Towards a First Vertical Prototyping of an Extremely Fine-grained Parallel Programming Approach

Dorit Naishlos[1 3 * †], Joseph Nuzman[2 3 *], Chau-Wen Tseng[1 3], Uzi Vishkin[2 3 4 *]

[1] Dept of Computer Science, University of Maryland, College Park, MD 20742

[2] Dept of Electrical and Computer Engineering, University of Maryland, College Park, MD 20742

[3] University of Maryland Institute of Advanced Computer Studies, College Park, MD 20742

[4] Dept of Computer Science, Technion - Israel Institute of Technology, Haifa 32000, Israel

{dorit, jnuzman, tseng, vishkin}@umiacs.umd.edu

ABSTRACT

Explicit-multithreading (XMT) is a parallel programming approach for exploiting on-chip parallelism. XMT introduces a computational framework with 1) a simple programming style that relies on fine-grained PRAM-style algorithms; 2) hardware support for low-overhead parallel threads, scalable load balancing, and efficient synchronization. The missing link between the algorithmic-programming level and the architecture level is provided by the first prototype XMT compiler. This paper also takes this new opportunity to evaluate the overall effectiveness of the interaction between the programming model and the hardware, and enhance its performance where needed, incorporating new optimizations into the XMT compiler. We present a wide range of applications, which written in XMT obtain significant speedups relative to the best serial programs. We show that XMT is especially useful for more advanced applications with dynamic, irregular access pattern, where for regular computations we demonstrate performance gains that scale up to much higher levels than have been demonstrated before for on-chip systems. When evaluating performance results in this paper, it is important to remember that they have been obtained using a simulator, but only an implementation could establish the actual overall performance of the architecture. Complex issues, such as the potential interference between the various new hardware mechanisms and existing superscalar-processor implementations, have been too hard to evaluate by simulations alone.

Keywords

Parallel programming, compilers, processor architecture.

1. INTRODUCTION

The challenge of exploiting the large and fast growing number of transistors available on modern chips has motivated the exploration of parallel architectures. Researchers have considered parallelism in two main categories. The first is at an ultra fine-grained level, among instruction sequences. However, the limited amount of instruction-level parallelism (ILP) present in programs, and the difficulties to uncover it [15] [33] prevent computers from fully exploiting the available on-chip resources. The peak amount of ILP (as represented by the maximum multiple issue of instructions per clock) in popular microprocessors has remained flat for the last decade. For documentation of this fact in 7 of the last 10 years, compare Figure 4.60 in [14] with Figure 3.61 in [15].

An alternative source of parallelism is at the level of coarse-grained multithreading, and has been traditionally expressed under the shared memory or the message-passing programming models, either directly by programmers, or with the aid of parallelizing compilers.

---

[*] Supported by NSF grant 9820955

[†] Current address: IBM Haifa Labs, Haifa University Campus, Mount Carmel, Haifa 31905, Israel

Two interesting representative design points for multi-threaded single-chip architectures are: chip multiprocessors (CMP) such as the Stanford Hydra [13], and Simultaneous Multithreading (SMT) [27]. A typical CMP architecture features independent processing units that share an L2 cache, each executing a different thread. The observation that resource utilization in a CMP architecture is limited by the available ILP within a single thread, has led the SMT design to permit all threads to share the processor functional units, allowing to exploit ILP across different threads at a single cycle.

While these new architectures have the ability to exploit fine-grained parallelism, both CMP and SMT report difficulties to obtain fine-grained multi-threaded codes for today's important uniprocessor programs [12]. On one hand, parallelizing compilers have been only partially successful at automatically converting serial codes. On the other hand, the programming models currently available to programmers were originally designed to target multi-chip parallel architectures, and therefore do not fit a single-chip environment. Current CMP and SMT related research concentrates its effort on developing advanced compiler techniques for fine-grained parallelization, and support for speculative execution.

XMT (explicit multi-threading) attempts to bridge the gap between the ultra fine-grained on-chip parallelism and the coarse-grained multi-threaded program by proposing a framework that encompasses both programming methodology and hardware design. XMT tries to increase available ILP using the rich knowledge base of parallel algorithms. Relying on parallel algorithms, rather than on a compiler, programmers express extremely fine-grained parallelism at the thread level. While some XMT architectural design aspects share motivations with the other proposed designs, CMP and SMT do not address programmability and their threading support is not targeted towards supporting a PRAM-style program. These projects tend to center their attention on multiprogramming and increasing the IPC (instructions per clock) rate, rather than reducing a single-task execution time and measuring speedups relative to the serial program, which is the focus of XMT.

In addition, XMT aspires to scale up to much higher levels of parallelism than other single-chip multithreaded architectures consider currently; for example, where CMP and SMT presentations typically discuss 4 to 8 processing units, special XMT hardware gadgets, such as one which allows fast parallel prefix-sum, allow us in this paper to examine configurations of up to 256 Thread Control Units (TCUs). Tile based architectures, such as MIT's Raw [35], and Stanford's Smart-Memories [17], also expect to scale to high levels of parallelism. However, Raw utilizes a message-passing model rather than the shared-memory model of XMT. In addition, Raw heavily relies on compiler technology to manage data distribution and movements between tiles. As such, it is much easier to program for the XMT architecture, and it is also expected to address a wider range of applications. Smart-Memories uses an architecture that can be tailored to match different computational requirements. Like XMT, it identifies scalability and general-purpose computing as two major design goals. XMT however advocates higher levels of abstraction, for which it is easier to program, yet, we argue that performance is still good.

The last point of comparison, the Tera (now Cray) Multi-Threaded Architecture (MTA) [1], also supports many threads on a given processor. There, however, the processors switch between threads to hide latencies, rather than running multiple threads concurrently. Moreover, the MTA, like other MPP machines, is designed for big computations with large inputs. XMT aims to achieve speed-ups for smaller input computations, such as those which might be encountered on desktop hardware.

Previous papers on XMT have discussed in detail its fine-grained SPMD multi-threaded programming model, architectural support for concurrently executing multiple contexts on-chip, and preliminary evaluation of several parallel algorithms using hand-coded assembly programs [29] [9]. The introduction of an XMT compiler, presented here, allows us to evaluate XMT for the first time as a complete environment ("vertical prototyping"), using a much larger benchmark suite (with longer codes) than before. Due to the rather broad nature of our work, specialized parts of the work – the evaluation of the XMT compiler and evaluation of the programming model – were published in two respective specialized workshops [19], [20]. This paper incorporates the feedback from these workshops, and is the first one to present the whole work, including the integrated results, as well as the interplay between the programming model and the other components of XMT (compiler and architecture).

We begin by reviewing the XMT multi-threaded programming model and architecture in section 2. Section 3 presents the prototype XMT compiler and code generation model. We then describe the XMT simulator and experimental methodology in section 4, and evaluate the efficiency of our implementation in section 5. Section 6 discusses compiler optimizations to coarsen threads. Section 7 puts it all together by revisiting the XMT programming features, and evaluating to what extent the hardware and compiler are able to support them efficiently. Section 8 discusses some performance issues and how the current paper addresses a long standing open problem, and section 9 concludes.

2. THE XMT FRAMEWORK

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

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

2.1 XMT Programming Model

The programming model underlying the XMT framework is an arbitrary CRCW (concurrent read concurrent write) SPMD (single program multiple data) programming model. In the XMT programming model, an arbitrary number of virtual threads, initiated by a *spawn* and terminated by a *join*, share the same code. The arbitrary CRCW aspect dictates that concurrent writes to the same memory location result in an arbitrary one committing. No assumption needs to be made beforehand about which will succeed. This permits each thread to progress at its own speed from its initiating spawn to its terminating join, without ever having to wait for other threads; that is, no thread busy-waits for another thread. The implied “independence of order semantics” (IOS) allows XMT to have a shared memory with a relatively weak coherence model. An advantage of using this easier to implement SPMD model is that it is also an extension of the classical PRAM model, for which a vast body of parallel algorithms is available in the literature. (Previous XMT papers related the relaxation in the synchrony of PRAM algorithms to works such as [6] on asynchronous PRAMs).

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

The XMT-C high-level language is an extension of standard C. The extensions are described individually in Appendix B. A parallel region is delineated by spawn and join statements. Synchronization is achieved through

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

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

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

The SpawnMT model of [29] does not allow for nested initiation of an arbitrary-size spawn within a parallel spawn region. Such a feature, while useful, would be difficult to realize efficiently with hardware support. As an alternative, [30] extended the programming model to support a fork operation. A thread can perform a fork operation to introduce a new virtual thread as work is discovered. Forks must be executed one at a time by a single thread, but forks from multiple threads can be performed in parallel. The fork extension allows the programmer to approach many problems in a more asynchronous and dynamic manner. In XMT-C, *fspawn* is used when forking may be necessary, and *xfork* performs the fork operation.

The attempt to develop general-purpose parallel algorithms that map well to different parallel machines, has led researchers to utilize different parallel models. In [23], the QSM model is used to design and analyze general-purpose parallel algorithms, along with a suitable cost metric (which is shown to be accurate for big problem sizes). The algorithms they present are adaptations of PRAM algorithms and their mapping to other parallel models is discussed. XMT is also motivated by the attempt to provide a way to map general-purpose parallel algorithms to a real parallel machine. The XMT architecture is designed to provide such a prototype target machine for which it is easy to map PRAM algorithms, using the XMT programming model.

