Looking to Parallel Algorithms for ILP and Decentralization

Efraim Berkovich[1,2,3], Bruce L. Jacob[1], Joseph Nuzman[1,2,3], and Uzi Vishkin[1,2,3]
Dept. of Electrical Engineering[1] and
University of Maryland Institute for Advanced Computer Studies (UMIACS)[2]
University of Maryland, College Park, MD

Abstract: *We introduce explicit multi-threading (XMT), a decentralized architecture that exploits fine-grained SPMD-style programming; a SPMD program can translate directly to MIPS assembly language using three additional instruction primitives. The motivation for XMT is: (i) to define an inherently decentralizable architecture, taking into account that the performance of future integrated circuits will be dominated by wire costs, (ii) to increase available instruction-level parallelism (ILP) by leveraging expertise in the world of parallel algorithms, and (iii) to reduce hardware complexity by alleviating the need to detect ILP at run-time: if parallel algorithms can give us an overabundance of work to do in the form of thread-level parallelism, one can extract instruction-level parallelism with greatly simplified dependence-checking.*

We show that implementations of such an architecture tend towards decentralization and that, when global communication is necessary, overall performance is relatively insensitive to large on-chip delays. We compare the performance of the design to more traditional parallel architectures and to a high-performance superscalar implementation, but the intent is merely to illustrate the performance behavior of the organization and to stimulate debate on the viability of introducing SPMD to the single-chip processor domain. We cannot offer at this stage hard comparisons with well-researched models of execution.

When programming for the SPMD model, the total number of operations that the processor has to perform is often slightly higher. To counter this, we have observed that the length of the critical path through the dynamic execution graph is smaller than in the serial domain, and the amount of ILP is correspondingly larger. Fine-grained SPMD programming connects with a broad knowledge base in parallel algorithms and scales down to provide good performance relative to high-performance superscalar designs even with small input sizes and small numbers of functional units.

Keywords: *Fine-grained SPMD, parallel algorithms. spawn-join, prefix-sum, instruction-level parallelism, decentralized architecture.*

---

[3] supported by NSF grant 9416890

1 Introduction

For some time, researchers have known that processor interconnect delays would some day constitute a dominating fraction of overall circuit delay [3, 29]. In future technologies, only a fraction of the processor will be reachable in a single clock cycle. For instance, at 0.1 μm (projected by SIA to occur within a decade [30]), only 16% of the die will be reachable in a clock cycle [26]. One conclusion is that future architectures must limit global communication; that is, most communication should be localized, and functions that require the propagation of cross-chip signals should be few and infrequently used. This has already surfaced in contemporary research and actual processors: for instance, Farkas, *et al*. have investigated implementing a partitioned register file to circumvent clock speed issues [8], and this type of partitioned register file has appeared in the recent Alpha 21264 [14].

In addition to the interconnect limit, architects are also faced with a perceived limit in available instruction-level parallelism; that is, our ability to extract ILP is growing less rapidly than our ability to integrate larger numbers of functional units on a single processor. This has led to several innovative paradigms in recent years to use functional units in novel ways [7, 9, 10, 23, 25, 27, 34, 36], as opposed to simply increasing the issue width of traditional superscalar designs.

To address these problems, we present an architecture that—like the multiscalar paradigm [10], the M-Machine architecture [9], "Raw" processors [36], single-chip multiprocessors [27], and simultaneous multi-threading [34]—maps well to future process technologies that are dominated by interconnect overheads and thus demand decentralization at an architecture level. Like the recent investigations of vector processing to more fully exploit on-chip bandwidth, to better map to future IC technologies, and to take full advantage of the large numbers of functional units available to today's microarchitects [7, 25, 23], this proposed architecture combines two things: (i) programming-model-specific support for an inherently parallel model of execution, and (ii) contemporary concepts in high-performance microarchitecture.

In our case, the inherently parallel model of execution is *single program, multiple data (SPMD)*, in which independent threads concurrently execute the same code on different data (e.g [22]). We use spawn-join "independence of order semantics" (IOS), where each virtual thread initiated by a spawn progresses at its own speed and terminates at a *join* instruction. Thus, no thread ever needs to wait on another thread. A process executing on the architecture will spawn threads that execute asynchronously in parallel, and when all threads have *join*ed, the process returns to serial mode until the next spawn. Our architecture, which is called *explicit multi-threading (XMT)*, supports this type of programming model. It implements spawn-join semantics in hardware, reminiscent of the *n*-way spawn-join semantics implemented in software by the KSR system [20] and the 2-way fork-join mechanism implemented in hardware by the P-RISC [28]. We provide no hardware support for separate stack space, as in P-RISC—all state local to a thread is contained in a separate hardware context: a local register file dedicated to each thread.

In this paper, we show that the XMT architecture inherently lends itself to decentralized implementations in which most on-chip communication is localized. We show that the architecture is fairly insensitive to the cost of on-chip communication even when the time it takes a signal to cross the chip is 32 clock cycles (which should correspond roughly to 0.05 μm technology [26, 30]). We describe a data cache organization and show how the programming paradigm inherently supports a relaxed consistency model, thereby allowing a simple cache coherence mechanism. We compare the performance of the design to high-performance superscalar designs; we also quote a comparison to more traditional parallel architectures. Our goal is to describe the architecture, motivate its design, and suggest the merit of further evaluating the paradigm as a means to designing high-performance execution engines for future technologies that are limited by interconnect delays.

1.1 This solution

Research over the last two decades by academic algorithm designers has produced a huge knowledge-base of parallel algorithmic methods, which is arguably second in its magnitude only to serial algorithms. The model of parallel computation used for developing this knowledge-base is called PRAM (for parallel random-access machine, or model). The (virtual) thread structure of PRAM-like algorithms is very dynamic: the number of threads that need to be generated changes frequently, new threads are generated and terminated frequently, and often threads are relatively short. Perhaps the most distinguishing feature about the XMT framework is that it envisions an extension to a standard instruction set which aspires to efficiently implement PRAM-like algorithms; XMT does so using explicit multi-threaded instruction-level parallelism.

The broad XMT framework [35] extends from algorithms to architecture. The purpose of this paper is to take a closer look at the architecture end by describing possible architecture choices and studying the microarchitecture performance implications. Our proposed architecture is inspired by IOS spawn-join SPMD programming with a flexible thread structure. The organization contains multiple hardware contexts on a single chip to which one can spawn independent threads to execute concurrently and allows for quick spawning, coordination, and termination of threads. Of course, during periods where the code is purely serial, these hardware contexts will lie dormant.

The XMT architecture integrates several well-understood and widely-used programming primitives that are usually implemented in software; the novelty of the architecture 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, similar to fetch-and-add [13].

A *spawn* instruction sets up the execution of a specified number of threads. The XMT architectural framework transparently manages the case when the number of threads to spawn is larger than the number of hardware contexts available. A *join* instruction signals that a particular hardware context has finished executing its thread; it either begins executing a new thread that is part of the current spawn (in the case where there is more work to do than processing elements to do it), or it lies dormant until the next spawn. The architecture described in the current paper has the compiler produce assembly code for explicitly starting and terminating the required threads, but direct hardware support for thread generation and termination may also be useful.

The prefix-sum operation is similar to an atomic fetch-and-increment [12] and can provide an emulation of serialization—that is, it enables conflict-free execution of multiple threads without the need for any thread to busy-wait. One can map any of a wide family of SPMD algorithms onto an architecture using these three primitives [35].

There are a number of benefits in following such an SPMD-style model of execution. First, it connects with a large body of knowledge in parallel algorithms, giving us algorithms and parallelizing compiler techniques with which to work. Second, by definition, concurrently executing threads are independent, which suggests a microarchitecture implementation in which separate areas of the chip can execute for periods of time without requiring any synchronization—this approaches the goal of decentralization. Third, the programming paradigm provides a degree of static memory disambiguation that one can exploit in the memory system: communication through the memory system will be highly structured, regular, and predictable; the XMT programming paradigm leads to trivially partitioned cache designs. And last, since the communication costs are low relative to those in more traditional parallel architectures, we can see the performance benefits of parallel programming at much lower data set sizes. The main potential drawback of the design is that it could exacerbate the already troublesome memory bandwidth problems since parallel programs typically require more data move-

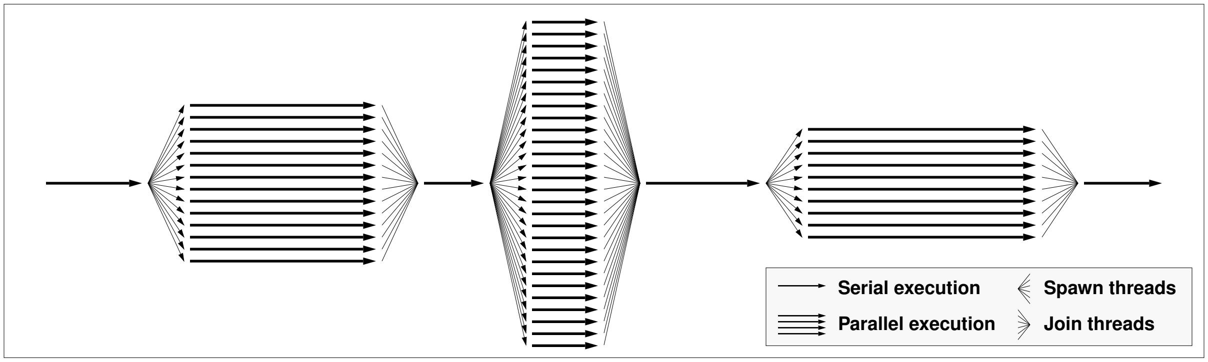

Figure 1: Serial and concurrent execution in a parallel application. A parallel application can spawn multiple threads during the course of its execution. The x-axis represents time; the heavier lines represent computation, and the lighter lines represent the creation and termination of concurrent threads.

ments than comparable serial programs. Another concern is the complexity of design verification. While the architecture appears attractive for its modularity, the interaction issues among the various elements as well as the magnitude of the design size may hinder verification.

