, development tools, and operating systems, one can simply divide the register space into two partitions so that the lower partition refers to the global registers and the upper partitions refers to the local registers. For instance, a reference to register `R5` in the MIPS assembly language implies a global register access, while a reference to register `R37` implies access to a local register.

Memory Model

The XMT framework supports a *shared memory* model; that is, all concurrent threads see a common, shared memory address space.

Memory Consistency Model. The XMT memory model supports a weak consistency model between the concurrent threads of a Spawn-Join pair. This stems from XMT's independence of order semantics. Memory reads and writes from concurrent threads are generally not ordered. When an inter-thread ordering is required, a prefix-sum instruction is used. The following example code illustrates this.

```
sw   t1, A(t0)    # write local value to A[i],
                  #   based on the thread ID
psi  g1, t2, 1    # use PS to coordinate thread accesses,
                  #   g1 is initialized to 0
beq  t2, r0, DONE # if the PS result is zero, we're done
xori t3, t0, 1    # if i is even, t3=i+1; else t3=i-1
lw   t4, A(t3)    # load A[t3] into t4
```

The above code can result in the following sequence of events:

Thread 0	**Thread 1**
Write to A[0]	Write to A[1]
PS operation (gets 0)	PS operation (gets 1)
PS result is 0, so go to DONE	PS result is 1, so continue
	Read A[0]

As per the semantics of the program, values written in one thread before the prefix-sum must be visible to the remaining threads once they execute their prefix-sum. Thus, the other thread is assured of getting the correct value of A[i]. We call this type of prefix-sum a "gatekeeper" prefix-sum. Gatekeeper prefix-sums can be identified at compile time.

B.2.4 A Design Principle Guiding XMT: No-busy-waits Finite State Machines

The XMT framework is based on an interesting design principle or "design ideal" called *no-busy-waits finite state machines (NBW FSMs).* According to that ideal, progress is achieved by sharing the work among FSMs; however, a delayed FSM cannot suspend the progress of other FSMs. The latter means that no FSM can force other FSMs to waste cycles by doing busy-waiting. The NBW FSMs ideal guided the design of the XMT high-level programming language and assembly language. The FSMs that are reflected in these languages are *virtual*. The IOS in the assembly language allows each thread to progress at its own speed from its initiating Spawn command to its terminating Join command, without having to ever wait for other threads. Synchronization occurs only at the Join command, where all threads must terminate before the execution can proceed as a serial thread. That is, in line with the NBW FSMs ideal, a virtual thread avoids busy-waits by terminating itself ("committing suicide") just before the thread could have run into a busy-wait situation. As we will see in Section 3, at the microarchitecture level we will only be able to alleviate violations of the NBW FSMs principle but not to completely avoid them.

B.3 Decentralized XMT Microarchitecture

The XMT framework can be implemented in the hardware in a variety of ways. This section details one possible microarchitecture implementation for XMT. This implementation is based on current technology, and is intended to offer architectural insight into XMT, not to stand as the sole microarchitecture choice for the XMT framework.

To begin with, we anticipate that much of the communication will be intra-thread. This suggests a decentralized organization, with multiple processing elements or *thread control units (TCUs)* to execute concurrent threads. Each TCU has its own fetch unit, decode unit, register file, and dispatch buffer. The TCUs are independent in that they do not perform cross-checking of dependencies while executing the instructions of their threads. In order to maximize the use of certain resources, several TCUs are grouped together as a *cluster*, as shown in Figure B.3. The TCUs in a cluster share a common per-cluster L1 instruction cache, a common set of functional units, and a common per-cluster L1 data cache.

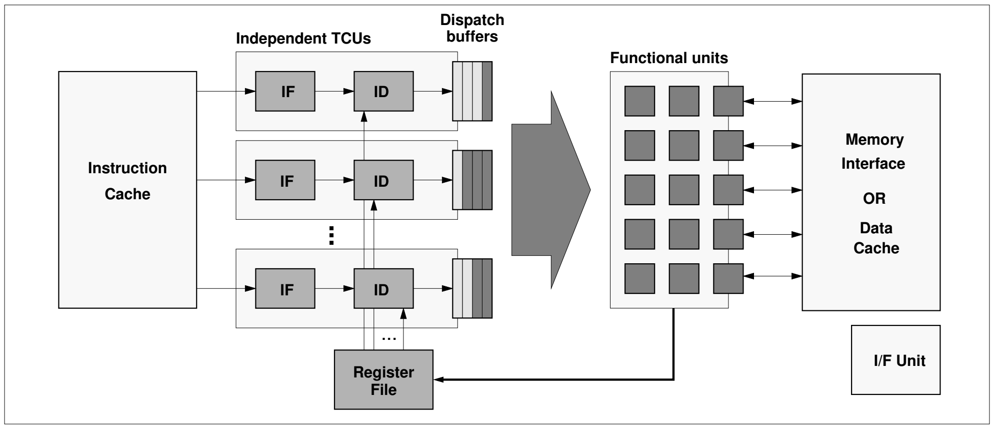


Figure B.3: An XMT-M Cluster. Each XMT-M cluster includes a shared instruction cache, a shared data cache, multiple TCUs, and a number of shared functional units.

How many TCUs should be there in a cluster? The answer depends on the interconnect wire delays within a cluster, i.e., the wire delays from the shared L1 instruction cache, the functional units, and the L1 data cache to the TCUs. If these data transfers take multiple clock cycles, then the benefits of clustering the TCUs together may become counter-productive.

Figure B.4 illustrates the organization of multiple clusters within the XMT-M processor, and shows the inter-cluster communication channels. These channels take multiple cycles for a data transfer. The operations requiring inter-cluster communication are: (i) prefix-sum, (ii) spawn/join, (iii) global register file access, and (iv) memory access.

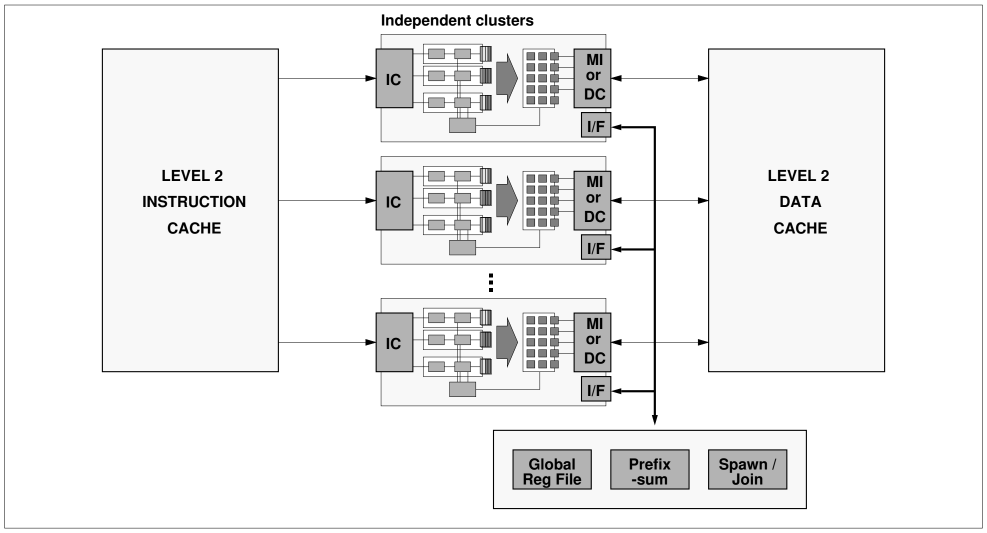


Figure B.4: An XMT-M processor. Communication paths in this diagram require multiple cycles.

B.3.1 Prefix-sum Implementation

The prefix-sum mechanism goes through a central facility that can compute results from n threads in O(1) time, not O(n) time [55]. There is a dedicated prefix-sum bus for broadcasting the prefix-sum results. Clusters have a point-to-point connection to the global prefix-sum unit to which they send their prefix-sum requests for processing.

All prefix-sum requests with a common base that arrive at the central prefix-sum unit in the same time slice are processed simultaneously, and the results are sent out simultaneously. The whole process is pipelined, so that a number of different prefix-sum operations can be in flight at the same time. The base register contents and register identifier are fired out on a shared bus as soon as the first request for a particular base arrives. The I/F (interface) unit in each cluster listens on the shared bus for these broadcasts.

First, we notice that a prefix-sum request that contributes zero to the base is equivalent to a read of the base, with no specified ordering. Thus, these requests can be handled locally at the cluster by reading a local copy of the base register. Non-zero prefix-sum requests from the same cluster using the same base can be combined into a single request to the global prefix-sum unit. The global unit groups these cluster requests into batches of the same base, performs a prefix-sum across the batch, and broadcasts the results on the prefix-sum bus. Each cluster listens on the bus, and derives its range of values within that cycle's batch. The cluster also updates its local copy of the base register. Each cluster assigns unique values from its prefix-sum range to its local prefix-sum requests. (A prefix-sum hardware implementation that avoids any serialization is described in [55]. We also note that in the case of 1-bit prefix-sum requests, the number of wires necessary for the shared bus is $c \log_2(n/c) + \log_2(c!)$, where c is the number of clusters and n is the number of TCUs.)

B.3.2 Spawn/Join Implementation

The XMT-M processor can be in one of two modes at any given time—serial or parallel. In the serial mode, only the first TCU is active. Execution of a Spawn instruction causes a transition from the serial mode into the parallel mode. The spawning is performed by the Spawn Control Unit (SCU). The SCU does two main things: (i) it activates the TCUs whenever a Spawn is executed, and (ii) it discovers when all virtual threads have been executed so that the processor can resume serial mode. The SCU broadcasts the Spawn command the the number of threads (`n`) on a bus that connects to the TCUs. Each TCU, upon receiving the Spawn command, executes the flowchart given in Figure B.5.