MIT's Cilk [11] also provides a multi-threaded programming interface and execution model, however, there are two important differences in scope. First, since Cilk is targeted at compatibility with existing SMP machines, load balancing in software was important. XMT provides hardware support to bind virtual threads to thread control units (TCUs) exactly as the TCUs become available. The low-overhead of XMT is designed to be applicable to a much broader range of applications. Second, Cilk presents a programming model that tries to match very closely standard serial programming constructs, where forking a thread takes the form of a function call. While XMT also bases its programming model on standard C, the programmer is expected to rethink the way parallelism is expressed. The wide-spawn capabilities and prefix-sum primitive are present to support the many algorithms targeted to the PRAM model. One has to keep in mind that XMT is PRAM-like programming, but not exactly PRAM programming.

2.2 The XMT Architecture

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

The XMT programming model allows programmers to specify an arbitrary degree of parallelism in their code. Clearly, real hardware has finite execution resources, so in general all threads cannot execute simultaneously. In

an XMT machine, a thread control unit (TCU) executes an individual virtual thread. Upon termination, the TCU performs a prefix-sum operation in order to receive a new thread ID. The TCU will then emulate the thread with

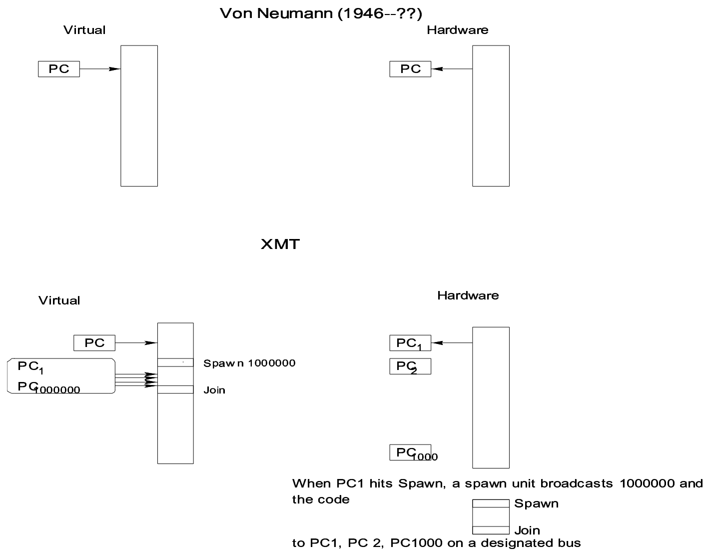

Figure 1a: Program counter + stored program

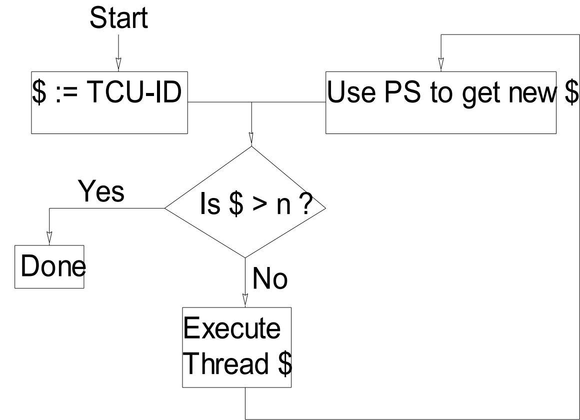

Figure 1b: TCU program snippet

Figure 1: XMT execution model

that new ID. All TCUs repeat the process until all the virtual threads have been completed.

Figure 1 illustrates this: (i) through a comparison with the von-Neumann stored program and program counter apparatus (Figure 1a), and (ii) through a snippet of the program of a TCU (Figure 1b).

We begin with Figure 1a. Its *upper part,* entitled "von-Neumann (1946--??)," illustrates the program counter apparatus in serial machines, which has dominated general-purpose computing since 1946, with no end to in sight to its reign. The *right hand side* (of the upper part of Figure 1a) depicts the hardware apparatus, where one command at a time is brought to the program counter. The *left hand side* (of the upper part of Figure 1a) demonstrates how the programmer is often educated to think about this apparatus--"the virtual outlook". Here the program counter is the one to move; it moves from one location of the memory to another, perhaps like a "book analogy", where the finger of a reader advances from one line of the book to another. The fact that this von-Neumann apparatus has survived orders of magnitude improvements in speed since the 1940s makes it a remarkable "Darwinistic success story". For this reason we sought to upgrade, rather than replace in a disruptive manner, this successful apparatus.

The *lower part* of Figure 1a, entitled "XMT", illustrates the new apparatus. The *left hand side* (of the lower part of Figure 1a) depicts the virtual description. There is still one computer program as in the von-Neumann apparatus. In the above book analogy, finger (program counter) number 1 moves from one line of the book to another, until it reaches a special command called Spawn. The Spawn command specifies a number of "threads" which can be performed in parallel. Since we discuss now the virtual side, any number of threads can be specified. Figure 1a mentions 1000000 threads. The virtual threads, initiated by a Spawn and terminated by a Join, share the same code. At run-time, different threads may have different lengths, based on individual control flow decisions. The programmer's understanding will be that each of the threads can progress (guided by one finger per thread) from the Spawn command to a subsequent Join command at its own speed. At the Join, the thread expires. Once all the virtual threads expire, finger number 1 continues. The main difference in the hardware description on the *right hand side* (of the lower part of Figure 1a), is that the number of program counters is fixed (the figure mentions 1000), and does not change as a function of the Spawn command at hand. When program counter number 1 executes a Spawn command it broadcasts the instructions of a thread (i.e., until

a Join command) to the other program counters. The broadcast is done on a broadcast fabric, which is very efficient. The program counters start by executing the first 1000 among the 1000000 threads, one thread each. When a program counter completes its thread, it starts executing one of the yet-to-be-executed threads. This is done until all of the 1000000 threads finish.

Figure 1b illustrates the program of a TCU. Suppose that n = 1000000 threads are to be executed as a result of a Spawn command. The figure assumes that n, and the SPMD code, were broadcast to all TCUs. TCU i starts by executing the respective virtual thread i, but only if i is not larger than n. Upon finishing the execution of a virtual thread, the TCU uses a prefix-sum computation to obtain the ID of the next virtual thread it should execute, and proceeds to execute it if that ID is not larger than n. Note that the only communication among TCUs above was through the prefix-sum computation.

This functionality is enabled by support at the instruction set level. With this architecture, all TCUs independently execute a serial program. Each accepts the standard MIPS instructions, and possesses a standard set of MIPS-type registers locally. The expanded ISA includes a set of specialized global registers, called prefix-sum registers (PR), and a few additional instructions. New instructions are used for thread management (Appendix B may be consulted for a glossary which includes separate descriptions of these new instructions). A spawn instruction activates all TCUs and broadcasts a new PC at which all TCUs will start executing. The *pinc* instruction operates on the PR registers, and performs a parallel prefix-sum with value 1. A specialized global prefix-sum unit can handle multiple *pinc*'s to the same PR register in parallel. Simultaneous *pinc*s from different TCUs are grouped, and the prefix-sum is computed and broadcast back to the TCUs. A prefix-sum functional unit with 64 single-bit inputs is very similar to a 64-bit integer adder (please see Appendix A or [28] for additional details on this unit); in each cycle only the differential increments are calculated and sent back to the clusters. This process is pipelined and completes within a constant number of cycles. *pread* performs a parallel read (prefix-sum with value 0) of a (replicated) PR register, and *pset* is used (serially) to initialize a PR register. The *psm* instruction allows for communication and synchronization between threads. It performs a prefix-sum operation with an arbitrary increment to any location in memory. It is an atomic operation, but due to hardware limitations, operations to the same base address are not performed in parallel (i.e., concurrent *psm*'s will be queued). This is similar to a fetch-and-increment [10] primitive (cf. [2]). A *ps* command at the XMT-C level is translated to a *psm* instruction.

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

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

The XMT model of computation provides an opportunity to break through the memory-bound limitations of serial computation and make more efficient use of memory. We consider how to revisit memory subsystem design for this very different computational model within the constraints of future chip technology. The copious transistor budgets enabled by future chip technology will allow for enough on-chip RAM to hold rather large working sets. Additionally, connectivity within a chip has the potential to provide as much communications bandwidth as needed.

The PRAM-like model presented by XMT demands that the memory subsystem support the simultaneous access by many threads to arbitrary locations in a shared memory space. It is clear that a large, monolithic cache could never support the demands of such a system. Also, since memory speed drops as size and number of ports increases, it pays to use smaller independent modules.

By instantiating enough small cache modules that can all be accessed simultaneously, one can allow for sufficient bandwidth to satisfy many threads at once. However, the memory subsystem must present a coherent global memory space. Maintaining coherence among potentially hundreds of modules appears to be too difficult to implement efficiently. Any sort of snooping scheme would incur a huge bandwidth cost. A directory-based cache coherence scheme would quickly become the limiting factor in data access. To sidestep the coherence issues, the memory address space may be partitioned among the memory modules (Figure 2). By specifying a single, fixed physical location for each address, we eliminate the need to keep values coherent between different modules. To minimize collisions among accesses from various threads, we use randomized hashing to distribute addresses among the memory modules. By distributing data at the granularity of blocks, spatial locality within a block can still be exploited.