1.2 Overview of this paper

In this paper, we address these issues and provide insight into the performance behavior of the architecture. Section 2 describes the XMT programming paradigm and the hardware architecture that supports it. Section 3 describes one possible implementation which exhibits the property of decentralization; it also describes the architecture used in our simulations. Section 4 provides insight into the performance behavior of the system, as compared to massively parallel processors and traditional superscalar designs. Section 5 addresses some of the potential drawbacks of implementing our model on a single chip. Section 6 describes related existing work and Section 7 concludes.

2 The XMT parallel programming paradigm

Spawn-join and *prefix-sum* translate to three instruction primitives that are orthogonal to the underlying instruction set architecture—they can be added retroactively to a standard instruction set.

2.1 Spawn-join semantics

An in-depth discussion or primer on parallel algorithms is beyond the scope of this paper; the interested reader is referred to [18]. Suffice it to say that many programs can be made to look like the diagram in Figure 1, where the heavier lines represent computation and the lighter lines represent the spawning and joining of multiple parallel threads. The diagram shows a process that splits several times during the course of its execution into multiple concurrent threads. Each time multiple threads are spawned, the number and execution length of the threads spawned may vary; each thread may

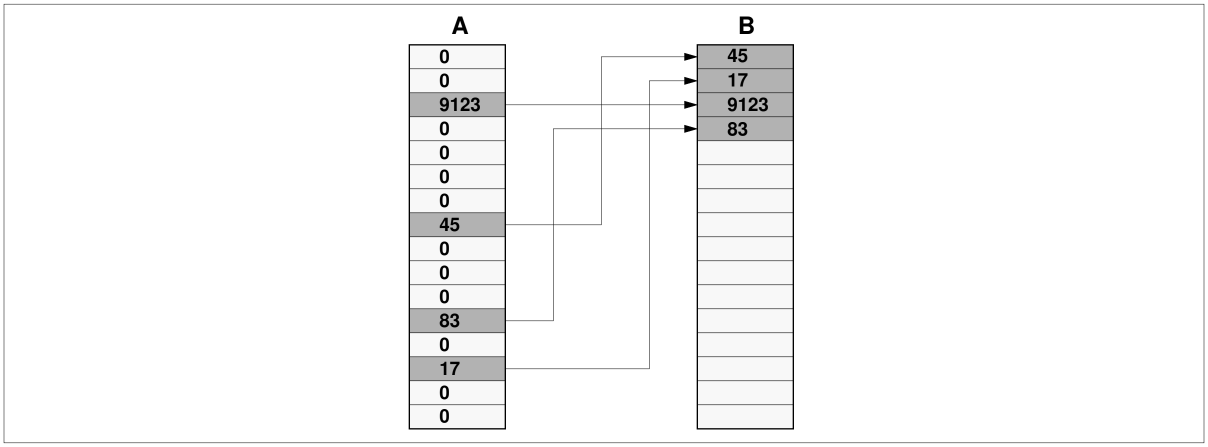

Figure 2: The array compaction problem. The non-zero values in array A are copied to array B, in an arbitrary order. This is easily parallelized using the prefix-sum operation.

proceed at its own speed from beginning to end.

Much code is known to be parallelizable without resorting to multi-threading. For example, in loop unrolling and software pipelining, we allow different loop iterations to execute concurrently. In a multithreaded design, one can assign different loop iterations to different independent threads, subject to the same loop-independence caveats.

2.2 Prefix-sum semantics

The prefix-sum is similar to a fetch-and-increment. It has the following semantics, where B is called the "base" of the prefix-sum, and R is a register that acts as both a source and a target:

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

The instruction by itself is not very interesting, but it happens to be very useful when several threads perform a prefix-sum against a common base. Since each prefix-sum is atomic, each thread will receive a different value in its local storage R; the mechanism can thus be used for parallel execution of threads that emulates serial execution without resorting to busy-waiting. Prefix-sum is frequently used in parallel algorithms for this and many other purposes. If a large number of prefix-sums using a common base can be performed in a short amount of time (e.g. a constant number of cycles: O(1), not O(n)), then the primitive can support efficient inter-thread communication; this is possible if the value to be added to the base is constrained to be small (for instance, a 1-bit value) [35].

To demonstrate the use of the prefix-sum, suppose we have an array of integers and wish to "compact" the array; that is, we wish to copy all non-zero values from the array into another array, thereby creating a smaller, denser array. Figure 2 illustrates. Here is the corresponding pseudocode using spawn, join, and prefix-sum primitives; all threads execute the same code (the *spawn-block*, delimited by curly brackets—the right bracket is an implicit *join*), and the variables within the brackets are assumed to be local to each thread:

```
int arrays A[n], B[n];
int base = 0;
spawn (n)
{
        int $ID; /* different for each thread: values between 0 and n-1 */
        int r = 1;
        if (A[$ID] != 0) {
                r = prefix_sum(r, base);
                B[r] = A[$ID];
        }
} /* join */
```

The variable $ID is the thread ID (analogous to a loop counter variable). Each thread in a spawn receives a different ID. We will discuss one implementation of assigning thread IDs in Section 3.

Given the array in Figure 2, 16 threads will execute the code in the spawn-block. Each will receive a unique thread ID and so each will load a different value from array A. Only four will load non-zero values. These four will attempt to write to array B, but will synchronize their access through a prefix-sum. Each executes the prefix-sum independently and (potentially) at a different point in time. After the execution of the prefix-sum, each of the remaining threads' local value *r* contains a unique value between 0 and 3 (because the prefix-sum is executed by four threads, thereby incrementing the base by 4). Each thread uses its $ID and *r* values to read and write to the arrays without fear of causing memory inconsistencies. No thread ever needs to busy-wait. Note that once all threads are finished, the variable *base* contains the size of the compacted array B.

2.3 XMT instruction set

In the SPMD-based XMT model, all threads of a common spawn-block run the same code; *i.e.* they begin execution at the same program counter. Therefore the spawn operation needs to specify a PC, the size of the spawn, and the initial thread ID. Without loss of generality, the IDs assigned to the threads of a spawn are all the integers between the initial ID and (the initial ID + the size of the spawn - 1). The join operation needs no arguments. The prefix-sum operation needs to specify a base (which can be either a global register or a global memory location) and a register local to the thread issuing the prefix-sum.

These primitives are orthogonal to any underlying instruction-set architecture, and for our simulations, we simply added the following instructions to a MIPS-like instruction set[1] (actually, the instruction set looks like SimpleScalar [4], as it does not use architected delay slots).

spawn rS, id, offset: Instantiates a number of threads all starting at a PC-relative offset. The size of the spawn is found in register **rS**, the initial thread ID is found in the immediate value **id**, and the PC-relative offset at which each thread begins execution is found in the immediate value **offset**. Because the base thread ID is usually an integral part of the algorithm, it is usually known at compile time; we therefore use an immediate value.

join: Signals the end of a spawn-block.

ps.r rR, rB: Executes a prefix-sum operation using a global register as a base. The base register is register **rB**, and the local register (which is used as both a source and a target for the prefix-sum) is register **rR**. After execution, register **rB** contains the sum of **rB** and **rR**, and the register **rR** contains the value previously found in register **rB**.

psi.r rR, rB, imm: Immediate-value form of the prefix-sum instruction **ps.r**. This behaves like **ps.r**, except that the value added to the base is the unsigned immediate value **imm**, therefore register **rR** is used only as a target, not as a source.

We also assume the availability of load and store instructions using scaled addressing mode [16] to simplify array addressing: **lwa** and **swa**. For the purposes of the examples in this paper, these take the form **op rT, C(rB)[rI]**, where **rT** is the target register, **C** is a constant offset, r**B** is the array base, and **rI** is the array index that is scaled (left-shifted) by the wordsize.