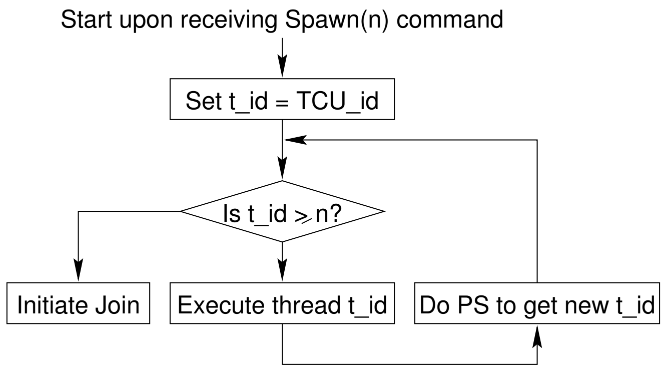


Figure B.5: Flowchart depicting activities of a TCU upon receiving a Spawn command

If the number of virtual threads is less than the number of TCUs, t, then all of the threads are initiated at the same time. If the number of virtual threads is more than the number of TCUs, then the first t virtual threads are initiated in the t TCUs. The remaining threads are initiated as and when individual TCUs become free. Notice that it would have been straightforward to spawn the second set of threads (threads with label $\geq t$) after all of the TCUs have completed the execution of the first set of threads assigned to them. However, if some TCUs terminate and wait, while others continue, a “gross violation” of the NBW-FSMs ideal occurs. To alleviate this violation, we have devised a hardware-based scheme that makes use of the IOS between threads. As and when a TCU completes the execution of its thread, it sends a prefix-sum request to the SCU. The SCU performs the prefix-sum operation, and sends to the waiting TCUs a new `t_id`. Each of the waiting TCU makes sure that its `t_id` is less than `n` before proceeding to execute the thread; otherwise, the TCU initiates a join operation. This type of thread allocation continues until all virtual threads of a Spawn instruction have been initiated.

In the parallel mode, we also take advantage of the SPMD style instruction code in the following way. The code is broadcast on the bus so that all TCUs can simultaneously get the code to be executed.

B.3.3 Global Register Coordination

It is likely that for global register coordination, we can use the same broadcast data bus as the prefix-sum unit. Because that bus activity depends on the frequency of prefix-sum operations, we can use the extra capacity to broadcast global register values when they are written. Each local register file can keep the values of the global registers for use by the cluster functional units.

Whenever a thread writes to a global register, the new value is written to the local copy of the shared register and then is sent out on the shared bus when the bus becomes available. To maintain coherence, the processor does not restart serial mode after a join until all register writes have been broadcast. Also, if one thread writes to a shared register and another thread (or threads) needs to read that value, those accesses are prioritized by using a *gatekeeper prefix-sum.* In such a situation, the thread that writes to the register would issue its prefix-sum request only after its write has gone out on the bus and is therefore visible to all the clusters. In this way, a delayed coherence can be maintained across all the global register copies, and the clusters can safely use their local global register values without fear that the register values are incorrect. Note that such a relaxed consistency model is possible among the register files, because the XMT programming model allows it.

B.3.4 Memory System

The XMT framework supports a *shared memory* model; that is, all concurrent threads see a common, shared memory address space. We can think of two alternatives for implementing the top portion of the memory hierarchy for such a system—shared cache and distributed caches. The shared cache implementation has the advantage of not having to deal with issues such as cache coherency. However, its access time is likely to be higher, because of interconnect demands. The distributed cache implementation permits each TCU or cluster to have a local cache, thereby providing faster access to the top portion of the memory hierarchy. However, it has to deal with the problem of maintaining coherency between the multiple caches. Further research is needed to determine which of the two options is best for the XMT framework for different technologies. For instance, if a high miss rate exists in the local caches, then each memory access is likely to be comparable in duration to an access of a shared cache. In that case, it makes sense to avoid the problem of cache coherence in the design, and implement only a single shared cache. The emphasis in this paper on a decentralized architecture component, and existing technology led us in the direction of local caches.

The memory system we investigate in this paper for XMT-M is as follows. Each cluster has a small level-1 cache. Multiple level-1 caches are connected together by an interconnect. The next level of the memory hierarchy consists of a large shared cache (level-2 cache), which connects to main memory. A large number of pipelined memory requests can be pending at a time, as in the Tera processor [3]. The idea is to use an overabundance of memory requests at each level of the memory hierarchy to hide memory latency. The interconnect used to tie the caches can be a crossbar, a shared bus, a ring, etc.

Cache Coherence

When a shared memory model is implemented in a distributed manner, maintaining a consistent view of the memory for all the processing elements is vital. The use of distributed caches necessitates implementing protocols for maintaining cache coherence. Cache coherence protocols come under two broad categories—invalidate-based and update-based. In a write-back write-invalidate coherence scheme, a processor doing a write waits to get access to the shared bus. It then broadcasts the write address. The other caches snoop the bus and invalidate that block if present. The writing processor then has exclusive access to that cache block and keeps writing to the copy in its local cache. Another processor reading that same block will cause a read miss at its cache, and after getting access to the bus will send a read request to memory. Because the processor with exclusive access to the block is snooping the bus, it will gain access to the bus, and send the updated version of the block and abort the access to memory. This type of protocol works well when there is not much of data sharing.

In the analogous case in an update-based protocol, a processor sends a write request to its local cache, and the request gets broadcast to the local caches of all processors. Upon receiving the update, the local caches update the relevant block if present. Any processor that needs to read a memory location will get the value from its local cache or from the next level of the memory hierarchy. This type of protocol generally results in high bandwidth requirements, because of using write-through caches.

For the XMT-M memory system, we chose an update-based protocol because the XMT memory consistency model (by virtue of its independence of order semantics (IOS)) permits a relaxed update policy. Therefore, after a TCU

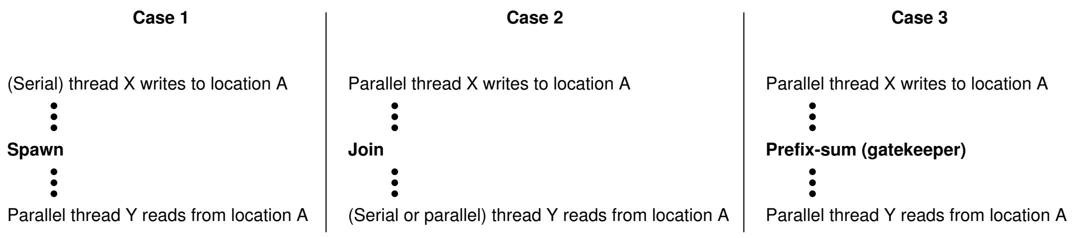

Case 1
(Serial) thread X writes to location A
Spawn
Parallel thread Y reads from location A
Case 2
Parallel thread X writes to location A
Join
(Serial or parallel) thread Y reads from location A
Case 3
Parallel thread X writes to location A
Prefix-sum (gatekeeper)
Parallel thread Y reads from location A

Figure B.6: Cache coherence hazards for XMT

sends a write request to its local cache, it can generally continue executing its thread without waiting for the write to be globally performed. However, there are three cases allowed by the programming model where this write-and-continue policy must be modified. These three "coherence hazards" are summarized in Figure B.6. The first case is that writes from the serial thread must complete before the system initiates a spawn. The second case is that writes from spawned threads must complete before the system restarts the serial thread. The third case occurs when a prefix-sum instruction and a branch based on the outcome of that prefix-sum instruction separate a read from a write. Thus, the writing TCU will stall executing its (1) spawn, (2) join, or (3) gatekeeper prefix-sum operation until it is sure that its write request has reached all other caches. The programming model guarantees that no thread will attempt to read the updated data before the next spawn, join, or gatekeeper prefix-sum operation executes. Thus, the caches are allowed to become inconsistent with each other for extended periods of time. This protocol may occasionally stall the writing thread; the stall time depends on how long it takes to broadcast a write to all the caches. With a ring-based interconnect, this stall time would be the time it takes for a write request to go around the ring plus the time it takes for the local ring segment to become available, but even in that case no busy-waits occur for the remaining threads.

B.4 Experimental Evaluation

The previous section presented a detailed description of a decentralized XMT microarchitecture. Next, we present a detailed simulation-based performance evaluation of XMT-M.

B.4.1 Experimental Framework

We have developed an XMT-M simulator for evaluating the performance potential of XMT-M and the XMT framework in general. This simulator uses the SimpleScalar ISA, with four new instructions added: a Spawn, a Join, and two Prefix-sums. The register set is also expanded to have both global and local registers. The simulator accepts XMT assembly code, and simulates its execution. All important features of the XMT-M system have been incorporated in the simulator.

Default XMT-M Configuration

- **Processor configuration**: Three different configurations: 2 clusters, 8 clusters, and 32 clusters.
- **Spawn-Join**: Dedicated PS unit for thread ID generation.
- **Prefix-sum (PS)**: Pipelined; 3 cycle latency after request received, total PS execution time is 3 cycles plus 2/8/32 cycle communication delay.
- **Cluster configuration**: 8 TCUs/cluster, 1 ALU (1 cycle latency); 1 Branch (1 cycle latency); 1 MultDiv (2 cycle multiply, 40 cycle divide,

neither is pipelined). 8 slot load-store buffer, 8 ports to L1 d-cache, 8 slot PS I/F.

- **TCU pipeline**: 6-stage single-issue in-order pipeline, stall on branch.
- **Cache configuration**: 4 word block, D-caches are 2-way set associative. I-caches are direct mapped. D-caches are write-through with no fetch on write-miss. On-chip L2 d-cache is 256K words. L1 d-caches are 16K words each. L2 i-cache is 1K words and L1 i-caches are 256 words each. Each L1 d-cache can send and receive 4 memory requests every C cycles (where C is the number of clusters), which roughly corresponds to a ring topology.
- **Main Memory**: 100-cycle latency, 8 words per cycle bandwidth. DRAM bank access conflicts are not modeled.