One can make effective use of off-chip memory by extending the address partitioning to independent memory modules that serve the on-chip caches. By supporting multiple outstanding external requests, off-chip bandwidth

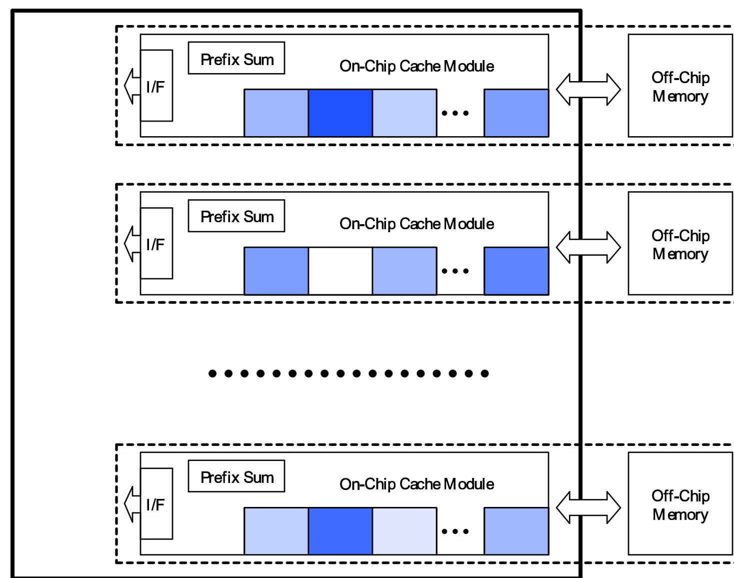

Figure 2: Parallel memory modules

can be fully utilized, even for non-streaming types of applications.

Support for multiple simultaneous prefix-sum-to-memory (*psm*) operations can be provided by incorporating an adder to each memory module, with serial access within each module. As mentioned earlier, support of many simultaneous accesses to a single register (*pinc*), is implemented using a separate interconnect which incorporates parallel prefix-sum hardware.

A substantial challenge for an XMT design is to provide connectivity between the many execution units and the many memory modules on-chip. While the capacity for sending signals increases with each technology shrink, the latency for propagating signals down a long wire is increasing. Due to the memory model supported, memory requests can travel to any memory location on the chip. A latency cost for such memory accesses cannot be avoided. Fortunately, the independence of order characteristic of XMT threading allows for such latency to be

tolerated. If enough thread contexts are instantiated simultaneously, while many threads wait for memory requests to propagate through the network, execution resources can be kept busy by other threads. For this to be effective, it is important that many simultaneous requests be supported by pipelining throughout the interconnection network.

The memory subsystem interconnect (along with the hardware prefix-sum mechanism) is one of a very few global resources in an XMT design. Providing a centralized scheduling resource to coordinate communication would be very costly for a large design. Additionally, driving a fast global clock across a deep-submicron chip is also very difficult and power consumptive. Both problems may be solved using a decentralized routing scheme (Figure 3). The basic topology of a crossbar can be maintained, with routing at each destination performed locally by a "tree of gatekeepers". This tree of gatekeepers is formed from locally synchronous switch points that select from two inputs depending on which arrives first. The gatekeepers structure mimics the multiplexor tree of a synchronous crossbar, but allows requests to proceed locally as they are able, rather than using a global schedule. There is a wire cost to the more localized scheme, as sources and destinations must be encoded and interpreted (the upcoming thesis [22] will provide a detailed solution; in one possible implementation of a global memory interconnect, a clock may be eliminated entirely, through the use of asynchronous switching and micropipelines [26] [21]). The hardware cost of tagging and local switching structures is easily supported by the benefits of such an asynchronous or loosely synchronous structure.

As mentioned earlier, the primary thrust of the memory architecture is to provide enough capacity to hide the

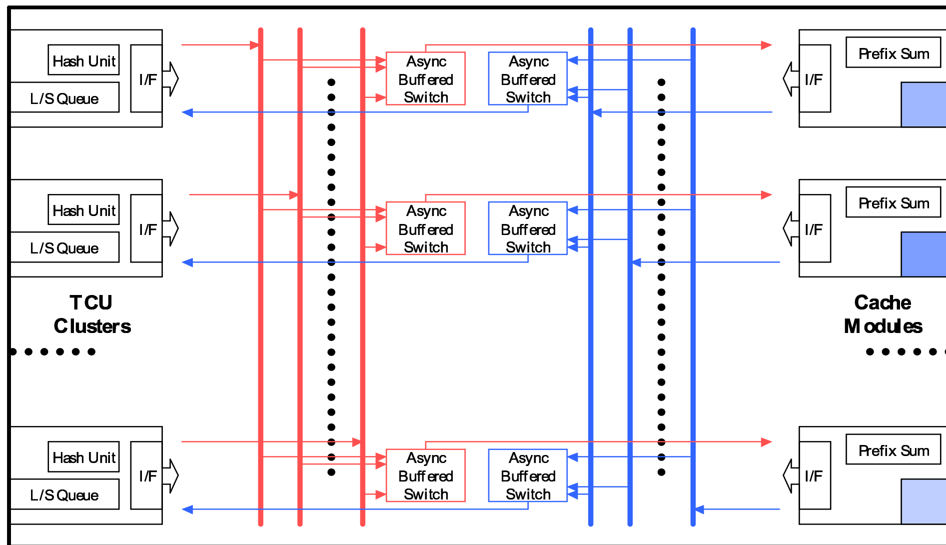

Figure 3: Memory system interconnect

latency of shared memory access. However, in some cases the latency can be reduced using a cluster-local L1 cache. For example, values in an L1 cache can be trusted locally between synchronization points (prefix-sums).

Reiterating a point from Figure 1, we note that supporting instruction fetch bandwidth for an XMT system is a substantially different problem than supporting data access, and should not be significantly more demanding overall than for serial computing. We can assume three things: (i) memory access is read-only, (ii) a thread's access patterns exhibit significant locality and predictability, (iii) the SPMD programming model suggests threads will usually be accessing the same locations. Considering these points, it is evident that fetching can occur from thread-local, non-coherent instruction caches. These caches can be backed by a shared low-

bandwidth broadcast interconnect. The first TCU (the one which initiates the spawn) can communicate instructions to be broadcast. This implementation targets spawn blocks whose static size is small enough to be effectively cached locally. Arbitrary instructions can be fetched through the data memory network, but such accesses will of course hurt performance.

3. THE XMT COMPILER

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

Parallel execution in the XMT architecture requires handling 1) Transition to parallel mode- activating all the TCUs and setting up their environment; 2) Thread creation and termination - emulate the virtual threads on each TCU – obtain a thread ID for each, and verify that it is a valid ID (i.e., less than the spawn size); 3) Transition back to serial mode - detect when all threads have terminated, and resume serial execution.

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

The prototype XMT compiler consists of two phases: 1) The front end ("Xpass") - a source-to-source translator based on SUIF [34]. This phase converts the XMT code with its parallel constructs into regular C code with specialized assembly templates for run-time threading support. 2) The back end (gcc) - builds an executable for the C code produced by Xpass. As we based our simulator implementation on the SimpleScalar ISA, we used the version of gcc from the SimpleScalar 2.0 package – gcc 2.6.3. Figure 4 summarizes the compilation process.

The general scheme used by Xpass is based on transforming parallel codes into parallel procedures. The compiler transforms the parallel region (the code in the spawn-join block) into the body of the procedure. When the procedure is called, the processing units are awakened, and each starts to execute the procedure body, which emulates the threads on each TCU. Figure 5 presents a high level example of the transformations performed by our compiler.

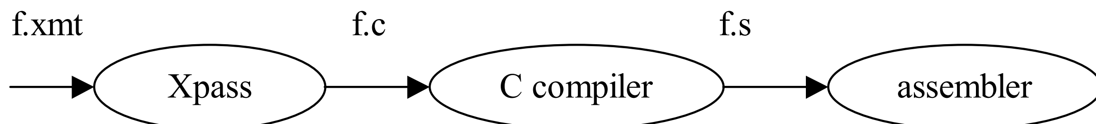

Figure 4: XMT compilation system

Original Xmt-C program

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

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

Transformed to

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

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

            **THREAD-CODE**

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

Figure 5: XMT code shape

Producing this structure involves two tasks: 1) Outlining. Detect all parallel regions (spawn-join blocks) and create a function definition for each (a “spawn-function”). Replace the spawn-join block with a call to the spawn-function. 2) Spawn-function transformation. Add TCU initialization code and thread emulation constructs to the spawn-function. These constructs include wrapping the body of the spawn-join block with a loop to emulate the threads, and inserting assembly templates.