To distinguish between registers local to a thread and registers global to all threads, we modify the assembly-code specification of registers while in a spawn-block. The label **tN** is meant to indicate register N local to the thread, and the label **gN** refers to global register N. To avoid breaking compatibility with existing binaries, assemblers, compilers, development tools, and operating systems for the instruction set, one can simply divide the architectural register file into two partitions: the lower partition contains the "global" registers, the upper partition refers to registers local to the thread. Therefore, a reference to register $5 in the MIPS assembly language implies a global register access, while a reference to register $37 implies access to a register local to an individual thread.

We can now rewrite the earlier example pseudocode in MIPS-style assembly language. To simplify the example, we assume that the arrays are small and that their base addresses can be refer-

---

1. The paper [35], this paper, and the simulator use slight variations of the assembly language described.

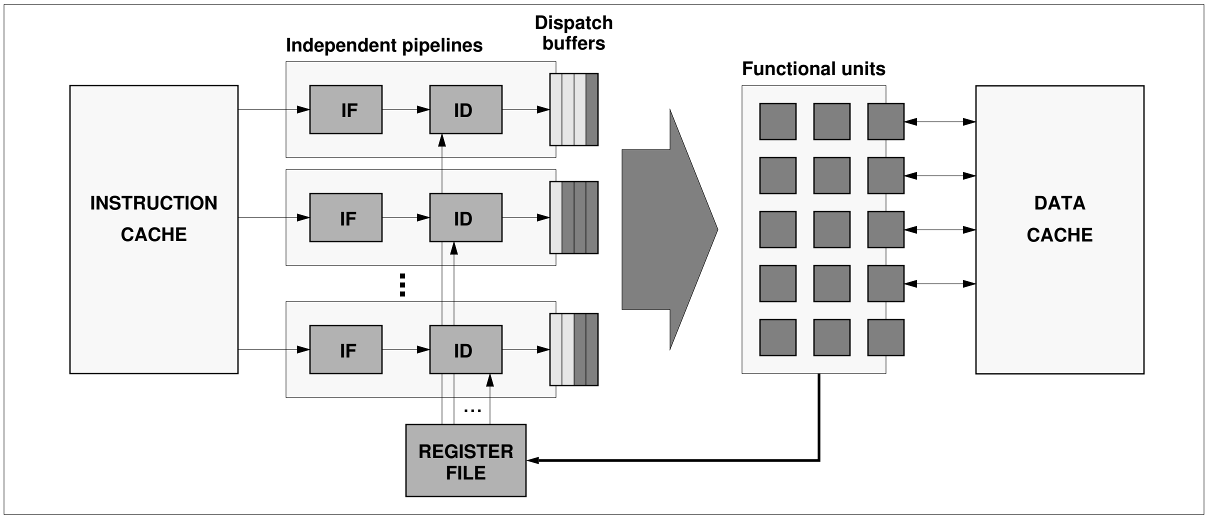

Figure 3: An XMT cluster. The XMT cluster includes a small instruction cache, a small data cache, a local register file, several independent pipelines, and a number of functional units shared among the pipelines. Its organization is similar to many contemporary superscalar designs, as well as an SMT processor.

enced by an immediate value (so we can use constants in the load and store instructions). Assume that the array size (and spawn size) *N* is a constant less than 65,536:

```
# Register assignments:
# g0 - is hard-wired to zero
# g1 - holds the spawn size (N)
# g3 - the prefix-sum base
# t0 - holds the unique thread ID
# t1 - holds the value A[ID]
# t2 - holds a unique index into array B

A:      .space 4 * N          # allocate array A
B:      .space 4 * N          # allocate array B
        ori g1, g0, N         # initialize register holding the spawn size
        ori g3, g0, 0         # initialize prefix-sum base
        spawn g1, 0, GO       # spawn N threads starting at PC go
GO:     # at this point, t0 is initialized with the unique thread ID
        lwa t1, A(g0)[t0]     # load the IDth element of array A (wordsize = 4)
        beq t1, g0, END       # if A[ID] is zero, skip to join instruction
        psi.r t2, g3, 1       # get unique index into array B
        swa t1, B(g0)[t2]     # store value loaded from A into B (wordsize = 4)
END:    join
```

3 A decentralized implementation

This section describes an XMT implementation; it is intended to offer insight into the XMT architectural paradigm, not to stand as the sole implementation of that paradigm. To begin with, we note that most communication will be intra-thread. This suggests a multi-clustered organization, wherein each cluster looks much like a contemporary superscalar architecture or SMT design. An example of a cluster is shown in Figure 3. The important differences between an XMT cluster and a contemporary superscalar design are that, due to the programming paradigm, the pipelines of the XMT cluster are independent (each manages a single thread) and need no cross-checking of dependencies to send instructions to the dispatch buffers. Similarly, the issue mechanism is simplified because the hardware need not cross-check dependencies between instructions in different dispatch buffers when issuing to functional units; and we can divide the register file into as many partitions as there are independent pipelines (i.e. it might be prudent to protect the registers of different threads), therefore the register file can be multiported through banking without fear of contention for read or write ports. Alternatively, one could share the register file among threads to implement low-latency inter-thread communication, but that is beyond the scope of this paper; for the moment we are interested in specifying as decentralized a design as possible. We will refer to a hardware thread context as a *thread control unit (*TCU*)*; each of the independent pipelines in the diagram is a TCU. Each TCU has a fixed unique ID used only to determine static ranking (i.e. which TCU should take priority in a given situation).

Each cluster should be able to handle as many threads (as many TCUs) as is reasonable, given the limitation of single-cycle communication within a cluster; that is, a cluster should be as large as possible without being so large that signals take multiple clock cycles to propagate within the cluster. Mechanisms that require extra-cluster communication can be centrally located to minimize the worst-case communication cost; other organizations such as those that optimize best-case communication are possible, but we do not consider them in this study.

Figure 4 illustrates the organization of independent clusters within the full architecture, and shows the extra-cluster communication channels. These are the channels that will take multiple cycles to send a signal across. The operations that require extra-cluster or inter-cluster communication are as follows:

- prefix-sum
- spawn/join
- references to global registers
- cache misses and cache coherence

The following sections describe each of these in more detail.

3.1 Prefix-sum coordinator

The prefix-sum mechanism goes through a central facility that can compute results from n threads in O(1) time, not O(n) time: there is a dedicated prefix-sum bus with 1 bit of data per hard-

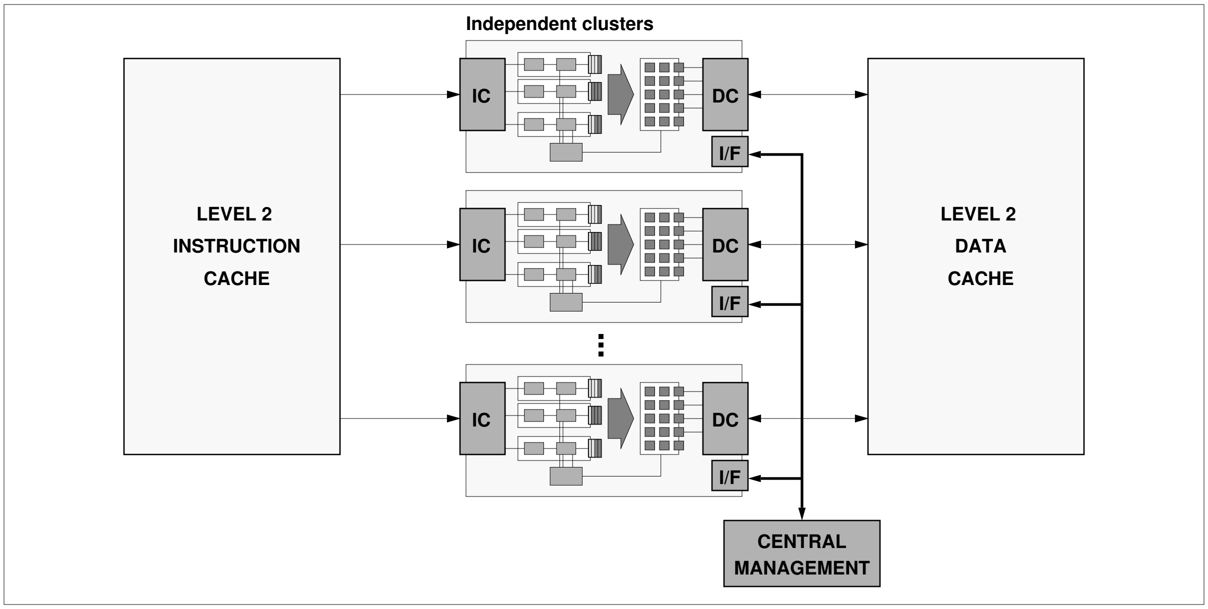

Figure 4: The full XMT architecture. Communication paths in this diagram require more than one cycle. The module labeled "central management" contains such functions as the prefix-sum operator, global registers, and the spawn-join coordinator.

ware context. All prefix-sum requests that arrive 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 are fired out on a dataword-wide bus as soon as the first request for a particular base arrives, followed by the register number identifying the base. The I/F (interface) units in each cluster keep track of these bases. A prefix-sum result is a single-cycle broadcast of two bits per hardware thread: one bit indicates whether the thread participated in the prefix-sum, the other bit is the value that the thread contributed to the sum. We can use a static ranking scheme (e.g. smallest TCU ID goes first) to determine the order in which threads add to the base, therefore each I/F unit can compute its own results in a distributed manner. The coordinator broadcasts a base value only once; it need not send out the contents of the base register again until a thread explicitly writes to that register (thus modifying it in a manner opaque to the I/F units). While this organization increases the size of each cluster, it reduces the amount of extra-cluster communication, as well as the widths of the buses required.

