XMT-M: A Scalable Decentralized Processor

Efraim Berkovich, Joseph Nuzman, Manoj Franklin, Bruce Jacob, and Uzi Vishkin
Department of Electrical and Computer Engineering, and
University of Maryland Institute for Advanced Computer Studies (UMIACS)
University of Maryland, College Park, MD 20742

Abstract

A defining challenge for research in computer science and engineering has been the ongoing quest for *reducing the completion time of a single computation task.* Even outside the parallel processing communities, there is little doubt that the key to further progress in this quest is to do parallel processing of some kind. A recently proposed parallel processing framework that spans the entire spectrum from (parallel) algorithms to architecture to implementation is the explicit multi-threading (XMT) framework. This framework provides: (i) simple and natural parallel algorithms for essentially every general-purpose application, including notoriously difficult irregular integer applications, and (ii) a multi-threaded programming model for these algorithms which allows an "independence-of-order" semantics: every thread can proceed at its own speed, independent of other concurrent threads. To the extent possible, the XMT framework uses established ideas in parallel processing.

This paper 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.

Keywords: Fine-grained SPMD, independence of order semantics, instruction-level parallelism (ILP), no-busy-wait finite state machines, parallel algorithms, prefix-sum, and spawn-join.

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 [9].

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) [19] 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 [9], *"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] [8] [18] 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 [8], 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 [29] 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].

[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 [8]. 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.

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* [31]. By wire delays we mean on-chip delay of connections between gates. The Semiconductor Industry Association estimates that, within a decade, only 16% of a chip will be reachable in a clock cycle [31]. 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.

2 Explicit Multi-Threading (XMT)

The XMT framework is grounded in a rather ambitious vision aimed at on-chip general-purpose parallel computing [29]; 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.

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 [18]. The XMT threads can be moderately long, providing some locality of reference[2]. Figure 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 occurs only at the Joins. We say that the XMT programming model inherits

[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.

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.

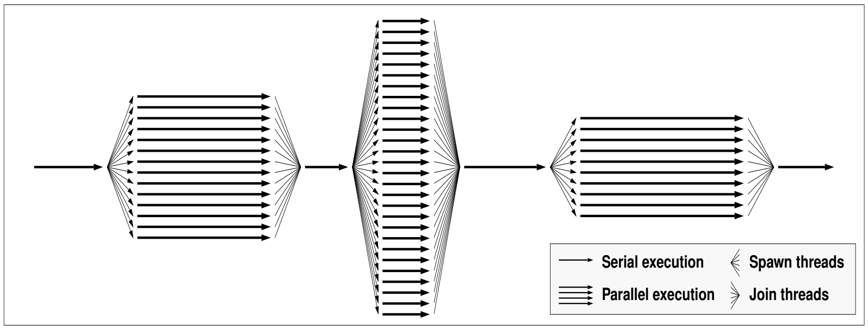


Figure 1: Parallelism Profile for the XMT Model

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 [14], and has the following semantics:

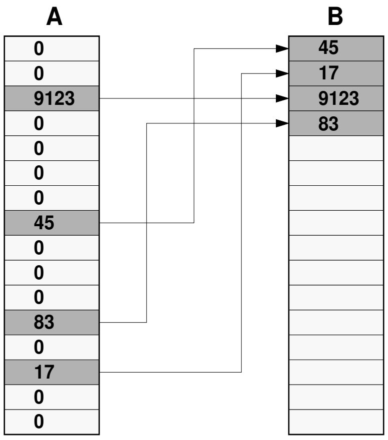
`Prefix-sum B, R;      (i) B = B + R and (ii) R = initial value of B`

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 2). The right hand side of Figure 2 gives a XMT high-level language code to do this array compaction.