4. EXPERIMENTAL METHODOLOGY

A behavioral simulator, comparable to SimpleScalar [4], has been developed for an XMT architecture. The fundamental units of execution for the simulated machine are the multiple TCUs, each of which contains a separate execution context. In hardware, an individual TCU basically consists of the fetch and decode stages of a simple pipelined processor. To increase resource utilization and to hide latencies, sets of TCUs are grouped together to form a cluster. The TCUs in a cluster share a common pool of functional units (similar to SMT), as well as memory access and prefix-sum resources. The clusters can be replicated repeatedly on a given chip. Some additional details about the architecture can be found in [5]. Please note, though, that the architecture presented here differs from previous designs in [5] in two important ways. Specifically, the simulated architecture does not have hard-wired thread management, and uses a fundamentally-different parallel banked-memory architecture.

For our experiments, we specify 8 TCUs in each cluster. Each cluster contains 4 integer ALUs, 2 integer multiply/divide units, 2 floating point adders, 2 floating point multiply/divide units, and 2 branch units. All functional unit latencies are set to the SimpleScalar sim-outorder defaults: integer divide, multiply and ALU ops take 20, 3 and 1 cycles respectively, floating point divide, multiply and addition ops take 12, 4 and 2 cycles respectively, and square root takes 24 cycles. Each cluster has a L1 cache of 8 KB. The L2 cache is composed of shared independent banks which total 1 MB. The number of banks is chosen to be twice the number of clusters. The L2 cache latency within each cache bank is 6 cycles and memory latency is 25 cycles. A penalty of 4 cycles is charged each way for every one-way traversal of the interconnection network. Each cluster can issue a single memory request per cycle, but multiple requests may be in-flight at once. For example, a single cluster may have 8 loads from 8 different threads being serviced at once.

Domain	Program	Description	Source	Input Data Set	# Cycles
Scientific computations	jacobi	2D PDE solver kernel		512 x 512	7,739,190
	tomcatv	Mesh generation program	SPEC95	64 x 64	209,250,860
Linear algebra	mmult	Matrix multiplication	Livermore loops	300 x 300	529,615,119
	dot	Inner product	Livermore loops	2 x 64K	2,064,536
Database	dbscan	SQL Select query on a non-indexed attributes relation.	[3]	2100000 nodes	128,975,233
	dbtree	A batch of indexed-tree searches.	MySQL	131072 nodes	3,504,809
Image processing	convolution	Image Convolution	[3]	128 x 128	64,228,266
	perimeter	Compute the total perimeter of a region in a binary image	Olden [24]	32K quad-tree	1,632,686
Sorting algorithms	quicksort	Recursive sort using pivots		16K nodes	7,645,175
	radixsort	Integer sort into buckets		16K nodes	2,112,779
Graph traversal	treeadd	Summation of binary tree nodes	Olden	4096 nodes	5,209,216
	dag	Find maximum path in a DAG.		1024 nodes 131,157 edges	1,140,753

Table 1: Benchmark programs

Configurations are simulated with 1, 4, 16, 64, and 256 TCUs. (The 1 and 4 TCU configurations obviously have fewer than 8 TCUs per cluster.) Keep in mind that these numbers indicate the number of simultaneous execution contexts, and do not imply hardware functionality equivalent to the same number of standard microprocessors.

The highest-end configuration simulated uses 32 clusters. At this point, connectivity to this degree has not been demonstrated for a single-chip system. The interconnection implementation is an important element of a scalable XMT hardware architecture. The simulator used reflects results of VLSI experiments with a specific design, which are discussed in detail elsewhere [22]. For the purposes of this paper, then, the results for the high-end configuration can be considered to be indicative of the potential for the XMT threading model to scale to high degrees of parallelism. This scalability is one of the most important features of the methods presented here.

We conducted our experiments on a variety of realistic applications representing a range of problem domains. Table 1 summarizes the benchmarks we used. The following information is provided for each program: 1) A brief description of the computation that it performs. 2) In some cases we used available source codes to implement the (serial version of the) program. In such cases, the column 'Source' indicates where we obtained these source codes. 3) Input problem size; unless stated otherwise, these are the problem sizes used throughout all the experiments. 4) Total execution time of the serial version of the program in number of cycles (running on 1 TCU).

Note that we have chosen small input sizes in most cases. For the XMT codes, relatively short threads (8-10 machine instructions) are often used; XMT is meant to be extremely fine-grained as the title of this paper suggests. Also bear in mind, when interpreting the performance results in the sections to follow, that only an implementation could establish the actual overall performance of the architecture. Complex issues, such as the potential interference between the various new hardware mechanisms and existing superscalar-processor implementations, have been too hard to evaluate by simulations alone.

5. EVALUATING THE XMT ENVIRONMENT

This section evaluates the efficiency of the XMT environment by examining 1) the overheads that the parallel constructs incur; 2) memory stall behavior, 3) load balance, and 4) scalability of the system. We focus here on general features of our platform, independently of programming considerations. Programming issues are discussed in section 7.

5.1 Overheads

Setting up a parallel region and managing the threads incur an overhead. We can break down this cost to the following different elements: 1) Spawn-Setup: setting up the environment, broadcasting data. 2) TCU-Init: initializing the TCUs context. 3) Thread Overhead: emulating threads on each TCU - obtain a thread ID and verify that it is less than the spawn size. 4) Load Imbalance (Spawn-End): idling at the end of a spawn until all threads complete, then transitioning back to serial mode. To give an idea of the size of these elements in the prototype environment, a single Spawn-Setup is typically 16 instructions and costs 150-300 cycles, a typical TCU-Init is 22 instructions and costs 140-250 cycles, a single Thread Overhead is 3 instructions and costs 15 cycles, and a Spawn-End is 9 static instructions with run-time cost varying considerably depending on load balance. Note that, as will be explained later in this section, the Spawn-Setup and TCU-Init costs were generally insignificant for our benchmarks. We expect these costs could be reduced significantly if necessary.

We examined the costs that the different kinds of overheads incur, and observed several trends. Overheads are

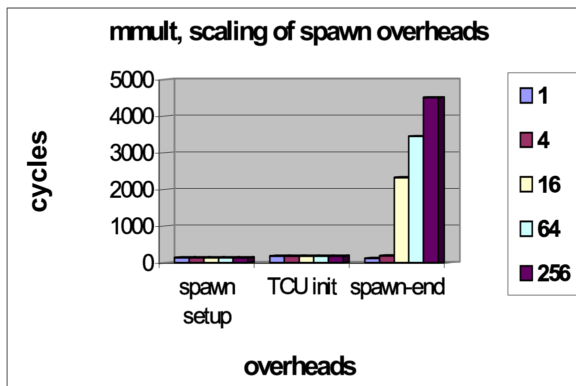

Figure 6: Spawn block overheads.

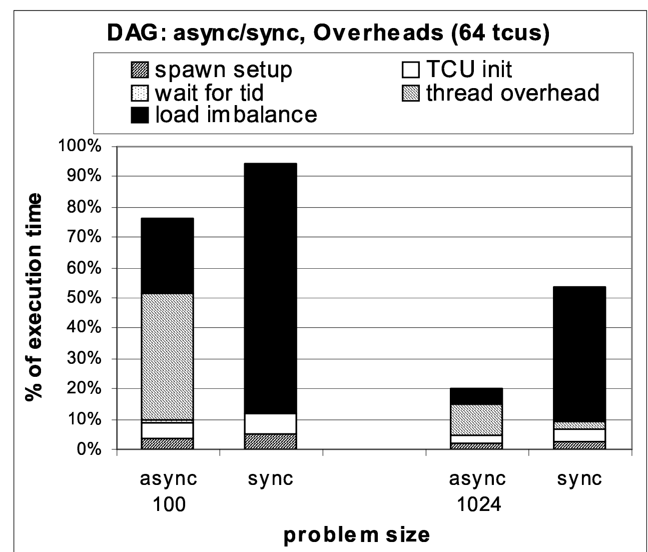

Figure 7: Fork versus synchronous programming

generally very low. This allows XMT to obtain good speedups even for very small problem sizes, and very fine-grained parallelism. Figure 6 reports the spawn-block overhead costs for a matrix multiplication program. We report the number of cycles that a TCU spends on each of three categories (spawn-setup, TCU-init and spawn-end) averaged across all TCUs.

These results, typical of XMT programs, demonstrate that setting up the parallel region is a cheap operation. The Spawn-Setup and TCU-Init overheads are in general negligible, and remain low under increasing problem sizes and increasing number of TCUs. As a result, programs that involve many spawns and joins still perform well. As the number of TCUs increases, the opportunity cost of idle TCUs at the end of the parallel region (Spawn-End) becomes more significant. Note however, that these overheads amount to a negligible percent of the entire execution time of the program (550 million cycles on 1 TCU, and 2.89 million cycles on 256 TCUs). The most dominant overhead is the one charged to thread creation. We therefore concentrate on optimizations that aim to reduce this overhead (section 6).

We also observe that the thread structure of the parallel algorithm greatly affects the overhead distribution. We demonstrate that using two versions of dag, a program that computes maximum length paths in a DAG. A synchronous version uses frequent spawns and joins, while an asynchronous version forks new threads to explore nodes as they are discovered. Figure 7 shows overhead breakdowns for both versions on two different graph

sizes: one graph consisting of 100 nodes and 473 edges, and a larger graph, consisting of 1024 nodes and 131157 edges. The data presented is averaged across all TCUs. As illustrated, the synchronous version pays a heavy price in load imbalance. The forking version is able to adapt to the unpredictable computational demands and avoid these costs. This advantage is evident in the speedups achieved, especially with more TCUs (see Figure 12).