3.2 Spawn/join coordinator

The central spawn/join coordinator needs to do two things: (a) to activate all the TCUs and broadcast the PC of the spawn to them and (b) to discover when all TCUs have stopped executing threads so that it can resume serial mode. The coordinator can broadcast the spawn-PC over the same data bus that the prefix-sum unit uses (requiring a few bits of control signal to identify the data on the bus). When they finish, the TCUs can signal the central coordinator using a single bit-line per cluster that ANDs the TCU signals. The implication is that once the spawn coordinator initiates a spawn, it must wait a minimum number of cycles before it can assume the *join* signals are valid. Thus a spawn-block of only one instruction will take as much time to execute as a spawn-block of two or three instructions. Provided that short threads are infrequent, we do not think this will be problematic; however, we are in the process of quantifying it.

Given prefix-sum, spawn/join becomes simple. The spawn coordinator broadcasts a spawn-PC and all TCUs begin executing at that program counter. The first task of each thread is *to obtain a thread ID and then verify that it is less than the spawn size*. (Note that, for simplicity, we left out that step earlier.) Why is this important? Doing so supports an extremely simple mechanism for assigning new threads to TCUs when they finish: when the central management sends out a single spawn, it need not do anything else until all TCUs signal that they have joined. A TCU keeps trying to instantiate new threads until it fails; then it joins. When the thread gets to the end of the spawn-block, it jumps back to the beginning to attempt to start up a new thread. As soon as it obtains an invalid thread ID, the TCU executes a join. Here is the final code for the array compaction algorithm:

```
# Register assignments:
# g0 - is hard-wired to zero
# g1 - holds the spawn size (N)
# g2 - holds the thread ID counter
# g3 - the prefix-sum base
# t0 - holds the unique thread ID
# t1 - holds the value A[ID]
# t2 - holds a unique index into array B

A:      .space 4 * N          # allocate array A
B:      .space 4 * N          # allocate array B
        ori g1, g0, N         # initialize register holding the spawn size
        ori g2, g0, 0         # initialize register holding the initial thread ID
        ori g3, g0, 0         # initialize prefix-sum base
        spawn g1, 0, GO       # spawn N threads starting at PC go
GO:     psi.r t0, g2, 1       # get unique thread ID into t0
        slt t1, t0, g1        # is the ID less than the spawn size?
        beq t1, g0, END       # if not, terminate execution
        lwa t1, A(g0)[t0]     # load the IDth element of array A
        beq t1, g0, GO        # if A[ID] is zero, try to start another thread
        psi.r t2, g3, 1       # get unique index into array B
        swa t1, B(g0)[t2]     # store value loaded from A into B
        j GO                  # try to instantiate a new thread
END:    join                  # really terminate
```

Note that we have initialized register **t0** (holding the unique thread ID) with a prefix-sum operation, then verified the value before using it. Since thread IDs are defined to be all integers between some base and base+spawnsize, prefix-sum is a natural mechanism to use, and doing so can reduce the amount of special hardware needed by the system. Note also that this implementation obviates the need to specify any parameters in the *spawn* instruction. Furthermore, it is possible to optimize this code a bit more since the initial thread IDs can be computed directly without the use of a prefix-sum:

```
        # TCU_ID is the static TCU identifier, 0 to num TCUs - 1

        spawn g1, 0, GO        # spawn N threads starting at PC go
GO:     addi t0, TCU_ID, 0     # get initial thread ID into t0
L1:     slt t1, t0, g1         # is the ID less than the spawn size?
        beq t1, g0, END        # if not, terminate execution
        ...
        swa t1, B(g0)[t2]      # store value loaded from A into B
        psi.r t0, g2, 1        # get unique thread ID into t0
        j L1                   # try to instantiate a new thread
END:    join                   # really terminate the initial batch of thread IDs.
```

3.3 Global register coordinator

It is likely that the global register coordinator can use the same broadcast data bus as the prefix-sum unit, since that bus is used once per initialization of a register to be used for a prefix-sum and once per spawn. The rest of the time the bus lies dormant. Therefore we believe we can also use it to broadcast global register values whenever they are written. Each local register file can cache these values, as, by virtue of the programming model, communication through these registers will be regular. During a spawn block, the programming model guarantees that a global register will be either read-only by many threads, exclusively read-write by one thread, or written by many threads by means of a prefix-sum. Therefore the clusters can safely cache the global register values and simply invalidate them at the end of a spawn block.

3.4 Cache consistency

The per-cluster instruction caches need not be kept coherent since they are read-only. By virtue of the programming model, during parallel execution (during a spawn), threads are guaranteed never to overwrite each other's data—they are completely independent. Therefore, the data caches need not be kept coherent during a spawn, and as in Hammond's scheme [15], we can have a very simple write-buffer mechanism to keep the on-chip level-2 cache up-to-date. It is likely that a write cache will be very useful in such an environment [19]. While operating in parallel mode, each cluster's data cache can withstand a degree of inconsistency with the data caches of other clusters. Once serial mode returns, the caches should become consistent, which means updating or invalidating every line that was partially written. This is similar to the global-registers mechanism that must invalidate cached copies of the global registers. We have not studied this but intend to.

3.5 Design optimizations

Many optimizations of the basic XMT design are possible. The implications and tradeoffs in these optimizations are beyond the scope of this paper. An extension which we have started to research is a prefix-sum to memory operation. This operation would be a prefix-sum which uses a memory address as a base instead of a register. The issues here which appear to preclude a low communication design are large (memory-address wide) base identifiers and the large number of bases.

For optimizing thread creation, one can add hardware support instead of the spawning and joining code described above. We implement this optimization in our simulator. The additional hardware required is a dedicated prefix-sum for the purpose of generating new thread IDs and a special spawn register will be used for the base instead of a general purpose register. The spawn prefix-sum unit can have simpler hardware than the general purpose prefix-sum because no mechanism for detecting different bases is necessary and the communication can be simplified.

While for simplicity of hardware and design single issue pipelines for TCUs were described, it should be possible to increase the issue width of each TCU by means of standard superscalar techniques. Adding optimizations like branch prediction may also be desirable. We defer this question to future research since it is not clear what level of extra hardware complexity per TCU is desirable.

Note that when the XMT processor is in serial mode, it is simply running a single thread on TCU 0. In our current work, TCU 0 is identical to any other TCU in the system. However, because this first unit has the unique task of running serial code, it may be augmented with additional superscalar improvements and a direct connection to global registers. This improvement should enable the processor to execute serial code with performance comparable to a standard superscalar.

3.6 Organization of simulator

The XMT simulator that produced the performance results is similar to the model described in this section, without the per-cluster caches. The rest of the differences are outlined in Table 1.

Table 1: XMT simulator organization

Spawn-Join	Hardware support instead of software implementation: dedicated PS unit for thread ID generation. Join handling early in pipeline.
Memory	D-cache and I-cache can have up to 200 pending read requests each, a cache line is fetched when a request is filled. Two configurations were used: 100 cycle latency and 1 read processed per cycle ("real" memory); 1 cycle latency and 8 reads processed per cycle ("perfect" memory)
Caches	Cache lines are 4 64-bit words long. D-cache is 2 way set associative. I-cache is direct mapped and holds 128 cache lines. D-cache is write through with no fetch on write-miss.

Table 1: XMT simulator organization

Clusters	6 ALUs (1 cycle latency); 4 Branch (1 cycle latency); 6 MultDiv (2 cycle multiply, 40 cycle divide, neither is pipelined). 4 slot load-store buffer, 4 ports to D-cache. PS I/F unit has 8 slot buffer for pending requests.
TCUs	Six stage single issue pipeline, stall on branch, 4 slot Dispatch buffer, 4 TCUs/Cluster
Prefix-sum (PS)	Units are pipelined with 3 cycle latency after request received.
Configurations	TCUs: 8, 32, 128. Each has a D-cache scaled with its size. D-Cache: 2K word; 8K word; 32K word. Two additional configurations with Join Delays of 4 and 16 cycles on the 128 TCU design. These 5 configurations were each set with "real" and "perfect" memory for a total of 10 configurations.

4 Performance behavior

In this section, we provide intuition into the performance behavior of the system. Since this is a new model of execution, we cannot hope to provide a thorough performance comparison with more established models such as superscalar, vector, etc. That is a future goal. Here, we make the first steps towards such a characterization.