For these experiments, we model the memory hierarchy without specifying the exact interconnection structure, by setting a fixed latency for every memory request from a level-1 cache to get to the shared cache (level-2) and to the other level-1 caches. To justify this modeling, consider a ring topology for connecting the level-1 caches. Assume that at each node on the ring there is an automaton that will give priority to a request from another node. Each level-1 cache has a buffer to hold requests that originate from its local cluster until these requests can be sent. Therefore, once a request makes it out onto the ring, its maximum latency will depend on the time taken to go around the ring. In the worst case of all nodes sending requests all the time, each node will have to wait for its original request to come back around the ring before it can issue another request. Note that our fixed latency modeling is therefore a somewhat conservative estimate for the processing of requests from the level-1 caches. For a relatively low number of nodes on the ring (as in the designs we simulated), the ring does not perform too badly when we have a relatively long main memory access latency. However, for higher number of nodes and/or shorter memory latencies, a different design would be a better choice.

Default Superscalar Configuration

- **Processor configuration**: 16-way superscalar, out-of-order execution, 128-entry instruction window, 64-entry load-store queue, 16 integer ALUs, 16 mult units, 2048-entry bimodal branch predictor.
- **Cache configuration**: Has separate i-cache and d-cache, both of which have single cycle access and have 4 word blocks. D-cache is same as XMT-M's L2 d-cache. I-cache has 4 times as many blocks as XMT-M's L2 i-cache.

Benchmarks and Performance Metrics

For benchmarks, we use a collection of code snippets with varying degrees of inherent parallelism, as described in Figure B.7. We are currently restricted to using snippets in this paper, because no big application exists yet for the XMT programming paradigm. We believe that the performance on snippets provide a reasonable intuition into the performance and scalability of XMT-M. We have selected code snippets with varying degrees of inherent parallelism. Evaluating XMT-M's performance for entire applications can be done only after re-writing the entire applications in the XMT programming model and then compiling those parallel programs to XMT code. We recognize the importance of eventually carrying out studies with entire applications.

A compiler-like translation from high-level to optimized assembly code is done

Benchmarks	Description	Input sizes
Serial **linkedlist**	Traverse a linked list randomly dispersed through memory and find the sum of the list item data values. This application is not one which we know how to parallelize, so it is implemented with a serial algorithm.	50 item list spaced in 200 words, 500 in 2K words, 5K in 20K words, 250K in 1M words.
Embarrassingly parallel **stream**	Based on the STREAM benchmark [34], we sequentially read arrays, perform some short calculations on the values, and write the results to another array. Since each iteration of the loop is independent, parallelization of execution is obvious. In the superscalar domain, one approach for speeding up this code is loop unrolling; we do that for the SimpleScalar version.	50 item array, 500 item array, 5K item array, 250K item array.
Interacting parallel **arrcomp**	Compacting an array, we take a sparse array and rewrite into a compact form. This application requires keeping a running count of the next available location in the new array. Two variants of **arrcomp** were simulated: **arrcomp_d** which just reads the original array (regular memory access) and **arrcomp_i** which uses indirection through another array (irregular memory access).	50 item array, 500 item array, 5K item array, 250K item array (uncompacted arrays are 1/4 full).
More frequently interacting parallel **max**	Find the maximum value of a list. In the serial case, we read through the list, keeping a running maximum. For the parallel case we choose a synchronous max-finding scheme. A balanced binary tree is formed where a node of the tree will have the result of a maximum operation on its two child nodes. The root of the tree will have the maximum of the list. The algorithm proceeds from leaves to root, synchronizing after every level in the tree. The threads are very short and there are log(n) spawn-joins.	50 item array, 500 item array, 5K item array, 250K item array.
Sorting **listsort**	Unraveling a linked list of known length which is packed within an array. This is a version of the problem called “list-ranking”. This application is useful for managing linked-list free space in OSes [32]. In the serial algorithm, we traverse the list and rewrite it in the proper order. For the XMT version, we use two algorithms: (1) Wyllie’s pointer jumping algorithm [19] for the 50 and 500 sized inputs and (2) the no-cut coin-tossing algorithm for the 5K and 250K sized inputs. The work [5] presented discussion of the various list-ranking algorithms on XMT.	50 item array, 500 item array, 5K item array, 250K item array.
integersort	A variant of radix-sort. It sorts integers from a range of values by applying bin-sort in iterations for a smaller range. For speed-up evaluations, one would wish to compare integersort with the fastest serial sorting algorithms, and not only serial radix-sort, as we did; however, the literature implies that for some memory architectures radix-sort is fastest [2], while for others other sorting routines are fastest [25].	50 item array, 500 item array, 5K item array, 250K item array.

Figure B.7: Benchmarks

manually, for lack of an XMT compiler. To have a fair basis for comparison, the serial versions of the applications are also generated by hand, and are optimized by using techniques such as loop unrolling.

We use small input data sizes to illustrate that the XMT-M processor can achieve better performance for even small input sizes. While comparing the performance against superscalar processors, the metric used is speedup (obtained by dividing the number of execution cycles taken by the superscalar processor by the number of cycles taken by XMT-M).

B.4.2 XMT-M Performance

In the first set of experiments, we measure the number of cycles taken by different XMT-M configurations to execute the benchmarks, with varying size inputs. The number of cycles taken by the XMT-M configurations are compared against those taken by the default centralized wide-issue superscalar processor. Figure B.8 presents the results obtained. The figure consists of 4 diagrams, corresponding to 4 different input sizes. In each diagram, the X-axis represents the benchmarks. For each benchmark, 3 histograms are plotted, one for each XMT-M configuration (a 2-cluster XMT-M, an 8-cluster XMT-M, and a 32-cluster XMT-M). The Y-axis denotes the speedup of the XMT-M configurations over that of the default superscalar configuration. Notice that comparing the IPCs (instructions per cycle) for the two processors will not be meaningful, as they execute different programs.

Let us look at the results of Figure B.8 closely. For an input size of 50 (cf. the first diagram in Figure 8), the 8-cluster XMT-M configuration performs better than the other two XMT-M configurations. The 2-cluster XMT-M does not have enough parallel resources to harness inter-thread parallelism; and not much inter-thread parallelism is available with an input size of 50 to compensate for the high latencies of the 32-cluster XMT-M. The 8-cluster performs substantially better than the centralized superscalar processor for three of the benchmarks, and performs slightly worse than the centralized superscalar processor for three of the benchmarks.

When the input size is increased to 500 (cf. the second diagram in Figure B.8), the 32-cluster XMT-M (despite its increased cross-chip latency) begins outperforming the 8-cluster XMT-M for most of the benchmarks, because of the increased inter-thread parallelism available. Even the 2-cluster XMT-M configuration outperforms the centralized superscalar processor in all but one of the benchmarks.

When the input size is increased beyond 500, the XMT-M configurations continue to harness more parallelism, as might be expected. It is important to point out that these results have to be analyzed in the proper context. The XMT-M configurations are performing **in-order execution** and **single-instruction issue** in each TCU. Thus, the TCUs in an XMT-M processor are not performing functions such as branch prediction, dynamic scheduling, register renaming, memory address disambiguation, etc. In short, the **XMT-M TCUs are not exploiting any intra-thread parallelism**, except for the overlap obtained in a 6-stage pipeline. This is very important. First, it suggests that our speedup results are conservative. Second, in the future as clock cycles continue to decrease, it becomes more and more difficult to perform centralized tasks such as dynamic scheduling and branch prediction within a limited cycle time. We would also like to point out that the XMT framework does not preclude the use of conventional techniques to extract intra-thread parallelism.

B.4.3 Effect of Cross-chip Communication Latency on XMT-M Performance

Our next set of experiments focus on studying the effect of global (cross-chip) communication latency on XMT-M performance. Cross-chip communication refers to prefix-sum operations, spawn/join operations, global register accesses, and L1 cache accesses. Three different latencies were modeled for one-way inter-cluster communication: 1, 4, and 16 clock cycles. This corresponds to latencies of 2, 8, and 32 cycles, respectively, for functions that require two-way communication.

Figure B.9 gives the results obtained in this study. Again, four diagrams are given, corresponding to four different input sizes. Each diagram records three histogram bars for each benchmark. Thus, the X-axis represents the 7 benchmarks, and for each benchmark the three inter-cluster communication latencies. The Y-axis represents the normalized performance; normalization is done with respect to the performance of the unit-latency configuration. That is, the height of the bar represents the value obtained by dividing the execution time of the benchmark for a latency of 1 cycle by the execution time of the benchmark for the corresponding latency.

The results of Figure B.9 indicate that for all input sizes considered, the performance of XMT-M does not change when the inter-cluster communication latency is increased from 1 to 4. Even when this latency is increased to 16 cycles, XMT-M's performance remains the same except for the two `arrcomp` benchmarks, for which there is a drop of about 5%. These results indicate that XMT-M is somewhat resilient to increased cross-chip interconnect delays.

B.5 Related Work

First of all, we should emphasize that we have not "invented parallel computing" with XMT. We have tried to build on available technologies to the extent possible.

The relaxation in the synchrony of PRAM algorithms is related to the works of [10] and [20] on asynchronous PRAMs. The high-level language we used for

XMT builds on Fork95 and its previous versions developed at the U. Saarbrucken, Germany, see [28] and [29]. Basic insights concerning the use of a prefix-sum like primitive go back to the Fetch-and-Add [21] or Fetch-and-Increment [18] primitives (cf. [2]). Insights concerning nested Spawns rely on the work of Guy Blelloch's group [6], [7] and others. The U. Wisconsin Multiscalar project [17] and the U. Washington simultaneous multi-threading (SMT) project [54] with their use of multiple program counters and the computer architecture literature on multi-threading (see, for instance [25]) have also been very useful; however, the way XMT proposes to attack the completion time of a single task, which is so central to XMT, makes XMT drastically different than these approaches; that is, the reliance on PRAM algorithms. Our experience has been that some knowledge of PRAM algorithms is a necessary condition for appreciating how big the difference is.

Simultaneous multi-threading [54] improves throughput by issuing instructions from several concurrently executing threads to multiple functional units each cycle. However, SMT does not appear to contribute towards designing a machine that look to the programmer like a PRAM [34].