5.2 Memory Stall Behavior

An interesting factor to examine is how memory stall behavior scales with the number of TCUs. We found that the ratio of time spent waiting on memory to time spent on processing was largely constant from 1 to 256 TCUs for most of the programs tested (Figure 9). As an example, Figure 8 shows the breakdown of TCU time between active processing (CPU), memory stalls, and idle-time - idling while waiting for other TCUs to complete emulating the remaining threads (note that time spent waiting on a functional unit is not considered idle-time; idling here reflects load imbalances, typically at the end of the spawn block). The data presented is averaged across all TCUs, as obtained from the simulation of dbtree - a program that performs a batch of indexed-tree searches. As the number of TCUs increases, the memory stall share does not excessively increase.

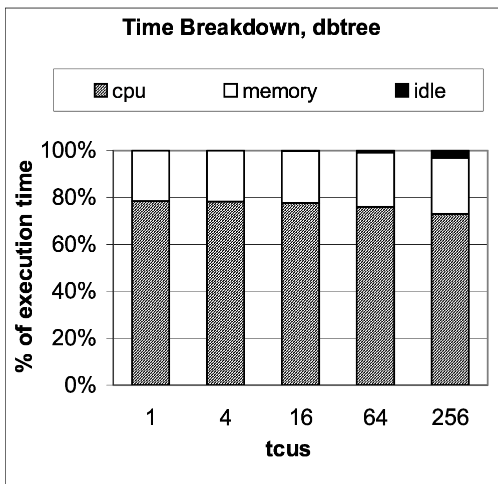

Figure 8: Memory stall behavior, dbtree.

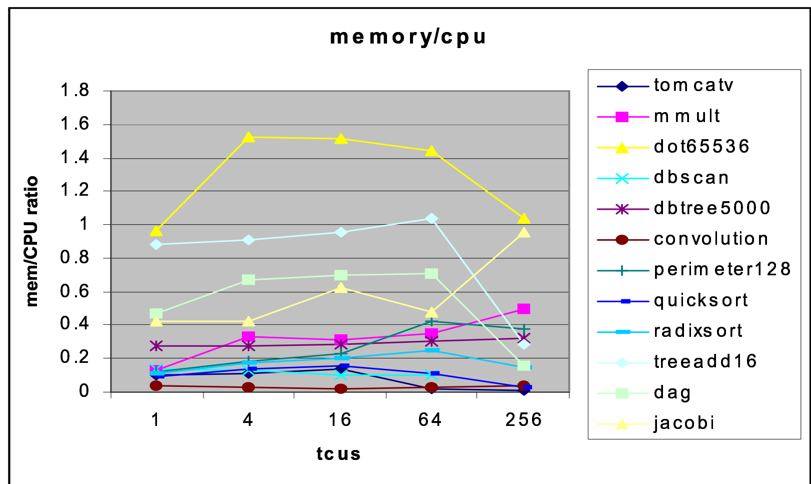

Figure 9 : Memory/CPU ratio

We observe this trend quite consistently across a broad range of benchmarks in Figure 9. This can be attributed to the XMT architecture design. The objective of this design is to provide high bandwidth to satisfy many simultaneous, unpredictable, unlocalized memory accesses. Parallel cache modules paired with a highly pipelined interconnection network provide scalable performance.

5.3 Dynamic Load Balancing

The XMT architecture provides dynamic load balancing; newly created threads are automatically assigned to TCUs without complicated programmer intervention. This dynamic load balancing is particularly useful for handling cases where work partitioning is of an unpredictable nature. Figure 10 reports results for such a program – dag, running on 16 TCUs. It shows the total number of cycles that each of the 16 TCUs spent during the execution of the program. We observe that the disparity between TCU workloads is relatively small considering the irregular nature of the computation, with large disparity between thread lengths (some threads terminate immediately, whereas other loop through all the outgoing edges of the nodes they operate on. As a result, thread lengths vary considerably).

5.4 Scalability

To demonstrate the scalability of XMT we present speedups of XMT programs relative to the best serial version, for applications that are considered to be relatively parallelizable: jacobi (a 2D PDE kernel), tomcatv (the SPEC95 mesh generation program), mmult (matrix multiply) and dot (dot product) from Livermore Loops, image convolution (from [3]), and two database kernels – dbscan (SQL select query on a non-indexed relation [3]) and dbtree (indexed-tree searches, taken from MySQL). These programs feature regular computations that operate on different entries of a data structure independently of one another. This allows a very simple parallelization scheme that involves small extra overhead, where basically a thread is spawned for each loop iteration in the serial version.

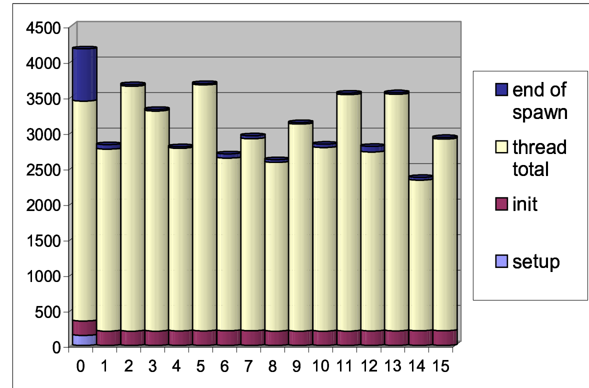

Figure 10: Load balance in dag.

The results shown in Figure 11 demonstrate that XMT programs are able to obtain good speedups, that scale up to much higher levels (256 TCUs) than have been demonstrated before for single-chip systems. Low speedups demonstrated by Tomcatv, are attributed to a problem size that is too small (64 columns), limiting the available parallelism for the scheme used. The slowdown rate of our simulator made it difficult to experiment with bigger problem sizes; see discussion on this issue in section 8.

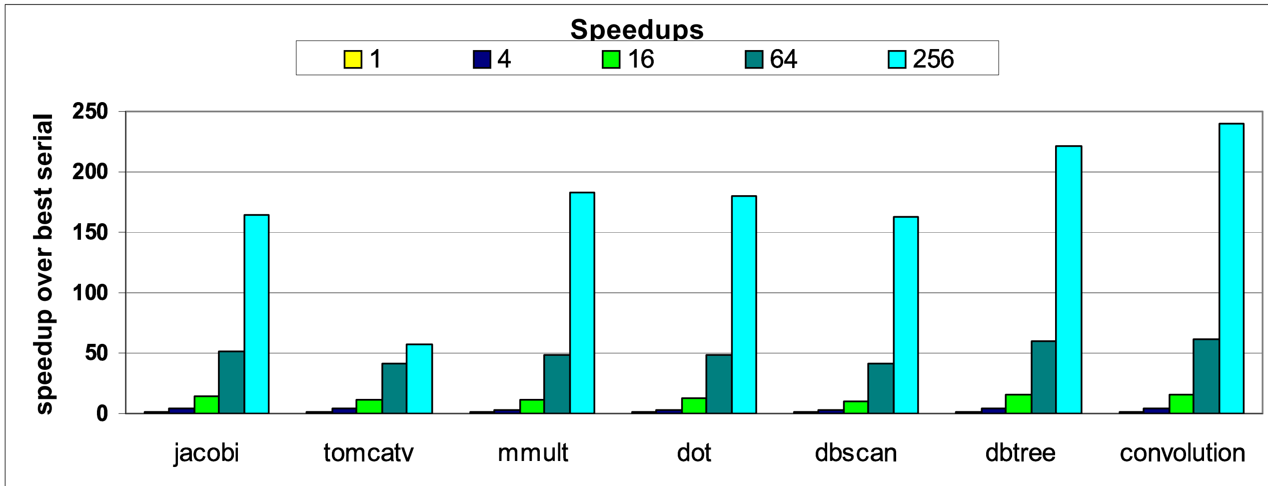

Figure 11: Scaling of speedups

6. COMPILER OPTIMIZATIONS

The XMT programming methodology encourages the programmer to express any parallelism, regardless of how fine-grained it may be. The low overheads involved in emulating the threads allow this fine-grained parallelism to be competitive. However, despite the efficient implementation, extremely fine-grained programs can benefit from coarsening, as it decreases the thread count, consequently reducing the overall thread overhead. Furthermore, thread coarsening may allow exploiting spatial locality, and reducing duplicate work; however, these opportunities occur only in regular codes, where it is also easy to automatically detect and optimize. By grouping consecutive threads, clustering exploits spatial locality, and allows the programmer to ignore granularity and task assignment considerations, which are otherwise relevant.

The XMT compiler detects cases where the length of the thread is sufficiently small (such that the thread overhead constitutes a significant enough portion of the thread). This parameter is evaluated at compile time using SUIF's static performance estimation utility. The compiler then automatically transforms these spawn-blocks such that fewer but longer threads are used. Furthermore, our optimization takes load balance considerations into account by reserving unclustered-threads at the tail of the spawn. Tuning this value can reduce the load imbalance cost, however at the expense of a small increase in thread overhead. Therefore, two sets of threads are emulated: the first set consists of the coarse clustered-threads, and the second is the set of the remaining fine-grained unclustered-threads.