4.1 Performance compared to MPPs

In MPPs, latency is typically measured in microseconds; on a single chip, latency will be measured in nanoseconds. This creates an opportunity; whereas reduced communication costs will not eliminate the inherent start-up costs of parallel algorithms, reduced communication costs do allow the XMT architecture to become competitive with serial implementations at much smaller data-set sizes. Also, manufacturing costs for XMT implementations should be orders of magnitude lower than those of MPPs. Thus, achieving a speedup over serial implementations that is commensurate with the cost ratio of the implementations should be possible at much lower data-set sizes. We have described this elsewhere [35]. Figure 5 gives an example of the comparison between XMT and the Intel Paragon on list ranking, which is a fundamental component of most parallel graph algorithms. Note that the data-set sizes where the parallel implementations become competitive, and

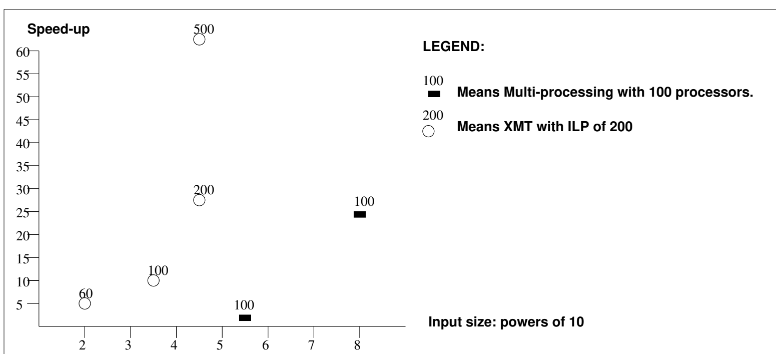

Figure 5: XMT vs. MPP on list ranking (very similar to the listsort benchmark discussed later).

The graph shows that XMT can achieve speedups over serial implementations at much lower data-set sizes than a traditional MPP. "ILP" simply means the total number of instructions the XMT design can have pending during a cycle.

especially the data-set sizes where the parallel implementations become cost-competitive, are much lower than in the massively parallel implementations.

4.2 Performance compared to superscalars

The problem with comparing a parallel model to a serial one is that it is very difficult to find benchmarks that are not biased against one of the execution models. Therefore we chose to compare not entire benchmarks but components of applications that are primitives used in both serial and parallel benchmarks. We are aware of the criticism of observing program kernels rather than observing entire programs, but we feel that it is a much more impartial comparison, and it still provides reasonable intuition into the performance of the XMT architecture. We picked code snippets with a variety of degrees of inherent parallelism. We tried to find representatives of a particular behavior rather than representatives of classes of applications.

We chose small data sets to illustrate the fact that the XMT model can achieve speedups over serial code for small input sizes. Table 2 gives descriptions of the benchmarks. The serial execution

Table 2: Code snippets used in experiments

Benchmarks	Description	Input sizes
Serial **linkedlist**	Traverse a linked list 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 item list spaced in 2000 words, 5000 item list spaced in 20K words
condition	An if-statement with six clauses ANDed together. Here we demonstrate how MT techniques can be used to speed up code which seems inherently serial by splitting each clause into a thread. Whereas this type of approach may be useful to implement at the compiler level on particular kinds of code (such as our example), it is not the thrust of the XMT programming model.	Six conditions
Embarrassingly parallel **stream**	Based on the STREAM benchmark [33], 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, 5000 item array
Mildly 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, 5000 item array (uncompacted arrays are 1/4 full)

Table 2: Code snippets used in experiments

Benchmarks	Description	Input sizes
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, we have two approaches. Both use balanced binary trees 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. In **max_synch**, we synchronize after every level in the tree. The threads are very short and there are log(n) spawn-joins. The **max_asynch** snippet spawns a thread for each value in the list. Each thread carries its value as far up the tree as it can: terminating if it is the first thread to arrive to a node and performing the comparison between two values and carrying the maximum higher if it is the second to arrive. The last thread to arrive to the root of the tree will have the maximum. We do not present results for **max_asynch** since it uses prefix-sum to memory operations and we do not discuss those in this paper.	50 item array, 500 item array, 5000 item array
Sorting **listsort**	Unraveling a linked list of known length which is packed within an array. This application is useful for managing linked-list free space in OSes [31]. In the serial algorithm, we traverse the list and rewrite it in the proper order. For the XMT version, we use the simplest (from a coding point of view) approach: Wyllie's pointer jumping algorithm [18]. We note that more efficient parallel approaches exist and were explored in [35].	50 item array, 500 item array, 5000 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 [24].	50 item array, 500 item array, 5000 item array

model is represented by SimpleScalar [4]. The XMT organizations simulated are described in Section 3. SimpleScalar was configured with perfect branch prediction; 8-way/32-way/128-way instruction fetch, decode, issue, and retire; and 2K/8K/32K word level-1 caches. Figures 6 and 7 give the results of the comparison with XMT.

The extra superscalar parallelism gave limited performance improvement; for example, SimpleScalar 128-way executed the 500 input size **stream** benchmark at only a 1.08 speedup over the 8-way configuration. For the larger input sizes, the increased cache size for the configuration had a much more significant effect on performance than the increased parallelism. On the other hand, XMT was able to take advantage of the increase in available parallelism as demonstrated in the graphs.

For XMT, we get significant speedups for 32-way parallelism for most of the snippets. Note that while these speedups typically increase for going to 128-way parallelism, we usually do not get to the theoretical factor of 4 improvement. An explanation is that the 100 cycle latency to memory does not allow more than 100 requests to be in-flight at the same time. A simple example follows. Consider 512 virtual threads in a program and suppose that each thread results in a single cache miss. For the 32 TCU configuration, this would mean processing the first 32 misses in 132 cycles, the next 32 in 100 more cycles, and so on for a total of 1632 cycles. For 128 TCUs, on the other hand, it is not better than 100 TCUs; it will take a total of 612 cycles to process the same 512 misses. Since 1632/612 is about 2.7, we see why approaching a factor of 4 improvement may be impossible. Note that for 8 TCUs it takes 6408 cycles to process the misses and that 6408/1632 is about 3.9. This effect may explain why the 32-way configuration often offered better performance over the 8-way than the 128-way offered over the 32-way.

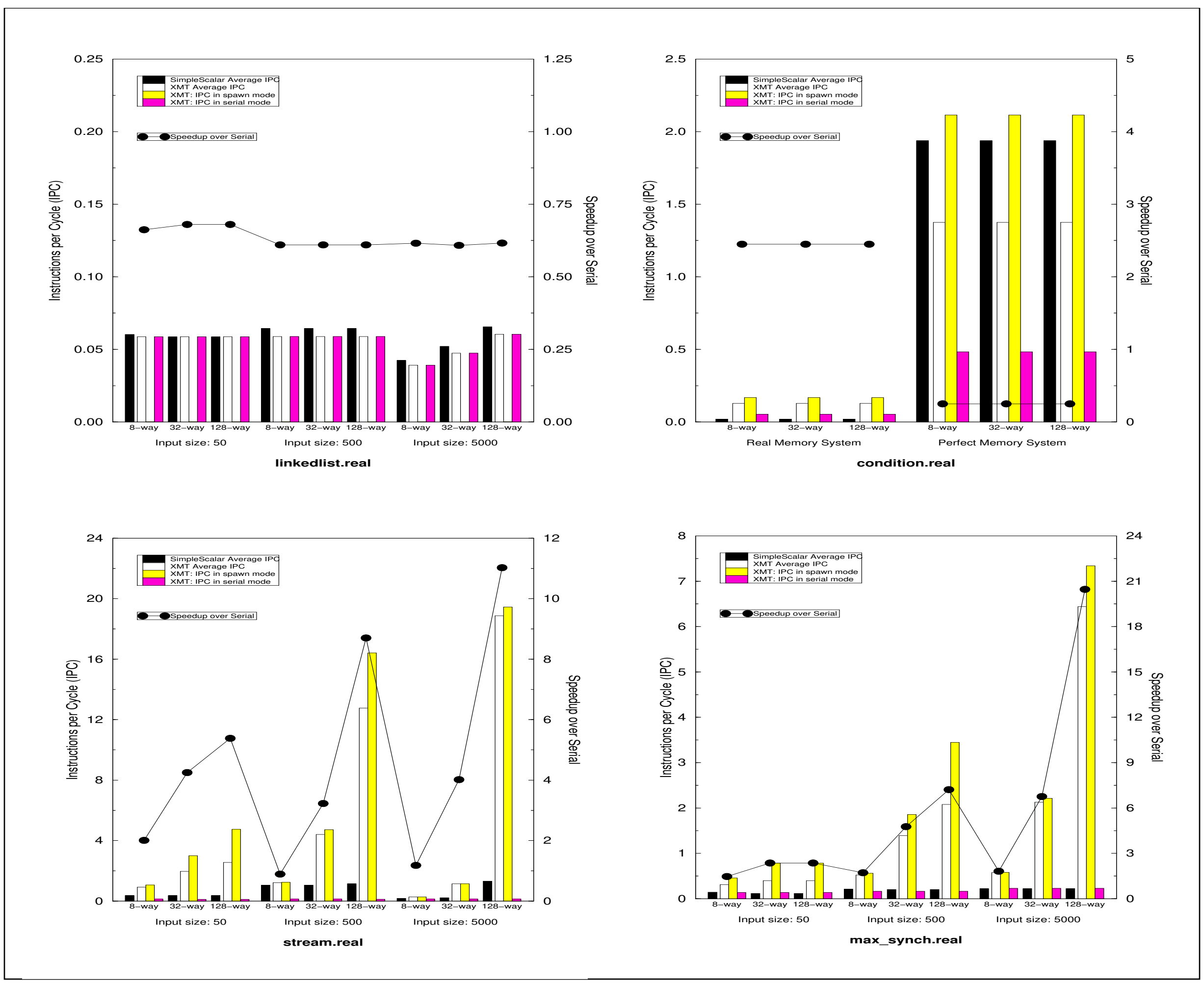

Figure 6: XMT vs. serial execution for linkedlist, condition, stream, and max.