The similarity of XMT to the pioneering Tera Multi-Threaded architecture [3] is very limited. Tera focuses on supporting a plurality of threads by constantly switching among threads; it does not issue instructions from more than one thread at the same cycle; this, in turn, would limit the relevance of our multi-operand and Spawn instructions for their architecture. Tera's multiprocessor was engineered to hide long latencies to memories for big applications. Its design aims at 256 processors each running 128 threads. XMT, on the other hand, is designed to provide competitive performance for even small input sizes, which makes it more practical, in general, and for desktop applications, in particular. To explain this, we observe that higher bandwidth and lower latencies, which are expected from on-chip designs in the billion transistor era, will allow a parallel algorithm to become competitive with its serial counterpart for a much smaller input size than for MPP paradigms such as Tera. But, why does this observation hold true? Parallelism provided by a parallel algorithm increases as the input size increases; now, when implemented on an MPP, part of this algorithm parallelism is used for hiding system deficiencies (such as latency); when latency is a minor problem, more algorithm parallelism can be applied directly to speed-ups. So, parallel algorithms can become competitive for *much smaller inputs*.

Another strong argument in favor of XMT is that it is anticipated that *explicit parallelism will provide for simpler hardware*. Explicit ILP generally means "static" extraction of ILP, which allows for simpler hardware than that for dynamic (i.e., by hardware) extraction of ILP. Stating simpler hardware as the main motivation, industry has demonstrated its interest in explicit instruction-level parallelism (ILP), by way of heavily investing in it.

Lee and DeVries investigate single-chip vector microprocessors as a way to exploit more parallelism with less hardware and reduced instruction bandwidth [33]. They expose more instruction-level parallelism to the processor core by moving to a more explicitly parallel programming model. Similarly, the IRAM project uses vector processing to increase parallelism; the project also aims to fully exploit the bandwidth possibilities of integrating DRAM onto the microprocessor [30]. Complementing this research, out-of-order, multi-threaded, and decoupled vector architectures have been proposed by Espasa, Mateo, and Smith [15] [16] as methods to improve the performance of vector processing. The XMT architecture has much in common with the recent vector approaches, as it is solving many of the same problems in much the same way; the primary difference is in the use of a SPMD-style parallel algorithm programming model instead of a vector model.

B.6 Concluding Remarks

The "von Neumann architecture" has provided a principled engineering design point for computing systems for over half a century. It has been very robust and resilient, withstanding dramatic changes in all the relevant technologies. Parallel computing has long been considered an antithesis to the von Neumann architecture. However, the main success thus far in using parallelism for reducing completion time of a single general-purpose task has been accomplished by what we called earlier the ILP bridgehead - yet another von Neumann architecture! However, technology changes due to the evolving sub-micron technology are making it increasingly difficult to extract parallelism by conventional ILP techniques.

In the past 20 years, the increased transistor budget of processor chips was applied to close the gap between microprocessors and high-end CPUs. As pointed out in [12], today this gap is fully closed and using the additional transistor budget for uniprocessors is well beyond the point of diminishing returns. It is becoming increasingly important, therefore, to tap into explicit parallel processing. The XMT approach tries to jump into filling this gap; the "no-busy-waits finite-state-machine" principle with its implied independence of order semantics (IOS) are designed to directly address the profound weakness of continued evolutionary development of the von Neumann approach.

The [56] paper introduced the broad XMT framework through few "bridging models" (or internal interfaces). This paper contributes a first microarchitecture implementation — a significant step for the overall XMT effort. The research area of XMT (and SPMD in general) as it connects with parallel algorithms offers an interesting design point: these algorithms map well to hardware support for multiple independent concurrent threads. This hardware support, in turn, maps well to the limitations of future sub-micron technologies—interconnect delays dominating the performance—which necessitate most of the communication to be localized. The XMT-M implementation highlights an important strength of XMT: it lends itself to *decentralized implementation* with almost no degradation in performance. An XMT-M processor can comprise numerous, simple, independent, identical thread execution units, and the nature of the programming paradigm is such that inter-thread communication is highly structured and regular. This paper shows that such a microarchitecture can withstand high cross-chip communication delays. It can also take advantage of a large number of functional units, as opposed to traditional superscalar designs, which typically cannot make use of more than one or two dozen functional units.

XMT-M integrates several well-understood and widely-used programming primitives that are usually implemented in software; the novelty of the microarchitecture is the integration of these primitives in a single-chip environment, which offers increased communication bandwidth and significantly decreased communication latency compared to more traditional parallel architectures. The integrated primitives are the *spawn-join* mechanism, which enables parallelism by initiating and terminating the concurrent execution of multiple threads of control, and the *prefix-sum* operation, which is used to coordinate the threads.

Finally, we note that XMT-M has achieved significant speedups by extracting only inter-thread parallelism (and no intra-thread parallelism), and that it can get additional benefit from extracting intra-thread parallelism with the use of standard ILP techniques.