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.




```
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 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.

2.3.1 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.

2.3.2 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.

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.

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 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.

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 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.

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 [28]. There is a dedicated prefix-sum bus for broadcasting the prefix-sum

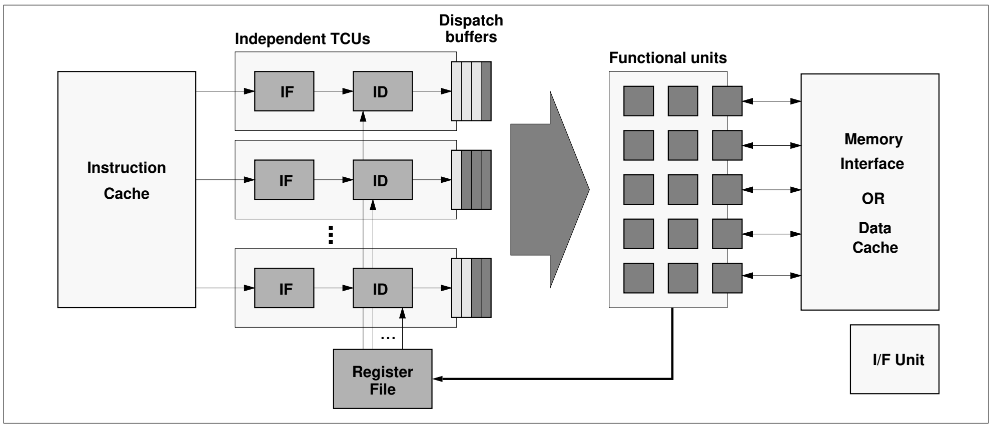


Figure 3: **An XMT-M Cluster.** Each XMT-M cluster includes a shared instruction cache, a shared data cache, multiple TCUs, and a number of shared functional units.

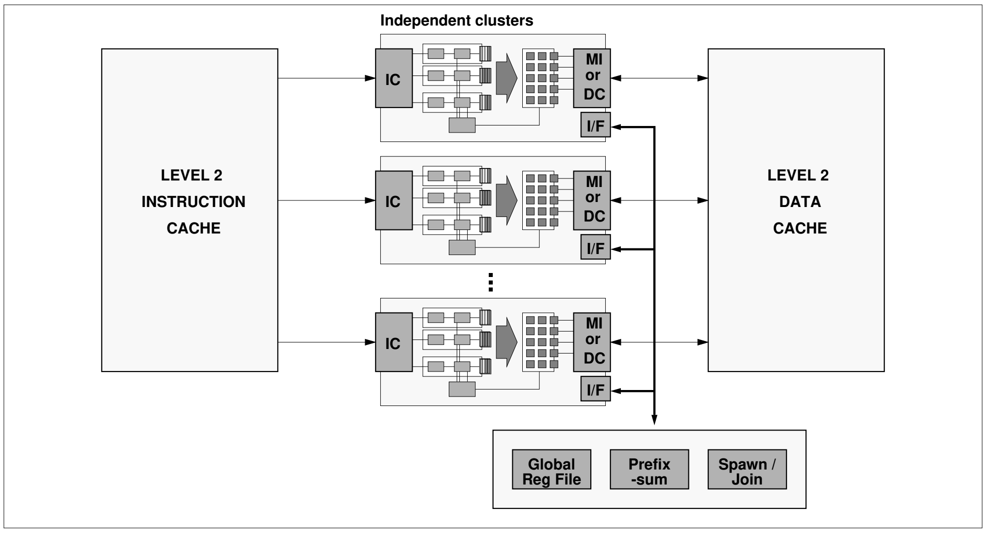


Figure 4: **An XMT-M Processor.** Communication paths in this diagram require multiple cycles.

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 [28]. 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.)

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 5.

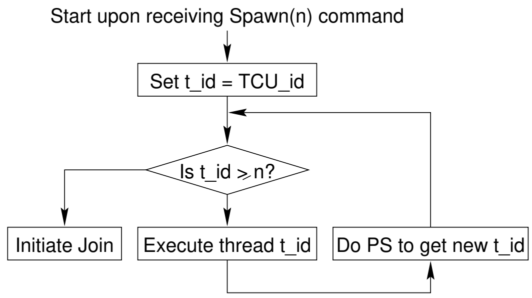


Figure 5: Flowchart depicting activities of a TCU upon receiving a Spawn command

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.

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.

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.

3.4.1 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.

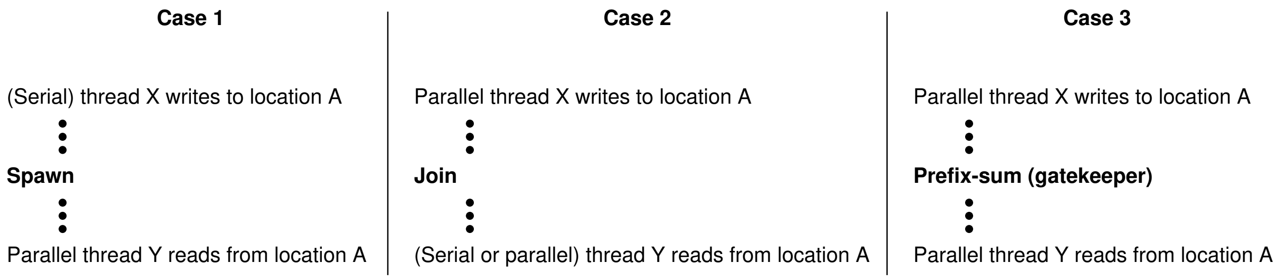


Figure 6: Cache Coherence Hazards for XMT