Rather than splitting the computation to two separate spawn-blocks (one for each set of threads), or introducing a conditional control to determine between the two, we use the following scheme: we create a single spawn block, which contains two separate thread emulation loops – one for the clustered-threads, and one for the unclustered-threads. Thus, given a fine-grained spawn-block of ***n*** threads, our compiler approach results in the following execution scheme. The execution starts with a coarse grained version, and then, after ***m*** out of the original ***n*** threads have been emulated through the coarse-grained version, the execution switches to a finer grained version, to finish all ***n*** threads. Once the XMT execution crosses a threshold, the SPMD code becomes the fine-grained version. So, any TCU that picks up virtual threads from that point on, executes directly the fine-grained version, rather than having each TCU refigure that we are in the tail case and only then jump to the code for the tail case.

Figure 12 presents the overall improvement obtained by clustering, as percentage of the original (fine-grained) execution time. We report results for LU, dbscan and jacobi. Two major factors contribute to performance improvement: 1) Exploiting spatial locality: clustering reduces overall memory stall time by 20%, 64% and 14% for LU, jacobi and dbscan, respectively. 2) Eliminating duplicate work: clustering can potentially incur less duplicate work by scheduling operations to have per-cluster cost rather than per-thread cost. Thus, the overall active processing time can be reduced. In LU and jacobi this is indeed the case; clustering reduces CPU time by 13% for LU, and by 21% for jacobi. However in dbscan clustering actually increases CPU time by 18%, due to the overheads that the clustering transformation introduces. The prefix-sum operation that the threads perform in dbscan inhibits the conservative compilation scheme from extracting computation out of the cluster loop. Consequently, clustering does not improve performance for dbscan.

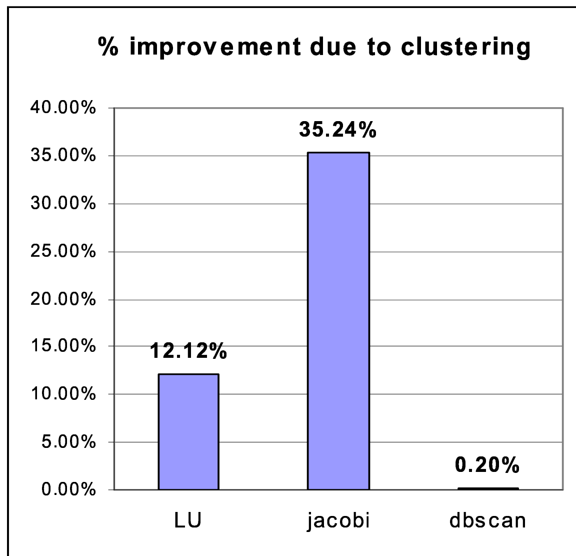

Figure 12: Impact of clustering.

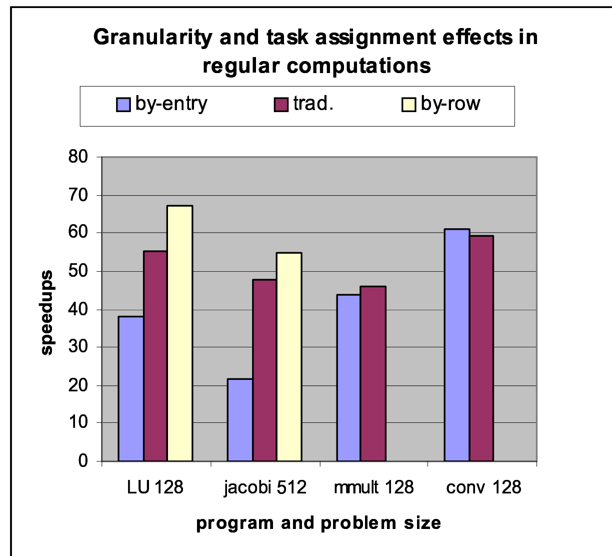

Figure 13: Granularity and task assignment.

7. EVALUATING THE XMT PROGRAMMING MODEL

While traditional shared memory programming consists of assigning as coarse-grained chunks of work to processes as possible, using locks and barriers for synchronization, XMT programs feature 1) No task assignment, 2) Fine-grained parallelism, and 3) No busy-wait. It remains to be examined to what extent this programming methodology is able to excel in an on-chip environment. In this section we discuss how these features are manifested in the programming and performance of two different types of computation domains - regular computations, and irregular dynamic computations. (Other types of computations, such as divide-and-conquer and sorting algorithms, are discussed in [20]).

7.1 Regular Computations

This family of algorithms encompasses any computation that takes the form of looping through a sequence of independent operations, all consisting of the same amount of work, typically applied to different elements of the data structure. The access pattern characteristic of these algorithms is therefore very regular and predictable. Many scientific applications rely on this type of algorithm, including PDE solvers, mesh generators and linear algebra codes.

The independence between the elements of computations in this type of program, naturally allows parallelism present across all elements to be exploited. In XMT, this is translated to a simple parallelization scheme, where a thread is spawned for each loop iteration. Traditional parallelization differs from XMT in having to first partition the domain to (coarse-grained) units of work, and devise a scheme that assigns these blocks of work to the processing units. For the simple family of algorithms discussed here, domain partitioning and task assignment take the form of scheduling loop iterations across the processors. For array based applications, locality considerations determine the scheduling scheme, whether block-wise, cyclic, or blocked-cyclic.

The different programming style directly affects the granularity of the parallelism that is expressed. The XMT programs, taking advantage of the parallelism present to the largest degree possible often result in very fine-grained computations. For the programs we examine in this category, the typical length of a thread is between 3 to 6 source lines. Traditional programming neglects the per-entry parallelism and distributes the work in a coarse-grained fashion.

To evaluate the effects of granularity and task assignment on performance we use LU, mmult, convolution and jacobi. We compare the following versions for each (Figure 13): 1) "by-entry"/"by-row", where a thread is spawned directly for each entry/row. For these regular computations, the coarser by-row versions outperform the fine by-entry versions, demonstrating the need for thread coarsening for this type of applications. (The by-entry version of mmult and conv is already relatively coarse-grained, which is why these programs are missing results for a by-row version). 2) "trad", where traditional style programming with respect to task assignment is used. Here, a thread is spawned for a group of rows/entries. For mmult and convolution, both versions achieve similar speedups. The traditional mmult is able to amortize some duplicate work, while the "by-entry" versions of convolution LU and jacobi take the lead by avoiding some task assignment overhead.

To conclude the discussion on regular computations, we discuss general reduction computations, as they provide a classic example for a case where XMT employs an entirely different algorithmic approach than the traditional one. We present two algorithms. Both utilize a binary tree structure [30]. The first, propagates values up the tree in a synchronous fashion, using a spawn and join for each layer of the tree. Each spawn block consists of a thread for each node in the parent layer that applies the reduction operation on the two children of that node. This is repeated for the next level, until the root of the tree is reached.

The second algorithm propagates values in an asynchronous fashion, involving only one spawn-join block within which threads advance without busy-waiting. This solution requires maintaining an additional data structure, a *gatekeeper*, to ensure that the reduction operation is applied on a node is after the value of both its children is ready. The computation proceeds as follows: after the value of a node has been computed, the thread performs a

prefix-sum relative to the gatekeeper of that node's parent. The result of the prefix-sum indicates if it was the first thread to do so, in which case it terminates. Otherwise it was the second; it proceeds to calculate the value for the parent, and continue.

Our experiments show that the asynchronous algorithm is outperformed by the synchronous one [20] due to the amount of storage and extra work that it involves. Asynchronous algorithms are more useful for irregular computations, as we show next.

7.2 Irregular, Dynamic Computations

The algorithms we discuss here are characterized by highly irregular and unpredictable access patterns. Specifically, we consider computations that begin with a limited amount of work and discover additional work as they proceed. The newly discovered work typically requires splitting the processing to subtasks and combining the contribution of each as they complete. For example, consider rendering techniques that rely on ray tracing. There, primary rays fired from a viewpoint encounter objects, and are reflected from and refracted through the objects, spawning new rays. The same operations are performed recursively on the new rays, all contributing to the intensity and color of the same pixel. This pattern of computation is also present in applications that rely on breadth-first-search style algorithms.

The irregular nature of this type of algorithm makes them good candidates for a less synchronous parallelization scheme. This is the programming approach that XMT employs, using dynamic forking as new units of work are discovered (be it rays, graph nodes/edges, etc.). Traditional parallel programming, with its coarse-grained work decomposition and task assignment, can't employ a simple static scheme as it would lead to severe load imbalances. Traditional parallelization techniques therefore require explicitly balancing processor workloads, either by intelligent partitioning or dynamic work stealing (such as the SPLASH-2 implementations for volume rendering, radiosity and a ray tracing [25]).