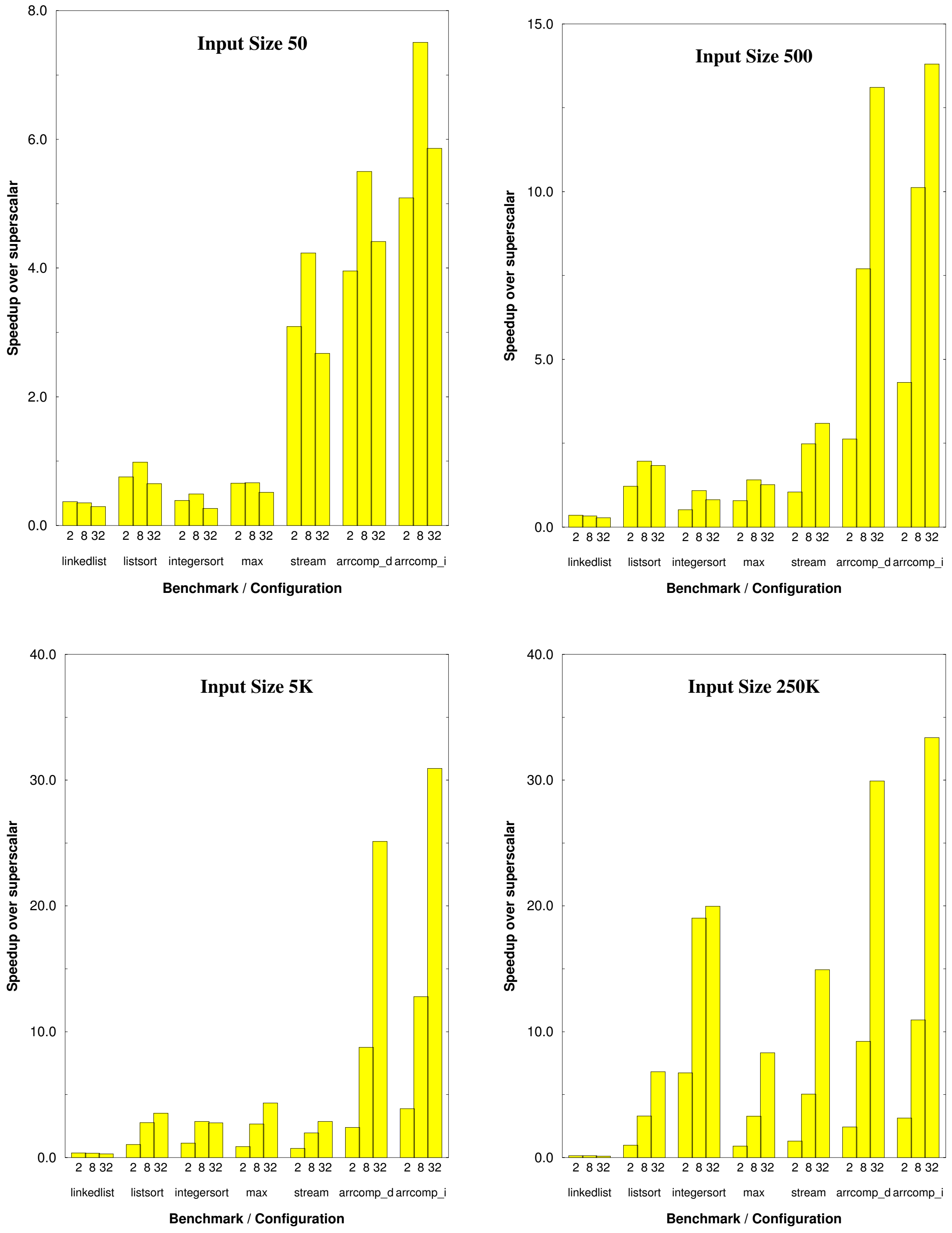


Figure B.8: XMT-M Speedups relative to serial computing

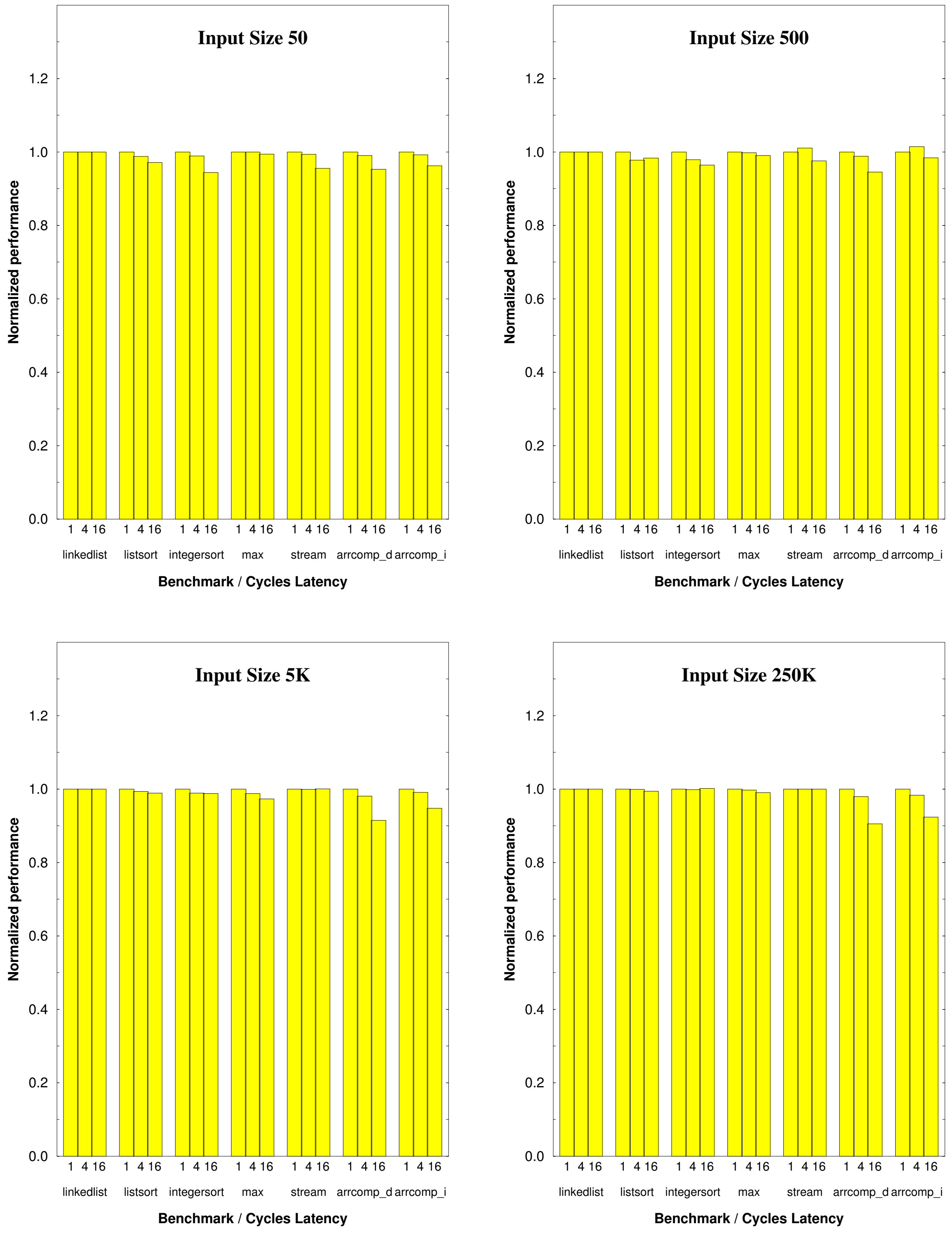


Figure B.9: Effect of cross-chip (global) communication latency on XMT-M performance

Appendix C

Synchronous Crossbar Interconnect

This appendix details an investigation into using a synchronous crossbar as the on-chip interconnect for an XMT system. This approach was abandoned in favor of the approach of Chapter 4, however it provides useful background.

C.1 A new approach to parallel memories

C.1.1 Modeling parallel memories

We wish to come up with a suitable model for a memory subsystem. We consider a system with several parallel processing elements, which access a shared memory space. We will call these processing elements PEs, and consider them to be individual modules which produce requests (reads and writes) which need to be serviced by memory.

We now make some choices about the memory model. The first is to break up the memory into modules. It is clear that the alternative (having a unified memory module) would not scale, as memory speed decreases and cost increases with increasing memory size and number of ports. The second choice is to partition the memory space rigidly among the different memory modules. This serves to eliminate coherence issues, as any given address can only reside in a single memory module. However, this also precludes caching of arbitrary address near each individual PEs. We choose to sacrifice locality in favor of high bandwidth at this point.

We will call the memory modules MMs. An individual MM is modeled as a module able to handle requests from its exclusive set of addresses. Consider a system with several PEs and several MMs. To encourage efficient implementation, we will further restrict each PE to performing one read or write at a time (multiple requests may be queued) and each MM to handling only one request at a time. Based on these specifications, it becomes clear what the ideal interconnection of these nodes should look like. Every PE must be able to connect to any MM, and any MM must be accessible from any PE. Interconnect should allow arbitrary connections of PEs to MMs, without the need to handle collisions. This is exactly the high level description of a crossbar network. With four PEs and four MMs, such a network might look conceptually like:

```
    M M M M
    M M M M
    | | | |
PE--+-+-+-+
PE--+-+-+-+
PE--+-+-+-+
PE--+-+-+-+
```

where "+" represents a cross point, and only one crosspoint should be active in any given row and any given column.

Two things are left to be determined. The first is how to schedule the operation of the interconnect so that PEs connect to the modules they need to talk to. The second is how to distribute the addresses to the memory modules so that PEs vie for the same MM as little as possible.

C.1.2 Parallel coordination

We wish to solve the problem of how to coordinate the parallel scheduling of the interconnection of p PEs and m MMs.

Ideally, the interconnect would operate in such a way that requests are handled in an optimum way to maximize throughput and minimize delays. Such a system would seem to require that the PEs communicate their requests to a centralized controller who would analyze the requests and then orchestrate the optimum schedule.

However, in a reasonably large system this controller would need to quickly process a lot of information, and be able to respond to all PEs across the system so that each knows when to request what. By nature, it seems contrary to the idea of a low-overhead, decentralized parallel system.

Simplistic scheme

The trivial alternative would be to set a cyclic, fixed schedule, where a given PE would have access to a given MM exactly once per cycle (a cycle would consist of max(p,m) steps). In our 4 by 4 example, the network might cycle through the permutations (1,2,3,4), (2,3,4,1), (3,4,1,2), and (4,1,2,3). Our goal is to achieve better performance by designing a more adaptable network.

All-to-all request/acknowledge

Consider a system where every PE would have m communications lines, one connected to each of the MMs. Each MM, then, would have p communications lines coming in to it. A particular PE could simultaneously request access to all the MMs for which it has pending requests by asserting the corresponding lines. All p PEs could make their requests at the same time.

Each MM would get a signal from every PE that wishes to talk to it. Since a MM can only handle one request at a time, it should choose just one of the requestors and send an acknowledge signal down its line. If a PE gets an acknowledge signal, it knows it has exclusive access to that MM, and sets the network to make the connection. Note that a PE may receive more that one acknowledge. In such a situation, it must pick only one, and the unpicked MMs will sit idle. This solution completely removes the idea of a centralized controller. The different nodes negotiate among themselves to arbitrate access.

Generalization using prefix-sum

With the above method, the modules must negotiate with a request/acknowledge cycle for every transfer. If there are many pending requests, such a scheme might cost a lot of overhead, especially if the system can't perform the scheduling negotiation at the same time data is being transferred.

It may be advantageous for a given memory module to take all simultaneous requests from PEs and schedule them to unique time slots over the next few cycles. Given p values of either 1 (PE_i wants to speak to this MM) or 0 (PE_i does not), the MM wishes to assign a unique cycle to all the 1-valued inputs. This is simply a prefix-sum of the p inputs on a base with value 0. The assigned slots (the results of the prefix sum) could be communicated to the PEs serially back across the 1-bit all-to-all network. If we allow for up to n slots to be scheduled at once, this would require lg(n) bits.

Scheduling issues

Similar to the 1-slot request/acknowledge scheme, if a PE is granted access to multiple MMs in a given time slot, it must choose only one, and the others will be idle during that slot. If the MMs all assign slots starting with 0, the early cycles will be over-scheduled, exacerbating the conflict issue. To alleviate this problem, different MMs could add different offsets (mod n) to the prefix-sum results.

Another concern is the issue of fairness. We likely do not want a PE to be starved if it consistently doesn't make the slot cutoff at a particular MM. This could be fixed by implementing a rotating priority among the PEs.

Adaptive scheduling scheme

The best scheduling scheme could very well depends on the number of requests pending at a given time. If the memory requests are fairly sparse, the 1-slot request-acknowledge scheme may provide the lowest-latency responses. However, if all the PEs are saturated with several requests.

We would like the system to be able to adapt to changing memory traffic conditions. However, all PEs and MMs must act in concert, using the same number of slots. We will see in the design study section a method whereby, individual MMs will track their slot utilization, and then vote to increase or decrease the number of slots per scheduling cycle.

C.1.3 Parallel distribution

We are exploring several possible ways to distribute addresses among the MMs, including a couple interesting randomized hashing schemes. Since the interconnect implementation is largely independent of the distribution scheme, we do not discuss the hashing here.

We also have ideas about how to handle multiple accesses to the same address, which are not independent of the interconnect implementation, but we suppress this issue here.

C.2 Implementation considerations

C.2.1 Interconnection network design

As we mentioned earlier, the most natural interconnect for our model is a crossbar network. If a crossbar can be designed with good area and delay costs, then we do not need to use a less-capable multi-stage network.

The simplest crossbar implementation is simply an array of switches. Each PE port connects to one end of a group of w horizontal wires. (where w is the width of the crossbar ports) Similarly, each MM port connects to a crossing group of w vertical wires. There is a transmission gate switch connecting horizontal and vertical lines at each cross point. A decoder at each PE selects which switches will be activated. The transmission gates allow the crossbar to be used for bidirectional communication.

An alternative implementation differentiates between writer and reader. Each writer drives a fixed set of w horizontal lines. Each reader selects which horizontal line to read from. The selection is performed by a tree of 2-to-1 muxes.

The area costs for both approaches are similar. In either design, the area is (very) roughly proportional to $M * N * W^2$, where M is the number of input ports, N is the number of output ports, and W is the width of each port. (This approximation is based on wire cost, for smaller designs, the device cost might dominate, however we wish to consider larger designs.)

Dutta [14] found the delay of the transmission gate crossbar to be competitive only for configurations with very small numbers of ports. In the mux-based crossbar, the buffered 2-to-1 trees insure light loading for quick propagation of values.

If the network delay is sufficiently large, it could be advantageous to pipeline the interconnect to improve throughput. The mux nodes provide a natural place to add up to $\lg(M)$ latches for pipelining. It is less obvious how to effectively pipeline a transmission gate design.

Note that we can reduce area and delay by reducing the number of ports or the port width. The port width can be reduced below the width of data to be transmitted by sending a series of smaller packets. Such a scheme might have the further advantage of being able to quickly pipeline several packets, without having to pay the cost of reconfiguring the network.

C.2.2 All-to-all connection

For our scheduling negotiation scheme, we'd like a 1-bit (at least) connection from all PEs to all MMs. We could statically wire all the necessary connections. Such an interconnect would require area proportional to $M^2 * N^2$. Note that if M, N, and W are of similar size, than the costs of the crossbar and of the all-to-all network are similar. In an advanced fabrication process with several metal layers, it may become possible to layer an all-wire network over a switched crossbar.