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 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 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.

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.

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.

4.1.1 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.

4.1.2 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.

4.1.3 Benchmarks and Performance Metrics

For benchmarks, we use a collection of code snippets with varying degrees of inherent parallelism, as described in Figure 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

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 [26], 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 [25]. 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 [18] for the 50 and 500 sized inputs and (2) the no-cut coin-tossing algorithm for the 5K and 250K sized inputs. The work [10] 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 [1], while for others other sorting routines are fastest [23].	50 item array, 500 item array, 5K item array, 250K item array.

Figure 7: Benchmarks

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 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).

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 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.

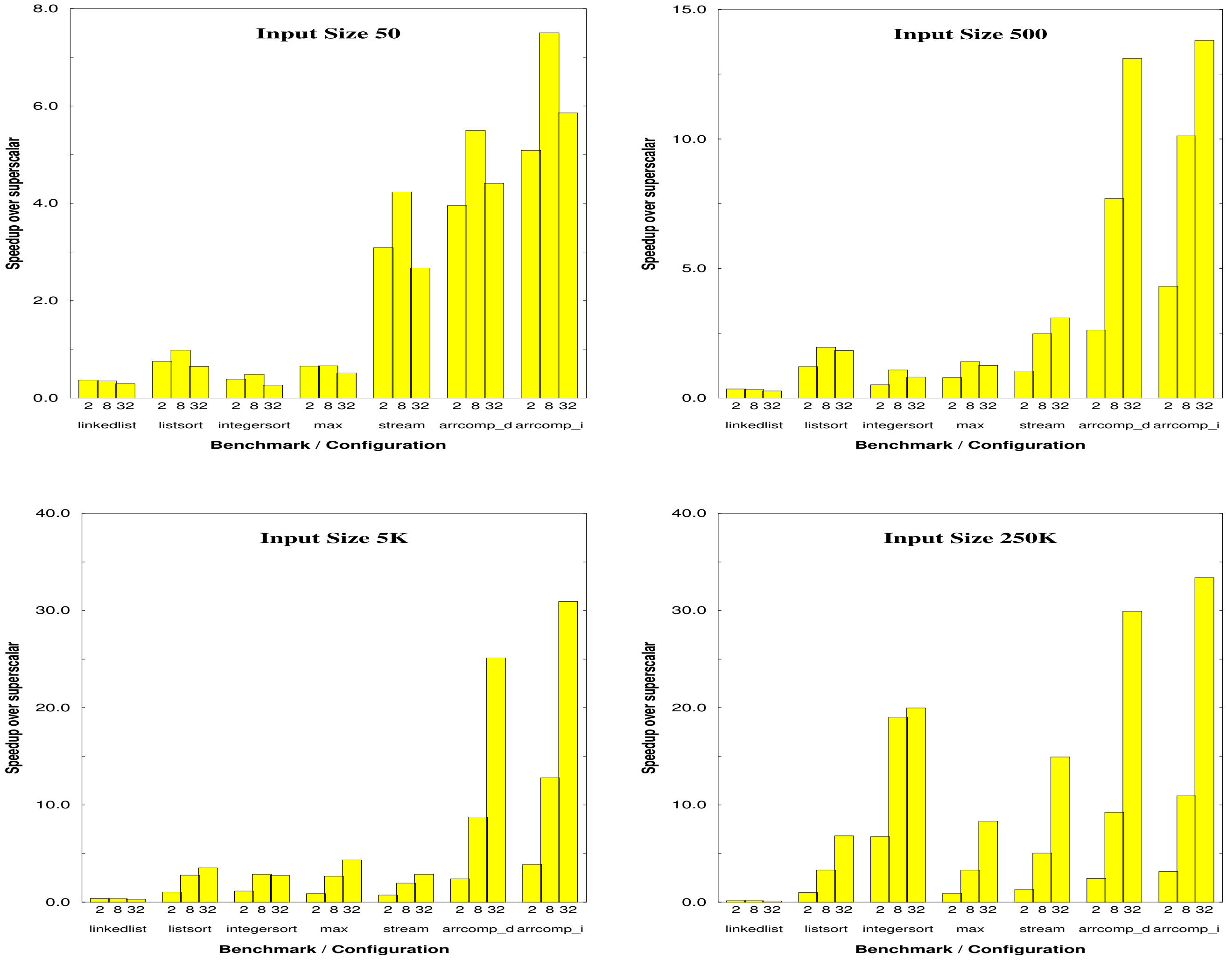


Figure 8: XMT-M Speedups relative to Serial Computing

Let us look at the results of Figure 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 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.

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 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 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.