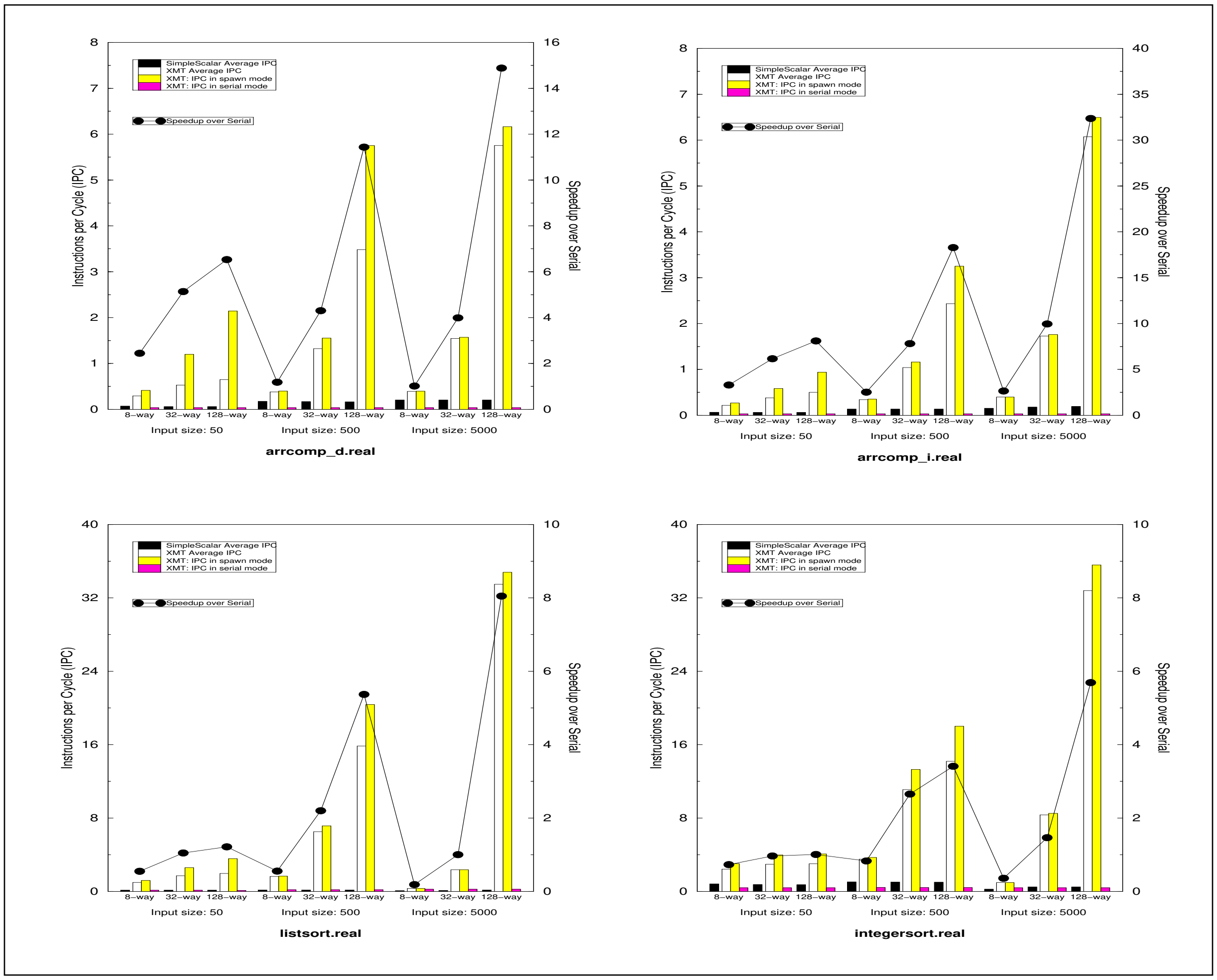

Figure 7: XMT vs. serial execution for arrcomp_d, arrcomp_i, listsort, and integersort.

Wyllie's algorithm (used in **listsort**) is competitive when we have small inputs and high parallelism because of its O(n log$n$) work behavior [35]; note the degradation in performance as input size increases. For input size of 5000, another parallel algorithm would be a better choice. At this input size, cache effects also start to play a role as the number of misses decreases from 128068 for the 8-way to 57834 for the 32-way to 8407 for the 128-way XMT (note the jump in performance from 32-way to 128-way); this is a peculiarity of the cache/input size interaction for this problem.

We modeled long-distance communication in the XMT simulator; we modeled thread ID generation requests (i.e., prefix-sum after a *join*) as taking 1, 4, and 16 cycles (corresponding to cross-chip communications of 2, 8, and 32 cycles). The normalized execution time results are given in Figure 8 for two of the benchmarks configured with "real" and "perfect" memory. The rest of the benchmarks behave similarly and show that the impact of increased communication cost on the performance behavior of a program is secondary to the latency issue and suggest that XMT is resilient to increased interconnect delays.

5 Potential drawbacks

5.1 Complexity of the design

There is an inherent complexity in a single-chip parallel-computer implementation; as a result, the interaction of numerous independent state machines will likely make verification of the design extremely difficult. While it is not intuitively clear whether verification will be more or less difficult than that of an enormously complex superscalar machine with branch prediction, speculative execution, out-of-order issue, and several forms of data prediction, it nonetheless is intuitively clear that the fundamental building blocks described in this paper (the XMT clusters) will individually be simpler to verify than a complex superscalar machine, as the implementation we studied here dispenses with all of that complexity and still achieves a speedup over a serial implementation. Similarly, the overall processor should be easier to design, since design involves little more than the "stapling together" of these clusters. The complexity could negatively affect clock speed compared to a superscalar design, but because we have restricted ourselves to using almost entirely components that are commonplace in today's high-speed processors, we do not consider this an issue.