A cheaper alternative might be to take advantage of the hardware already present in the crossbar. Assume, for the moment, that $M = N = W$. A crossbar for such a system has all the wires we need for the all-to-all connection, the switches just need to be set properly. We can add a special mode to the crossbar control logic, where the individual bits from a given port no longer switch together, but instead connect to separate (predefined) output wires. We cannot operate both functions simultaneously, but with minimal cost we can support both with the same hardware.

We can generalize this idea to configurations where $M \mathrel{!=} N \mathrel{!=} W$ by allowing for time-multiplexing of the 1 bit signals, if necessary.

C.2.3 Prefix-sum

We have studied efficient implementation of the prefix-sum mechanism. We do not discuss it here but simply say that a 64 bit prefix-sum with 1-bit increments should perform similarly to a 64-bit parallel-prefix adder.

C.3 Design study

Since an on-chip memory interconnect of this type has yet to be constructed in a real system, it is important to verify that this nonlinearly-scaling structure will be implementable at the scale we may wish to deploy in the near-future VLSI domain. To that end, we present a design study in which we push a specific design all the way to the VLSI layout level. This section represents the primary experimental contribution of this project.

C.3.1 Specific choices

We first make several choices to guide design. We will attempt to design the interconnect to handle 64 PEs, 64 MMs, and 64-bit wide ports. This configuration was chosen because of its symmetry, and because these aggressive numbers represent what we expect is somewhere close to the limits for our current design ideas.

The VLSI components will be designed using the MOSIS Scalable CMOS design rules. While any real submicron process will differ in its requirements, we hope that SCMOS will provide a reasonable approximation while allowing us to consider different technology sizes. We stick to a three metal-layer design.

We will attempt to implement the dual-mode crossbar described earlier. For simplicity, we will not try to include any sort of pipelining.

Recall that between the two discussed crossbar designs, only the mux-based exhibited acceptable delays with many ports. For our purposes, this is the clear winner.

C.3.2 Unidirectional interconnect - Pairing of nodes

Since the mux-based crossbar incorporated buffers at each mux node, it is an inherently unidirectional design. Since both PEs and MMs both read and write, we must allow for bidirectional capability. In Dutta's Video Signal Processor design, he uses a folded crossbar, where each port consists of an input port and an output port next to each other. To give each PE and each MM in our design an in and an out port would require 128 ports.

However, this solves a tougher problem than we need to. The connectivity graph in our design is bipartite; PEs need only talk to MMs and MMs need only talk to PEs.

If we make a slight tweak to our prefix-sum based scheduling scheme, we create a situation where, at any given time, only PEs are writing and only MMs reading or vice versa. We need only separate out memory read requests and memory write requests. With this distinction, it is natural to suggest placing both a MM and a PE at each pair of in/out ports. The MM and PE can then share the resource without conflict.

As an aside, we note that this design choice now enables the voting scheme alluded to earlier. A particular MM can simultaneously broadcast to all MMs a 1-bit vote to either increase or decrease the number of scheduled slots. Each MM can then independently tally the votes (easily done with the prefix-sum hardware already sitting at each output port) and then act according to some predefined threshold.

C.3.3 Building block: Mux node

Dutta's crossbar design uses vertically laid out trees of 2-to-1 mux nodes to select among N wires. We first attempt a similar design. The primary building block is a buffered mux with two inputs and one output on the same side.

Our design for this component appears in Figure C.1.

C.3.4 For a better aspect ratio: Triple mux node

When taking Dutta's design and extrapolating it to 64 by 64 by 64, a disturbing picture emerges. A vertical mux tree selects from among 64 wires, one for each

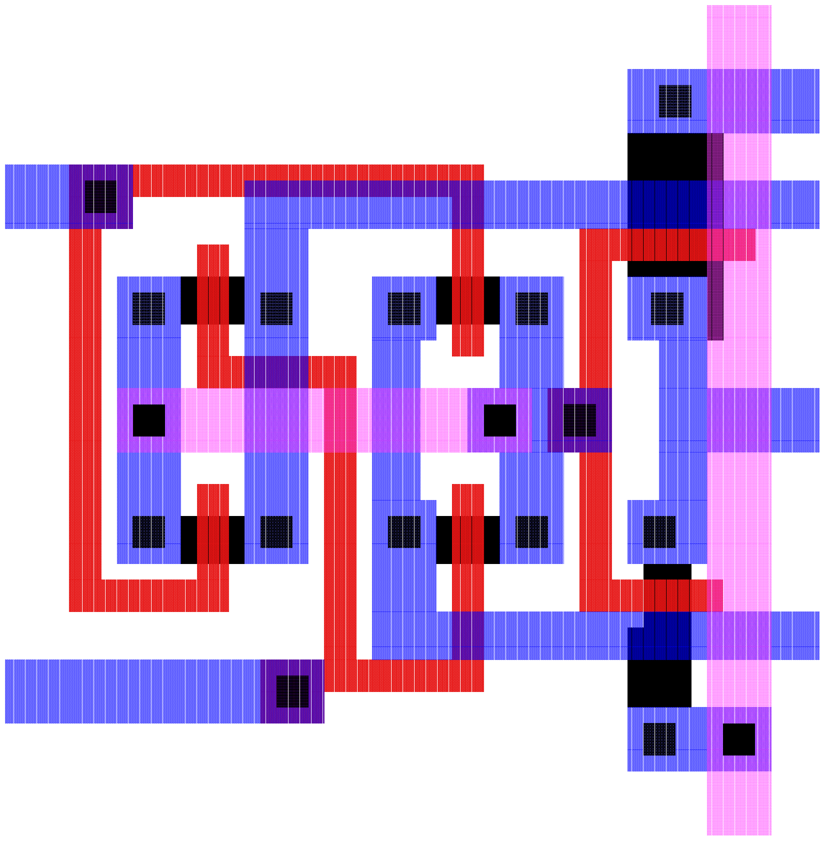


Figure C.1: Mux node VLSI layout

possible writer. We must replicate 64 of these trees one over the other to get the 64 bit width. The result is a design that is very much taller than it is wide. This implies overly-long wire lengths, and is also troublesome since all in/out ports (where all PEs and MMs sit) are along the top and bottom.

We note that this design under-utilizes the vertical space. Each mux node only selects from among two horizontal wires, however up to 5 stacked wires could fit within the vertical dimensions of the mux node. This suggests to use two mux nodes in a single vertical slot to select from two wires each, and a third mux node to select from the two. We then connect the rows of three together with a vertical tree as before. The result is that the mux tree moves from a 63x1 node rectangle to a 16x4 node rectangle. Now the horizontal data lines are packed almost as tightly as possible.

An efficient implementation of the triple mux node appears in Figure C.2. These can be efficiently stacked, as in Figure C.3. Here, the horizontal wires are packed as tightly as possible, with groups of four lines alternating with a power line. To select a single bit in our design, the 8 rows of Figure C.3 would appear above another set of 8 rows, with a single output line passing between. The other 15 mux nodes would appear in a single column to the left, with the middle node driving the output line. We need 8 control lines to the left of the mux tree to provide 4 complemented controls to the mux tree. We need two additional control lines to provided a complemented ”switch to alternate mode” signal and two additional specially wired signals to assist in the alternate mode implementation.

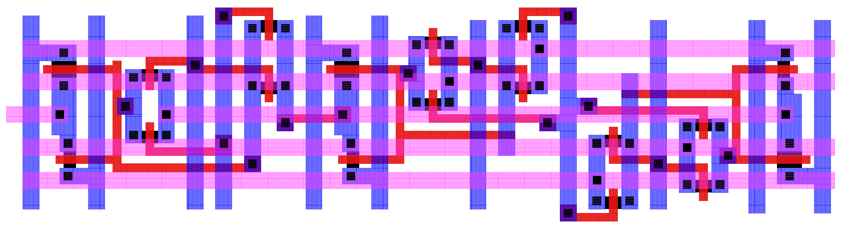


Figure C.2: Triple mux node VLSI layout

C.3.5 Big picture

The system floor plan would have 32 PE/MM pairs along the top and 32 along the bottom. A PE/MM pair on the top shares its 128 bit in/out vertical data channel with the corresponding pair on the bottom using the folding mechanism. One pair will control a mux tree structure on the left, and the other will control a mirror image structure on the right. The structures will be replicated 64 times on top of each other to fill out the 64 bits.

C.3.6 Area analysis

We are now in a position to calculate the area requirements for the interconnect. For flexibility we will parameterize the equations by λ (the technology scaling

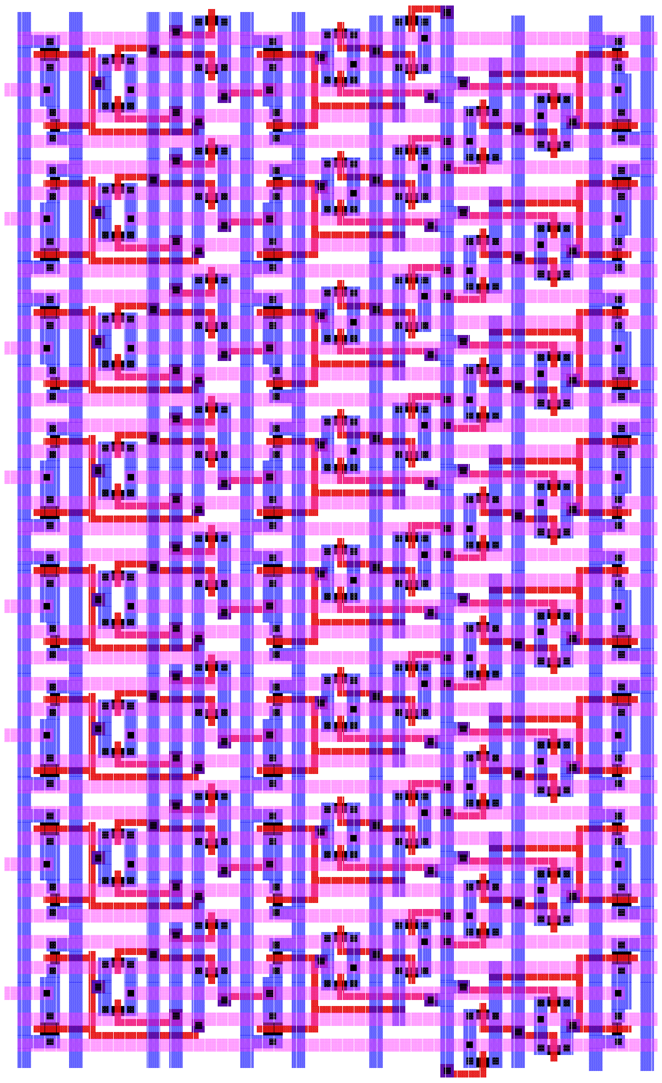


Figure C.3: Stacked triple mux nodes VLSI layout

factor), N (the number of PEs and MMs, which we will force to be equal), and W (the width of the ports).