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 [7] and [15] 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 [20] and [21]. Basic insights concerning the use of a prefix-sum like primitive go back to the Fetch-and-Add [16] or Fetch-and-Increment [14] primitives (cf. [2]). Insights concerning nested Spawns rely on the work of Guy Blelloch's group [4], [5] and others.

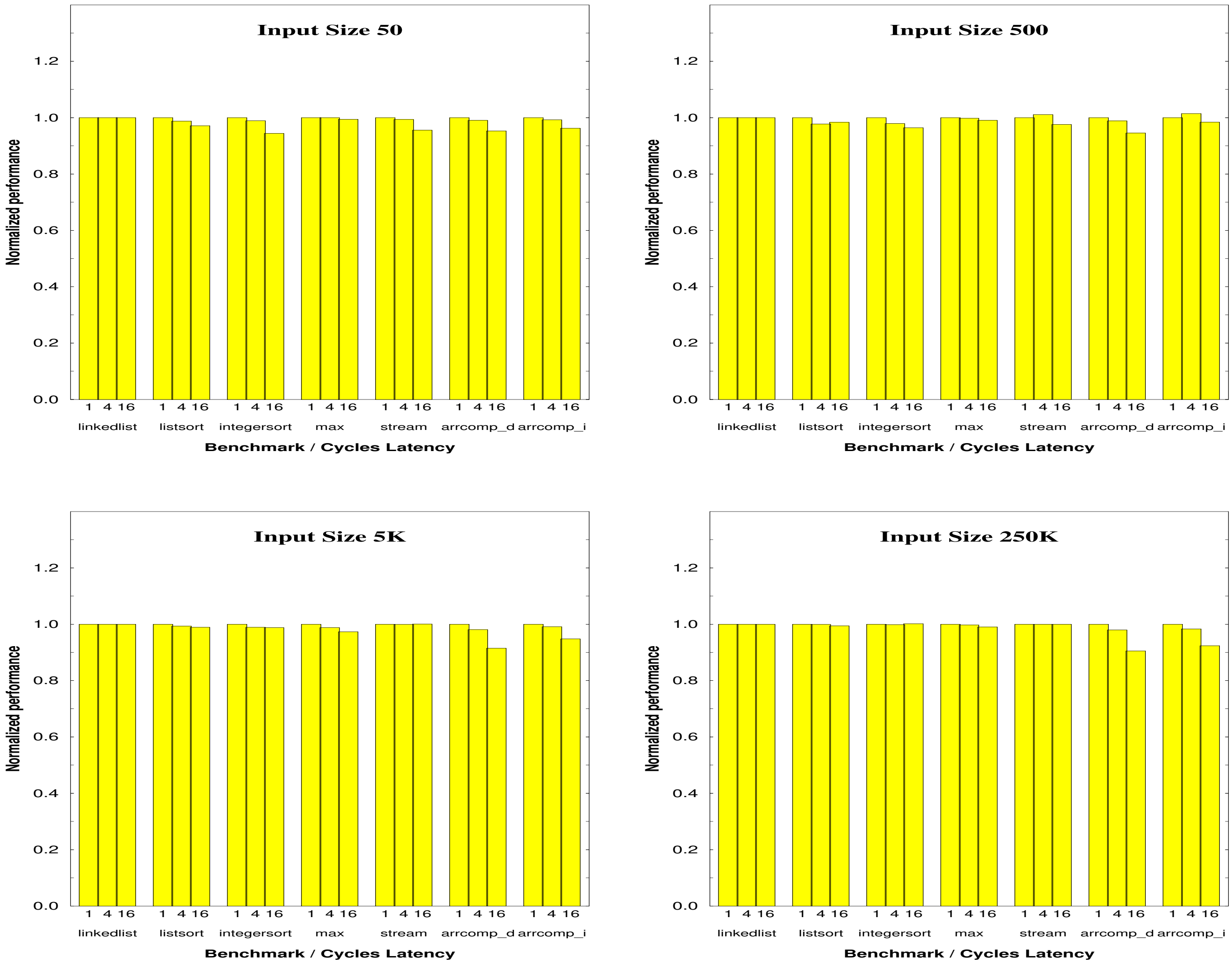


Figure 9: Effect of Cross-Chip (Global) Communication Latency on XMT-M Performance

The U. Wisconsin Multiscalar project [13] and the U. Washington simultaneous multi-threading (SMT) project [27] with their use of multiple program counters and the computer architecture literature on multi-threading (see, for instance [17]) 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 [27] 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 [30].

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 [24]. 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 [22]. Complementing this research, out-of-order, multi-threaded, and decoupled vector architectures have been proposed by Espasa, Mateo, and Smith [11] [12] 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.

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 [9], 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 [29] 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.

References

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

[2] G.S. Almasi A. Gottlieb. *Highly Parallel Computing, Second Edition.* Benjamin/Cummings, 1994.

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

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

[5] 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.

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

[7] R. Cole and O. Zajicek, "The APRAM: incorporating asynchrony into the PRAM model," *Proc. 1st ACM-SPAA*, pp. 169-178, 1989.

[8] D. E. Culler and J. P. Singh, Parallel Computer Architecture. Morgan Kaufmann, 1999.

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

[10] S. Dascal and U. Vishkin, "Experiments with List Ranking on Explicit Multi-Threaded (XMT) Instruction Parallelism," *Proc. 3rd Workshop on Algorithms Engineering (WAE-99)*, July 1999, London, U.K. Downloadable from http://www.umiacs.umd.edu/~vishkin/XMT/.

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

[12] 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.

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

[14] 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)*, 1991.

[15] P.B. Gibbons. "A more practical PRAM algorithm," *Proc. 1st ACM-SPAA*, pp. 158-168, 1989.

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

[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] 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.

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

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

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

[23] 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.

[24] 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.

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

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

[27] 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.

[28] 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.

[29] 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 http://www.umiacs.umd.edu/~vishkin/XMT/

[30] Personal communication with H. Levy, August 1997.

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