XMT implementations take one of the following approaches: 1) A synchronous approach performs a spawn at each stage of the computation. The first spawn block creates a thread for each preliminary unit of work (a primary ray, a node with in-degree 0, etc.). After a join, threads are spawned for newly discovered units of work. The process is repeated until all the work units (rays/nodes) are processed ("sync"). 2) Less synchronous approaches fork a thread for every new piece of work as it is discovered. Typically, only one spawn block is used. The granularity of the work units for which a thread is forked, varies between the different approaches. For example, in a breadth-first search, a thread can be spawned for every node ("async-node") or alternatively for every out-going edge ("async-edge") for a more fine-grained, less-synchronous algorithm.

Traditional style versions are based on the "sync" version (layer by layer topological sort), adding the necessary task decomposition. In either approach taken, certain data accesses in this computation require synchronization. Where traditional programming style uses locks ("trad-lock"), XMT programs use the prefix-sum instead ("trad).

Figure 14 demonstrates programming tradeoffs for irregular applications using the dag program. We show results for two different graph sizes: a relatively small graph, consisting of 100 nodes and 473 edges, and a larger graph, consisting of 1024 nodes and 131157 edges. The phenomena we discuss below can be observed in both graph sizes, but are evident to a greater extent for smaller graph:

1. XMT programs can be much simpler, as domain distribution and task assignment are not needed. This is particularly important for dag, where devising a scheme that achieves good load balance may be very challenging, and requires substantial effort.
2. Reduced synchrony is often achieved at the expense of some additional programming effort. However, asynchronous programs should excel by enabling parallelism as soon as it is discovered (illustrated by the superiority of "async-edge/node" over "sync").
3. Traditional programming using locks and barriers can be supported in XMT; Programs can be implemented in XMT in the same way they are implemented under traditional parallel programming

models. Furthermore, the traditional synchronization mechanisms can be replaced with more efficient and scalable XMT utilities, such as prefix-sum. (illustrated by the superiority of "trad" over "trad-lock").

4. The impact of fine-granularity on programming is more significant in irregular programs that have traditionally resisted parallel solutions due to their unpredictable access patterns. Algorithms for such applications can often take advantage of fine-grained parallelism (illustrated by the superiority of "async-edge" over "async-node").

Figure 15 shows results for other programs that have resisted parallel solutions due to dynamic, irregular access patterns of computation.

Radix is another example of a program that is known to be very problematic with regard to obtaining speedups by parallelization. Similarly to dag, it requires a lot of all-to-all communication. SPLASH-2 reports very low speedups on their shared memory multiprocessor [36]. To maximize scalability, our implementation of radix uses fine-grained parallelism wherever possible. This algorithm is much more work-intensive than the serial version, and hence does not achieve speedups for less than 16 TCUs.

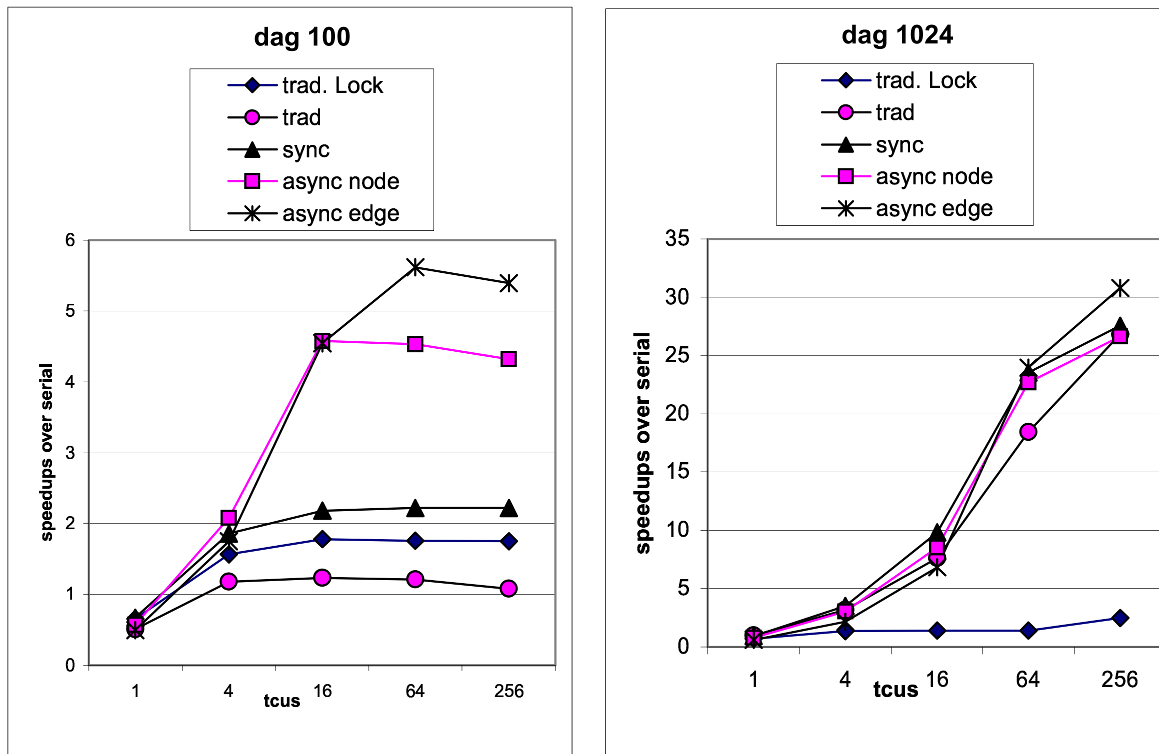

Figure 14: Different programming styles for dag computation

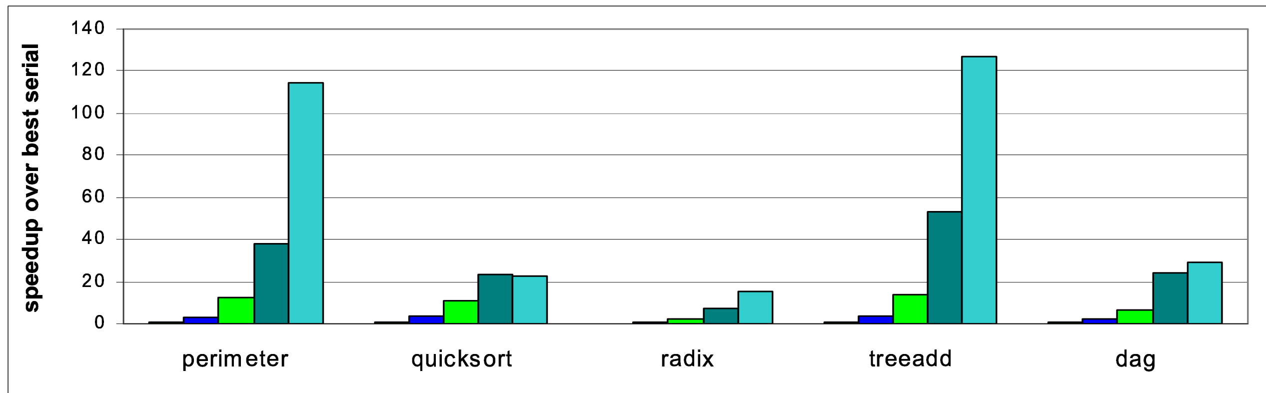

Figure 15: Speedups for irregular applications.

Perimeter and treeadd, both involve traversing a tree from the root down, forking threads along the way, until the leaves are reached. Then, the threads work their way up the tree performing the fine-grained computation.

In quicksort we use a hybrid algorithm, where we start in a synchronous, extremely fine-grained fashion until sufficient partitions have been created. We then switch to handling all the partitions in parallel, in a divide-and-conquer manner. The first part involves a lot of spawning and joining, whereas the second part is a single spawn that forks threads as new partitions are created.

These results demonstrate that XMT is able to obtain speedups for programs where traditional programming approaches have achieved very limited success. XMT achieves good speedups for these applications, at problem sizes well below those for which traditional parallel machines begin to become competitive with serial machines. One can reference the comparison of XMT to the Intel Paragon for the list-ranking problem in [9] for another illustration of this point. XMT hardware enables efficient implementation of PRAM-style algorithms. In turn, these parallel algorithms allow a broad class of applications to yield to Amdahl's law.

8. DISCUSSION

8.1 Performance Issues

Accurate simulation of a complex system is known to be very computationally costly. For example, many human-years of effort were put into making Simplescalar efficient using techniques such as predecoding of instructions, but still it has been rated at 1300 slowdown [4]. The XMT simulator evolved from an early version which emphasized functionality and transparency rather than simulation performance. Future work on the XMT simulator, which is modeled after Simplescalar, will try to offer similar improvements to the performance issue. One solution would be to reimplement the simulator using the Simplescalar toolkit, which is designed for easy extension.

Another, more philosophical, problem seems to stand in the way of obtaining good performance for simulating XMT on an existing (serial, or parallel) machine. For serial machines, the parallelism of XMT implies a number of concurrent operations, which is much larger than in a contemporary serial machine. For parallel machines, the much finer grained parallelism of XMT does not map well into the coarser grained existing parallel machines. More generally, we believe that an XMT architecture will provide a good "universal parallel machine" - a machine which could simulate efficiently other parallel architectures; however, one of the problems which motivated the introduction of XMT is the understanding that currently, a good universal parallel machine does not exist. In other words, the fact that the simulation is time consuming is consistent with the case we make for why an XMT machine ought to be built.