For the vertical dimension, everything is packed such that the size is determined solely by wire spacing.

$$\begin{aligned} y &= (N/4 + N + 1) * W * (8 * \lambda) \\ &= (10 * N + 8) * W * \lambda \end{aligned}$$

For the horizontal dimension, the width of the vertical mux tree nodes ($vnode_x$) as well as the width of the horizontal triple mux nodes ($hnode_x$).

$$vnode_x = 48 * \lambda$$

$$hnode_x = 198 * \lambda$$

$$\begin{aligned} x = (2 + \lg(N) - 2 + \lg(N) - 3 + W) * N * (8 * \lambda) + N * (vnode_x + hnode_x) \\ = (8 * W + 16 * \lg(N) - 24) * N * \lambda + 246 * N * \lambda \\ = (8 * W + 16 * \lg(N) + 222) * N * \lambda \end{aligned}$$

For the case of $N = W = 64$,

$$y = 41472\lambda$$

$$x = 53120\lambda$$

which is a good ratio. The area for a 0.15 micron process ($\lambda = 0.075 * 10^{-3}$ mm) would be:

$$A = 12.4 \text{ mm}^2$$

The area of a 2 cm x 2 cm chip is 400 mm^2, so this certainly seems like a reasonable price to pay for such an important system component.

C.4 Integration with XMT

We now take a moment to briefly describe some of the implications of our design in the XMT context.

C.4.1 Cluster corresponds to PE?

It is obvious that there should be some connection between the abstract idea of PE and the independent Tread Control Units of the XMT architecture. It would seem sensible to continue to cluster TCUs together to maximize interconnect port utilization. Each cluster would then correspond to what we have called a PE.

C.4.2 General prefix-sum implementation

One of the most exciting parts of this design is how it might fit with the implementation of prefix-sums in a XMT system. We need only step back a bit from our memory subsystem design to realize that almost all the machinery is there to provide a distributed prefix-sum implementation. One could assign each port to handle prefix-sum to a particular base with the hardware already there. With this idea, multiple simultaneous prefix-sums to many different bases becomes very easy. The centralized prefix-sum unit described in previous work can be completely eliminated.

C.4.3 Spawn implementation

The crossbar design we describe allows for one-to-many broadcasting. Recall that each port (shared between a single TCU and a single memory module) can be set to read from any source port. In serial mode, the inactive TCUs could all select to read from the same TCU, TCU 0. The serial thread could then broadcast global register updates as they occur. Additionally a spawn could be broadcast immediately to all TCUs, perhaps also including the first few instructions to begin executing.

Appendix D

Hardware Implementation of Prefix-Sum

A hardware implementation, due to Uzi Vishkin, for small-integer prefix-sum, as presented in [55] follows. It is included here for convenience.

Generally, one would expect each (single) prefix-sum to need an adder. see e.g. [2]. In this section we show that hardware cost can be drastically reduced, in one case where the input values for a multiple prefix-sum are limited.

We say that a multiple prefix-sum instruction

$PS\ R_0\ R_1\ PS\ R_0\ R_2\ \ldots\ PS\ R_0\ R_k$

is *small-integer* if each input value in registers R_1 through R_k can only be a small integer; for example, integers between 0 and 3. No limits are placed on the input value in the base register R_0.

The number of individual prefix-sum instructions that can be combined into a compound prefix-sum hardware implementation should certainly be machine dependent. In case this number is $h(< k)$ we will first perform the first h individual prefix-sum instructions, then the next such h instructions and so on till the full k-wide multiple prefix-sum has been executed.

The proposed hardware implementation will demonstrate that a small-integer prefix-sum may not need much more hardware resources than binary integer addition. This, together with demonstration of usability, may support the case for implementing compound prefix-sum in hardware, or even adding one or more compound prefix-sum functional units to standard processors.

We proceed to outline an implementation of the multiple small-integer prefix-sum instruction:

$PS\ R_0\ R_1$

$PS\ R_0\ R_2$

...

$PS\ R_0\ R_h$

into

$z_0\ =\ R_0$

$z_1\ =\ R_0\ +\ R_1$

$z_2\ =\ R_0\ +\ R_1\ +\ R_2$

...

$z_h\ =\ R_0\ +\ R_1\ +\ R_2 + ... + R_h$

1. Compute the base-zero prefix-sum, using dedicated hardware. By base-zero we mean that the input values of registers R_1 through R$_h$ are taken as they are, but the base register R_0 is taken as 0. Store these prefix-sums in an intermediate array V of h elements $v_1, v_2\ ...\ v_h$.

2. Compute the following "suffix sum", using dedicated hardware into an intermediate array U of h elements $u_1, u_2\ ...\ u_{h-1}$.

$u_{h-1}\ =\ R_h$

$u_{h-2} \ = \ R_{h-1} \ + \ R_h$

...

$u_1 \ = \ R_2 \ + \ ... \ + \ R_h$

3. Compute the last output element z_h by adding the last intermediate element v_h to the original input base R_0, using an adder.

Overview of the rest of the implementation. The rest of the output elements, z_1 through z_{h-1}, are found by using arithmetic only on a small number of bits, either by additions of elements of the intermediate array V to the input element R_0 or by subtractions of elements of intermediate array U from z_h. These additions and subtractions are based on first determining the index, i, of the most significant bit at which R_0 and z_h differ, and constructing an auxiliary number, w, which has one in its i-th bit and zeroes in all bits less significant than the i-th bit, and is identical to both R_0 and z_h, for more significant bits than the i-th bit. The remaining steps of the hardware implementation are as follows:

4. Find index i as follows: Apply exclusive-or bitwise to the binary representation of R_0 and z_h. i is the index of the most significant bit for which the exclusive-or yields one.

5. Derive w.

6. Get the values of the rest of the output which are smaller than w by adding the low-index elements of the intermediate array V to R_0.

7. Get the values of the rest of the output which are equal to or greater than w by subtracting the high-index elements of the intermediate array U from z_h.

Because h is small, and all the elements in R_1 through R_h are small, the additions and subtractions of the last two steps involve only short carries. They can be implemented using short look-ahead carry generators (for look-ahead carry generators, see, e.g., [35]).

BIBLIOGRAPHY

[1] R.C. Agarwal, "A Super Scalar Sort Algorithm for RISC Processors," *Proc. ACM SIGMOD*, 1996.

[2] G.S. Almasi and A. Gottlieb, *Highly Parallel Computing*, second edition, Benjamin/Cummings, Redwood City, CA, 1994.

[3] R. Alverson, D. Callahan, D. Cummings, B. Koblenz, A. Porterfield, and B. Smith, "The Tera Computer System," *Proc. International Conference on Supercomputing*, pp. 1–6, 1990.

[4] E. Berkovich, "An Explicit Multi-Threaded Architecture and Directed Acyclic Graphs," Masters thesis, Department of Electrical and Computer Engineering, University of Maryland, 1998.

[5] E. Berkovich, J. Nuzman, M. Franklin, B. Jacob, and U. Vishkin, "XMT-M: A Scalable Decentralized Processor," UMIACS TR 99-55, University of Maryland, September 1999.

[6] G.E. Blelloch, *Vector Models for Data-Parallel Computing*, MIT Press, 1990.

[7] G.E. Blelloch, S. Chatterjee, J.C. Hardwick, J. Sipelstein, and M. Zagha, "Implementation of a portable nested data-parallel language," *Proc. 4th ACM PPOPP*, pp. 102-111, 1993.

[8] D. Burger and T.M. Austin, "The SimpleScalar Tool Set, Version 2.0," Tech. Report CS-1342, University of Wisconsin-Madison, June 1997.

[9] J.-H. Chern, J. Huang, L. Arledge, P.-C. Li, P. Yang, "Multilevel Metal Capacitance Models For CAD Design Synthesis Systems," *IEEE Electron Device Letters*, Vol. 13, No. 1, pp. 32–34, January 1992.

[10] R. Cole and O. Zajicek, "The APRAM: Incorporating Asynchrony into the PRAM Model," *Proc. 1st ACM Symposium on Parallel Algorithms and Architectures*, pp. 169–178, 1989.

[11] D.E. Culler and J.P. Singh, *Parallel Computer Architecture*, Morgan Kaufmann, San Mateo, CA, 1999.

[12] W.J. Dally and S. Lacy, "VLSI Architecture: Past, Present, and Future," *Proc. Adv. Research in VLSI Conf.*, 1999.

[13] S. Dascal and U. Vishkin, "Experiments with List Ranking on Explicit Multi-Threaded (XMT) Instruction Parallelism," *Proc. 3rd Workshop on Algorithms Engineering (WAE-99)*, London, pp. 43–59, July 1999. Extended version in *ACM Journal of Experimental Algorithmics*, 5, article 10, 2000.

[14] S. Dutta, "VLSI Issues and Architectural Trade-Offs in Advanced Video Signal Processors," Doctoral dissertation, Department of Electrical Engineering, Princeton University, 1996.

[15] R. Espasa and M. Valero, "Multithreaded Vector Architectures," *Proc. Third International Symposium on High Performance Computer Architecture (HPCA-3)*, pp. 237–248, 1997.