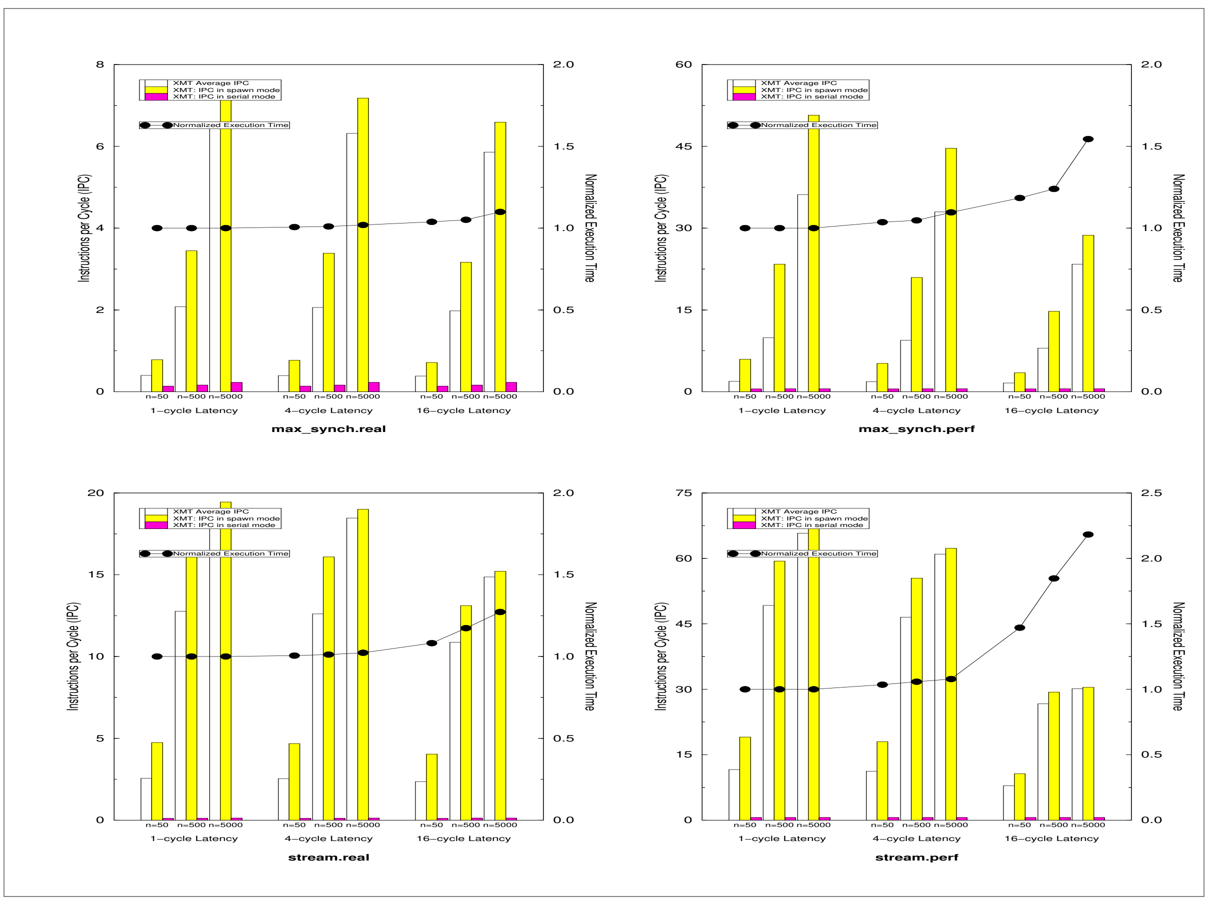

Figure 8: A look at the effects of interconnect delays.

There is also the issue of scalability: the prefix-sum mechanism, to reduce the amount and overhead of long-distance communication, uses a bus that scales with the number of TCUs in the system, which is inherently problematic if one wishes to move to large numbers of hardware contexts. There are other implementation possibilities, however: a more scalable implementation could work as follows. First, we notice that a prefix-sum request which contributes zero is equivalent to a read of the base, with no specified ordering. Thus, these requests can be handled locally by the cluster I/F unit. Prefix-sum instructions 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 requests into batches of the same base, performs a prefix-sum across the batch, and broadcasts the results. Each cluster would effectively be given a range within that cycle's prefix-sum and it can decide which of its local prefix-sum requests gets which value. It would reduce the number of wires from $2n$, where n is the number of TCUs, to $c*\log_2(n/c)$, where c is the number of clusters. This approach can generalize to a hierarchical tree structure in which each node in the tree is given a range within the prefix-sum by its parent and gives sub-ranges of that range to its child nodes.

5.2 Memory traffic

A serious issue is the increased memory traffic because of the pattern in which parallel algorithms tend to revisit more of their data relative to their serial counterparts. In XMT, we reuse TCUs for different virtual threads and so flush data which could actually be reused. This large-scale data-movement is necessary for XMT but would often be redundant in a serial uniprocessor executing a different algorithm. However, since a parallel algorithm provides better memory latency hiding, improved throughput to memory can make the performance of the parallel algorithm competitive. This effect of multi-threading is demonstrated by the data in Table 3.

Table 3: Memory throughput comparison

Benchmark	SimpleScalar (128-way)	XMT (128 TCUs)	Factor increase XMT vs. SS
arrcomp_d (5000 item array)	dcache misses: 1251 time in cycles: 70945 mem fetches/cycle: 0.0176	dcache misses: 1251 time in cycles: 4767 mem fetches/cycle: 0.262	misses: 1.00 speedup: 14.9 mem throughput: 14.9
arrcomp_i (5000 item array)	dcache misses: 2095 time in cycles: 159230 mem fetches/cycle: 0.0132	dcache misses: 2095 time in cycles: 4919 mem fetches/cycle: 0.426	misses:1.00 speedup: 32.4 mem throughput: 32.4
integersort (5000 item array)	dcache misses: 2274 time in cycles: 168541 mem fetches/cycle: 0.0135	dcache misses: 4566 time in cycles: 29617 mem fetches/cycle: 0.154	misses: 2.01 speedup: 5.69 mem throughput: 11.4

Table 3: Memory throughput comparison

Benchmark	SimpleScalar (128-way)	XMT (128 TCUs)	Factor increase XMT vs. SS
linkedlist (5000 item list)	dcache misses: 4365 time in cycles: 305569 mem fetches/cycle: 0.0143	dcache misses: 4365 time in cycles: 495966 mem fetches/cycle: 0.0088	misses: 1.00 speedup: 0.616 mem throughput: 0.616
listsort (5000 item list)	dcache misses: 2501 time in cycles: 195214 mem fetches/cycle: 0.0128	dcache misses: 8407 time in cycles: 24252 mem fetches/cycle: 0.347	misses: 3.36 speedup: 8.05 mem throughput: 27.1
max (5000 item array)	dcache misses: 1251 time in cycles: 135146 mem fetches/cycle: 0.00926	dcache misses: 2501 time in cycles: 6605 mem fetches/cycle: 0.379	misses: 2.00 speedup: 20.5 mem throughput: 40.9
stream (5000 item array)	dcache misses: 3751 time in cycles: 192050 mem fetches/cycle: 0.0195	dcache misses: 3751 time in cycles: 17421 mem fetches/cycle: 0.215	misses: 1.00 speedup: 11.0 mem throughput: 11.0

6 Related work

Lee and DeVries investigate single-chip vector microprocessors as a way to exploit more parallelism with less hardware and reduced instruction bandwidth [25]. 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 [23]. Complementing this research, out-of-order, multi-threaded, and decoupled vector architectures have been proposed by Espasa, Mateo, and Smith [5, 6, 7] as methods to improve the performance of vector processing. The XMT architecture has much in common with the recent vector approaches, as we are solving many of the same problems in much the same way: we are looking at increasing the abundance of instruction-level parallelism by relying on the programming paradigm—the primary difference is in our use of a SPMD-style parallel algorithm programming model instead of a vector model.

Several architectures exist that range from moderately to highly decentralized. The multiscalar paradigm [10, 32] uses a uni-directional ring of processing elements in which each processing element need only communicate to its immediate neighbor: long-distance communication is mostly limited to servicing memory requests; even the address resolution mechanism is decentralized [11]. The XMT architecture differs in that hardware contexts never communicate directly with each other.

The edited book [17] is an excellent general reference for known approaches to multithreading. The approaches described there emphasize the case where in a given cycle instructions from only one thread can be issued. A very interesting architecture is presented in the Tera computer [1] which uses multithreading for memory latency hiding. The difference with our approach is that Tera focuses on supporting a plurality of threads by constantly switching among threads, but Tera does not issue instructions from more than one thread at the same cycle. This would limit the relevance of our multi-operand and spawn-join instructions for their architecture.

Simultaneous multi-threading [34] improves parallelism by issuing instructions from several concurrently executing threads to multiple functional units each cycle. The architecture is targeted at supporting standard programming (the simulations are based on Multiflow-compiled SPEC benchmarks), and proposed SMT designs have supported up to eight concurrently executing contexts, which is appropriate for most standard code. The goal of the XMT architecture is to support parallel programming, in addition to standard code, and so we envision much larger numbers of hardware contexts.

Single-chip multiprocessors [27] are analogous to symmetric multiprocessors and are aimed at exploiting thread-level and process-level parallelism, especially as supported by multitasking operating systems [15]. The M-Machine architecture [9] is a similar example. These architectures are inherently decentralized, as an implementation is comprised of several identical processor cores communicating through an on-chip level-2 cache. Raw processors [36] go a step further: rather than implementing medium-scale SMPs on a single chip, the Raw architecture is intended to implement an MPP on a single chip, supporting 128 nodes or more. Each processing node is very simple, and the nodes on a chip are connected in a 2D mesh network. The on-chip cache and even the on-chip register file is distributed among the processing elements. XMT differs from the Raw architecture in that resources local to a hardware thread context are not directly addressable by other hardware thread contexts, as they are in the Raw machines.

Multi-prefix primitives (*i.e.* prefix-sum, prefix-max, *etc.*) have been proposed for multiprocessor architectures (see the NYU-Ultracomputer [13], and SB-PRAM [21]), but not for instruction-level parallelism.

In contrast to [12], who studied fetch-and-increment in a multiprocessor setting, XMT discusses an on-chip implementation of fetch-and-increment and provides a sense for which fetch-and-increment gives wait-free coordination. Wait-free was stated as a desired goal in [12] who used fetch-and-increment for bottleneck-free coordination (which is weaker than wait-free). Another difference is the use of spawn-join. The paper [35] explains how our use of Arbitrary CRCW PRAM does not limit (relative to the stronger CRCW PRAM models) the algorithms that can be used for XMT and enabled its significant wait-free advantages. This contrasts with [12] whose weaker coordination concept supports a more general CRCW PRAM algorithmic model.

7 Concluding remarks

The performance of future IC technologies will be dominated by interconnect delays, therefore microprocessor architectures should aim for designs wherein most communication is localized. Functions that require the propagation of cross-chip signals should be few and infrequently used.

The research area of SPMD 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 IC technologies: a SPMD 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 an architecture can withstand high cross-chip communication costs. It can also take advantage of very large numbers of execution units, as opposed to traditional superscalar designs, which typically cannot make use of more than one or two dozen execution units.

Finally, we note that while significant speedups can be achieved by SPMD programs, the speedup of standard code often requires other techniques. The XMT "ideology" does not exclude using other methods within the design, since intra-thread parallelism can be exploited in addition to the explicit inter-thread parallelism of XMT.

References

[1] R. Alverson, D. Callahan, D. Cummings, B. Koblenz, A. Porterfield, and B. Smith. "The Tera computer system." *International Conference on Supercomputing.* 1990.

[2] R.C. Argarwal. "A Super Scalar Sort Algorithm for RISC Processors." *ACM-SIGMOD 96.* Montreal, Canada. June 1996.

[3] M. T. Bohr. "Interconnect scaling—the real limiter to high performance ULSI." In *Proc. 1995 IEEE International Electron Devices Meeting*, 1995, pp. 241–244.

[4] D. Burger and T. M. Austin. "The SimpleScalar tool set, version 2.0." Tech. Rep. CS-1342, University of Wisconsin-Madison, June 1997.

[5] R. Espasa and M. Valero. "Decoupled vector architectures." In *Proc. Second International Symposium on High Performance Computer Architecture (HPCA-2)*, San Jose CA, February 1996, pp. 281–290.

[6] R. Espasa and M. Valero. "Multithreaded vector architectures." In *Proc. Third International Symposium on High Performance Computer Architecture (HPCA-3)*, San Antonio TX, February 1997, pp. 237–248.

[7] R. Espasa, M. Valero, and J. E. Smith. "Out-of-order vector architectures." In *Proc. 30th Annual International Symposium on Microarchitecture (MICRO-30)*, Research Triangle Park NC, December 1997, pp. 160–170.

[8] K. I. Farkas, P. Chow, N. P. Jouppi, and Z. Vranesic. "The multicluster architecture: Reducing cycle time through partitioning." In *Proc. 30th Annual International Symposium on Microarchitecture (MICRO-30)*, Research Triangle Park NC, December 1997, pp. 149–159.

[9] M. Fillo, S. Keckler, W. Dally, N. Carter, A. Chang, Y. Gurevich, and W. Lee. "The M-Machine multicomputer." In *Proc. 28th Annual International Symposium on Microarchitecture (MICRO-28)*, Ann Arbor MI, November 1995, pp. 146–156.

[10] M. Franklin and G. Sohi. "The expandable split window paradigm for exploiting fine-grain parallelism." In *Proc. 19th Annual International Symposium on Computer Architecture (ISCA-19)*, Gold Coast, Australia, May 1992, pp. 58–67.

[11] M. Franklin and G. S. Sohi. "ARB: A hardware mechanism for dynamic reordering of memory references." *IEEE Transactions on Computers*, vol. 45, no. 5, pp. 552–571, May 1996.

[12] E. Freudenthal and A. Gottlieb. "Process Coordination with Fetch-and-Increment." *Fourth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-4).* Santa Clara, CA. April 1991.

[13] 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*, vol. 5, no. 2, 1983.

[14] L. Gwennap. "Digital 21264 sets new standard." *Microprocessor Report*, vol. 10, no. 14, October 1996.

[15] L. Hammond, B. Nayfeh, and K. Olukotun. "A single-chip multiprocessor." *IEEE Computer*, vol. 30, no. 9, pp. 79–85, September 1997.

[16] J. L. Hennessy and D. A. Patterson. *Computer Architecture: A Quantitative Approach, 2nd Edition*. Morgan Kaufmann Publishers, Inc., 1996.

[17] 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.

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

[19] N. P. Jouppi. "Cache write policies and performance." In *Proc. 20th Annual International Symposium on Computer Architecture (ISCA-20)*, May 1993, pp. 191–201.

[20] Kendall Square Research. *KSR/Series Parallel Programming*. Kendall Square Research Corporation, Cambridge MA, 1993.

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

[22] C. W. Kessler and H. Seidl. "Integrating synchronous and asynchronous paradigms: The Fork95 parallel programming language." Tech. Rep. 95-05, Fachbereich 4 Informatik, Univ. Trier, D-54286 Trier, Germany, 1995.

[23] C. Kozyrakis, S. Perissakis, D. Patterson, T. Anderson, K. Asanovic, N. Cardwell, R. Fromm, J. Golbus, B. Gribstad, K. Keeton, R. Thomas, N. Treuhaft, and K. Yelick. "Scalable processors in the billion-transistor era: IRAM." *IEEE Computer*, vol. 30, no. 9, pp. 75–78, September 1997.

[24] A. LaMarca and R. E. Ladner. "The Influence of Caches on the Performance of Sorting." *Proceedings of the Eighth Annual ACM-SIAM Symposium on Discrete Algorithms,* 1997. pp. 370-379.

[25] C. G. Lee and D. J. DeVries. "Initial results on the performance and cost of vector microprocessors." In *Proc. 30th Annual International Symposium on Microarchitecture (MICRO-30)*, Research Triangle Park NC, December 1997, pp. 171–182.

[26] D. Matzke. "Will physical scalability sabotage performance gains?" *IEEE Computer*, vol. 30, no. 9, pp. 37–39, September 1997.

[27] B. Nayfeh, L. Hammond, and K. Olukotun. "Evaluation of design alternatives for a multiprocessor microprocessor." In *Proc. 23rd Annual International Symposium on Computer Architecture (ISCA-23)*, Philadelphia PA, May 1996, pp. 67–77.

[28] R. S. Nikhil and Arvind. "Can dataflow subsume von Neumann computing?" In *Proc. 16th Annual International Symposium on Computer Architecture (ISCA-16)*, Jerusalem, Israel, May 1989, pp. 262–272.

[29] K. Saraswat and F. Mohammadi. "Effect of scaling interconnections on the time delay of VLSI circuits." *IEEE Transactions on Electron Devices*, vol. 29, no. 4, pp. 645–650, April 1982.

[30] Semiconductor Industry Association. *The National Technology Roadmap for Semiconductors, 1997 Edition*. SIA: The Semiconductor Industry Association, 1997.

[31] A. Silberschatz and P. B. Galvin. *Operating System Concepts, Fifth Edition.* Addison Wesley Longman, Inc. 1998. p. 384

[32] G. S. Sohi, S. E. Breach, and T. N. Vijaykumar. "Multiscalar processors." In *Proc. 22nd Annual International Symposium on Computer Architecture (ISCA-22)*, Santa Margherita Ligure, Italy, June 1995, pp. 414–425.

[33] STREAM. *STREAM: Measuring Sustainable Memory Bandwidth in High Performance Computers*. The University of Virginia, http://www.cs.virginia.edu/stream/, 1997.

[34] 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." In *Proc. 23rd Annual International Symposium on Computer Architecture (ISCA-23)*, Philadelphia PA, May 1996, pp. 191–202.

[35] U. Vishkin, S. Dascal, E. Berkovich, and J. Nuzman. “Explicit Multi-threaded (XMT) bridging models for Instruction Parallelism.” In *Proc. of Tenth Annual ACM Symposium on Parallel Algorithms and Architectures (SPAA)*. Puerto Vallarta, Mexico. July 1998

[36] E. Waingold, M. Tayor, D. Srikrishna, V. Sarkar, W. Lee, V. Lee, J. Kim, M. Frank, P. Finch, R. Barua, J. Babb, S. Amarasinghe, and A. Agarwal. “Baring it all to software: Raw machines.” *IEEE Computer*, vol. 30, no. 9, pp. 86–93, September 1997.

Appendix: Comparing parallel programming/algorithms approaches

	length 1 “threads”	**longer threads**	
Examples	Vector	XMT	Existing multi-threading Many, including KSR, SMT
Programming/ algorithms	Priority CRCW PRAM, Vector programming & compiling	Arbitrary CRCW PRAM (SPMD)	Shared memory programming, Message passing programming (e.g. SPMD)
Synchronization	Each step	Joins only	Many forms
To substitute one Spawn-Join for existing SPMD	many steps	a few Spawn-Joins	same (i.e., one Spawn-Join)
Thread interaction	special rules (concurrent read-write conventions)	no-busy-wait prefix-sum	many forms

Advantages

Vector: Mature compiler technology. Fast when data availability is predictable (e.g. where data access is regular). Parallel algorithms knowledge base applies.
XMT: Parallel algorithms knowledge base applies. Each thread can progress at its own speed. Resilient to slow execution (and upredictable data access time) of some instructions.
Existing multi-threading: Suits existing serial and parallel machines.

Disadvantages

Vector: Slow for unpredictable/irregular data access: “everything is delayed” if one instruction is slow to finish. May encourage algorithms that result in more data movements.
XMT: May encourage algorithms that will result in more data movements than for existing multi-threading; however, for the same algorithm, usually does not require more data movements than the other two approaches
Existing multi-threading: More difficult to program. Smaller algorithmic knowledge base.