[16] R. Espasa, M. Valero, and J.E. Smith, "Out-of-order Vector Architectures," *Proc. 30th Annual International Symposium on Microarchitecture (MICRO-30)*, pp. 160–170, 1997.

[17] M. Franklin, "The Multiscalar Architecture," Ph.D. thesis, Technical Report TR 1196, Computer Sciences Department, University of Wisconsin-Madison, December 1993.

[18] E. Freudenthal and A. Gottlieb, "Process Coordination with Fetch-and-Increment," *Proc. Fourth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-IV)*, pp. 260–268, 1991.

[19] M. Frigo, C. Leiserson, K. Randall, "The Implementation of the Cilk-5 Multithreaded Language," *Proc. 1998 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pp. 212–223, 1998.

[20] P.B. Gibbons, "A more practical PRAM algorithm," *Proc. 1st ACM Symposium on Parallel Algorithms and Architectures*, pp. 158–168, 1989.

[21] A. Gottlieb, B. Lubachevksy, and L. Rudolph, "Basic techniques for the efficient coordination of large numbers of cooperating sequential processors," *ACM Transactions on Programming Languages and Systems*, 5, 2, pp. 105–111, 1983.

[22] L. Hammond, B. Nayfeh, and K. Olukotun, "A Single-Chip Multiprocessor," *IEEE Computer*, 30:79–85, September 1997.

[23] L. Hammond, B.A. Hubbert, M. Siu, M.K. Prabhu, M. Chen, and K. Olukotun, "The Stanford Hydra CMP," *IEEE MICRO Magazine*, March–April 2000, pp. 71–84.

[24] J.L. Hennessy and D.A. Patterson, *Computer Architecture: A Quantitative Approach*, second edition, Morgan Kaufmann, San Francisco, CA, 1996.

[25] R.A. Iannucci, G.R. Gao, R.H. Halstead, and B. Smith (editors), *Multithreaded Computer Architecture - A Summary of the State of the Art*, Kluwer, Boston, MA, 1994.

[26] J. JáJá, *An Introduction to Parallel Algorithms*, Addison-Wesley, Reading, MA, 1992.

[27] R. Joy and K. Kennedy, *President's Information Technology Advisory Committee (PITAC) — Interim Report to the President*, National Coordination Office for Computing, Information and Communication, 4201 Wilson Blvd, Suite 690, Arlington, VA 22230, August 10, 1998.

[28] C.W. Keßler and H. Seidl, "Integrating synchronous and asynchronous paradigms: the Fork95 parallel programming language," Technical report no. 95-05, Fachbereich 4 Informatik, Universität Trier, D-54286 Trier, Germany, 1995.

[29] C.W. Keßler, "Quick reference guides: (i) Fork95, and (ii) SB-PRAM: Instruction set simulator system software," Fachbereich 4 Informatik, Universität Trier, D-54286 Trier, Germany, May 1996.

[30] C. Kozyrakis, *et al.*, "Scalable Processors in the Billion-Transistor Era: IRAM," *IEEE Computer*, Vol. 30, pp. 75–78, September 1997.

[31] M.S. Lam, R.P. Wilson, "Limits of Control Flow on Parallelism," *Proc. 19th International Symposium on Computer Architecture (ISCA)*, May 1992, pp. 19–21.

[32] A. LaMarca and R.E. Ladner, "The Influence of Caches on the Performance of Sorting," *Proc. 8th Annual ACM-SIAM Symposium on Discrete Algorithms*, pp. 370–379, 1997.

[33] C.G. Lee and D.J. DeVries, "Initial Results on the Performance and Cost of Vector Microprocessors," *Proc. 30th Annual International Symposium on Microarchitecture (MICRO-30)*, pp. 171–182, 1997.

[34] H. Levy, personal communication, August 1997.

[35] M. Mano, *Digital Logic and Computer Design*, Prentice Hall, Englewood Cliffs, NJ, 1979.

[36] J.D. McCalpin, "STREAM: Sustainable Memory Bandwidth in High Performance Computers," WWW document, University of Virginia. URL http://www.cs.virginia.edu/stream/

[37] K. Mehlhorn and U. Vishkin, "Randomized and Deterministic Simulations of PRAMs by Parallel Machines with Restricted Granularity of Parallel Memories," *Acta Informatica*, 21:339–374, 1984.

[38] J. Montanaro *et al.*, "A 160-MHz, 32-b, 0.5-W CMOS RISC PROCESSOR," *IEEE Journal of Solid-State Circuits*, volume 31, number 11, pp. 1703–1714, November 1996.

[39] D. Naishlos, J. Nuzman, C.-W. Tseng, and U. Vishkin, "Evaluating Multithreading in the Prototype XMT Environment," *Proc. 4th Workshop on*

Multi-Threaded Execution, Architecture and Compilation (MTEAC2000), December 2000. Best Paper Award.

[40] D. Naishlos, J. Nuzman, C.-W. Tseng, and U. Vishkin, "Evaluating the XMT Parallel Programming Model," *Proc. 6th Workshop on High-Level Parallel Programming Models and Supportive Environments (HIPS-6)*, pp. 95–108, April 2001.

[41] D. Naishlos, J. Nuzman, C.-W. Tseng, and U. Vishkin, "Towards a First Vertical Prototyping of an Extremely Fine-Grained Parallel Programming Approach," *Proc. 13th ACM Symposium on Parallel Algorithms and Architectures (SPAA)*, 2001. Extended version in *Theory of Computing Systems*, volume 36, pp. 521–552, August 2003. Invited paper for the SPAA 01 special issue.

[42] "The National Technology Roadmap for Semiconductors," Semiconductor Industry Association, 1997.

[43] J. Nuzman and U. Vishkin, "Circuit Architecture for Reduced-Synchrony On-Chip Interconnect," U.S. Provisional Patent Application 60/0297,248, June 2001.

[44] W. Oed and M. Walker, "An overview of Cray Research computers including the Y-MP/C90 and the new MPP T3D," *Proc. 5th ACM Symposium on Parallel Algorithms and Architectures (SPAA)*, pp. 271–272, 1993.

[45] V. Ramachandran, B. Grayson, M. Dahlin, "Emulations Between QSM, BSP and LogP: A Framework for General-Purpose Parallel Algorithm Design,"

Proc. 1999 ACM-SIAM Symposium on Discrete Algorithms (SODA '99), pp. 957–958, 1999.

[46] T. Sakurai, "Closed-Form Expressions for Interconnect Delay, Coupling, and Crosstalk in VLSI's," *IEEE Transactions on Electron Devices*, Vol. 40, No. 1, pp. 118–124, January 1993.

[47] Semiconductor Industry Association, *The International Technology Roadmap for Semiconductors*, 1999.

[48] U. Sigmund, M. Steinhaus, Th. Ungerer, "On Performance, Transistor Count and Chip Space Assessment of Multimedia-enhanced Simultaneous Multithreaded Processors," *Workshop on Multi-Threaded Execution, Architecture and Compilation (MTEAC-4)*, Monterrey, CA, December 10, 2000.

[49] A. Silberschatz and P.B. Galvin, *Operating System Concepts*, fifth edition, Addison Wesley Longman, Inc., 1998, p. 384.

[50] M. Steinhaus, R. Kolla, J.L. Larriba-Pey, Th. Ungerer, and M. Valero, "Transistor Count and Chip-Space Estimation of Simulated Microprocessors," submitted for publication, available as Research report UPC-DAC-2001-16, UPC, Barcelona, Spain, 2001.

[51] I. Sutherland, "Micropipelines," *Communications of the ACM*, 32(6):720–738, June 1989. Turing Award Lecture.

[52] D. Sylvester and K. Keutzer, "Rethinking Deep-Submicron Circuit Design," *IEEE Computer*, pp. 25–33, November 1999.

[53] D.M. Tullsen, S.J. Eggers, H.M. Levy, "Simultaneous Multithreading: Maximizing On-Chip Parallelism," *Proc. 22nd Annual International Symposium*

on Computer Architecture, Santa Margherita Ligure, Italy, pp. 392–403, June 1995.

[54] D.M. Tullsen, S.J. Eggers, J.S. Emer, H.M. Levy, J.L. Lo, and R.L. Stamm, "Exploiting Choice: Instruction Fetch and Issue on an Implementable Simultaneous Multithreading Processor," *Proc. 23rd Annual International Symposium on Computer Architecture (ISCA)*, pp. 191–202, 1996.

[55] U. Vishkin, "From Algorithm Parallelism to Instruction-Level Parallelism: An Encode-Decode Chain Using Prefix-sum," *Proc. 9th ACM Symposium on Parallel Algorithms and Architectures (SPAA)*, pp. 260–271, 1997.

[56] U. Vishkin, S. Dascal, E. Berkovich, and J. Nuzman, "Explicit Multi-threaded (XMT) Bridging Models for Instruction Parallelism," *Proc. 10th ACM Symposium on Parallel Algorithms and Architectures (SPAA)*, pp. 140–151, 1998. See also, the XMT home page URL http://www.umiacs.umd.edu/~vishkin/XMT/

[57] U. Vishkin, "A No-Busy-Wait Balanced Tree Parallel Algorithmic Paradigm," *Proc. 12th ACM Symposium on Parallel Algorithms and Architectures (SPAA)*, pp. 147–155, 2000.

[58] U. Vishkin, "Spawn-Join Instruction Set Architecture for Providing Explicit Multithreading (XMT)," U.S. Patent 6,463,527, October 8, 2002.

[59] D.W. Wall, "Limits of Instruction-Level Parallelism," DEC-WRL Research Report 93/6, November 1993.

[60] R. Wilson, R. French, C. Wilson, S. Amarasinghe, J. Anderson, S. Tjiang, S.-W. Liao, C.-W. Tseng, M. Hall, M. Lam, and J. Hennessy, "SUIF: An In-

frastructure for Research on Parallelizing and Optimizing Compilers," *ACM SIGPLAN Notices*, 29(12):31–37, December 1994.

[61] W. Wulf and S. McKee, "Hitting the memory wall: Implications of the obvious," *ACM SIGARCH Computer Architecture News*, 23(1):20–24, March 1995.