8.2 On the Challenge of Supporting a "Virtual-PRAM"

Under the title "Potential Breakthroughs," [8] suggest to "somehow design machines in a cost-effective way that makes it much less important for a programmer to worry about the data locality and communication; that is, to truly design a machine that can look to the programmer like a PRAM". Overall, the question of whether designing such a machine is possible is one of the defining open problems of the SPAA community.

The main element specific to XMT is its support of a PRAM-like programming model. Quite a few papers, including [10], recognized that the PRAM ("the most widely used parallel model") allows the development of "surprisingly fast parallel algorithms" and provides an unmatched intellectual asset.

During the 1990s, the statement "the PRAM is unrealistic" has become almost a mantra among academic and other computer scientists. The influential paper [7] represents one of many publications that made this statement. If it reaches successful implementation, the current paper will lead towards establishing that a "virtual-PRAM platform" is realistic, and that the PRAM is a good model of computation that possibly may need to be studied by every computer scientist.

9. CONCLUSION

XMT is a computation paradigm that spans from parallel algorithms, through their programming, to the hardware design. The compilation techniques described here offer competitive performance for XMT programs. Results show the XMT architecture generally succeeds in providing low-overhead parallel threads and uniform access times on-chip. However, compiler optimizations to cluster (coarsen) threads are still needed for very fine-grained threads. The prefix-sum instruction provides more scalable synchronization than traditional locks, and the flexible programming style also encourages the development of new algorithms to take advantage of properties of on-chip parallelism. The compilation scheme, combined with the efficient architecture and simple programming model, allow XMT to realize its uncompromising approach to parallelism. Performance gains are achieved for a wider range of problem sizes, granularities, and types of algorithms and computations.

10. ACKNOWLEDGEMENT

Help by Yosi Ben-Asher and the hosting of D. Naishlos by the IBM Haifa Research Lab in January 2000 are gratefully acknowledged.

11. REFERENCES

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

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

[3] A. Acharya, M. Uysal, J. Saltz. Active Disks: Programming Model, Algorithms, and Evaluation. Proc. ASPLOS'98, October 1998.

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

[5] E. Berkovich, J. Nuzman, M. Franklin, B. Jacob, U. Vishkin, "XMT-M: A scalable decentralized processor," UMIACS TR 99-55, September 1999.

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

[7] D. Culler, R. Karp, D. Patterson, A. Sahay, K.E. Schauser, E. Santos, R. Subramonian, T. von Eicken, "LogP: Towards a Realistic Model of Parallel Computation," Proc. Principles and Practice of Parallel Programming, 1-12, 1993.

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

[9] 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. To appear in ACM Journal of Experimental Algorithmics.

[10] 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.

[11] M. Frigo, C. Leiserson, K. Randall, "The Implementation of the Cilk-5 Multi-threaded Language," Proc. of the 1998 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI), 1998.

[12] L. Hammond, B. a. Hubbert, M. Siu, M.k. Prabhu, M. Chen, K. Olukotun, “The Stanford Hydra CMP,” IEEE MICRO magazine, March-April 2000, pp. 71-84.

[13] L. Hammond, B. Nayfeh, and K. Olukotun, ”A Single-Chip Multiprocessor,” IEEE Computer, Vol. 30, pp. 79-85, September 1997.

[14] J.L. Hennessy and D.A. Patterson. "Computer Architecture A Quantitative Approach," Second Edition. Morgan-Kaufmann, San Francisco, 1996.

[15] J.L. Hennessy and D.A. Patterson. "Computer Architecture A Quantitative Approach," Third Edition. Morgan-Kaufmann, San Francisco, 2003.

[16] M. S. Lam, R. P. Wilson, “Limits of Control Flow on Parallelism,” Proceeding of the 19th International Symposium on Computer Architecture (ISCA), May 1992, pages 19-21.

[17] K. Mai, T. Paaske, N. Jayasena, R. Ho, W.J. Dally, M. Horowitz, “Smart Memories: A Modular Reconfigurable Architecture,” In Proc. of the 12th ACM Symposium on Parallel Algorithms and Architectures (SPAA), 2000.

[18] M. Mano, “Digital Logic and Computer Design,” Prentice Hall, 1979.

[19] D. Naishlos, J. Nuzman, C.-W. Tseng, and U. Vishkin, "Evaluating Multi-threading in the Prototype XMT Environment," In Proc. 4th Workshop on Multi-Threaded Execution, Architecture and Compilation (MTEAC2000), December 2000. Best Paper Award.

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

[21] J. Nuzman and U. Vishkin, “Circuit architecture for reduced-synchrony on-chip interconnect,” U.S. Provisional Patent Application 60/0297,248, June 2001.

[22] J. Nuzman, Masters thesis, University of Maryland, Department of Electrical and Computer Engineering. In preparation.

[23] V. Ramachandran, B. Grayson, M. Dahlin, “Emulations Between QSM, BSP and LogP: A Framework for General-Purpose Parallel Algorithm Design,” In Proc. of 1999 ACM-SIAM Symp. on Discrete Algorithms (SODA’99).

[24] A. Rogers, M. Carlisle, J. Reppy, L. Hendren, “Supporting Dynamic Data Structures on Distributed-Memory Machines,” ACM Transactions on Programming Languages and Systems, 17(2) pp. 223-263, March 1995.

[25] J. P. Singh, A. Gupta, M. Levoy, “Parallel Visualization Algorithms: Performance and Architectural Implications,” IEEE Computer 27(7):45-55, July 1994.

[26] I. Sutherland, “Micropipelines,” Communications of the ACM, June 1989, Turing Award Lecture.

[27] D. M. Tullsen, S. J. Eggers, H. M. Levy, "Simultaneous Multithreading: Maximizing On-Chip Parallelism," In Proc. of the 22nd Annual International Symposium on Computer Architecture, Santa Margherita Ligure, Italy, June 1995.

[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.

[30] U. Vishkin, "A No-Busy-Wait Balanced Tree Parallel Algorithmic Paradigm," In Proc. of the 12th ACM Symposium on Parallel Algorithms and Architectures (SPAA), 2000.

[31] U. Vishkin, "Spawn-join instruction set architecture for providing explicit multithreading (XMT)," U.S. Patent 6,463,527, October 8, 2002.

[32] U. Vishkin, "Prefix sums and an application thereof," U.S. Patent 6,542,918, April 1, 2003.

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

[34] R. Wilson et al, "SUIF: An Infrastructure for Research on Parallelizing and Optimizing Compilers," ACM SIGPLAN Notices, v. 29, n. 12, pp. 31-37, December 1994.

[35] 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, pp. 86-93, September 1997.

[36] S. Woo, M. Ohara, E. Torrie, J. Singh, A. Gupta, "The SPLASH-2 Programs: Characterization and Methodological Considerations," Proc. of the 22nd Annual International Symposium on computer Architecture, pp. 24-36, June 1999.

12. APPENDIX A – HARDWARE IMPLEMENTATION OF PREFIX-SUM

A hardware implementation for small-integer prefix-sum, as presented in [28] and also in [32], follows. Generally, one would expect each (single) prefix-sum to need an adder. see e.g. [2]. In this section we show that hardware cost can be drastically reduced, in one case where the input values for a multiple prefix-sum are limited.

We say that a multiple prefix-sum instruction

```
PS R_0 R_1
PS R_0 R_2
…
PS R_0 R_k
```

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

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

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

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

$$PS\ R_0\ R_1$$
$$PS\ R_0\ R_2$$
$$\ldots$$
$$PS\ R_0\ R_h$$

into

$$z_0 = R_0$$
$$z_1 = R_0 + R_1$$
$$z_2 = R_0 + R_1 + R_2$$
$$\ldots$$
$$z_h = R_0 + R_1 + R_2 + \ldots + R_h$$

1. Compute the base-zero prefix-sum, using dedicated hardware. By base-zero we mean that the input values of registers R_1 through R_h are taken as they are, but the base register R_0 is taken as 0. Store these prefix-sums in an intermediate array V of h elements $v_1, v_2 \ldots v_h$.
2. Compute the following "suffix sum", using dedicated hardware into an intermediate array U of h elements $u_1, u_2 \ldots u_{h-1}$.

$$u_{h-1} = R_h$$
$$u_{h-2} = R_{h-1} + R_h$$
$$\ldots$$
$$u_1 = R_2 + \ldots + R_h$$

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

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

4. Find index i as follows: Apply exclusive-or bitwise to the binary representation of R_0 and z_h. i is the index of the most significant bit for which the exclusive-or yields one.
5. Derive w.
6. Get the values of the rest of the output which are smaller than w by adding the low-index elements of the intermediate array V to R_0.
7. Get the values of the rest of the output which are equal to or greater than w by subtracting the high-index elements of the intermediate array U from z_h.

Because h is small, and all the elements in R_1 through R_h are small, the additions and subtractions of the last two steps involve only short carries. They can be implemented using short look-ahead carry generators (for look-ahead carry generators, see, e.g., [18]).

13. APPENDIX B - GLOSSARY OF NEW PROGRAMMING CONSTRUCTS

fspawn [C] Spawn of virtual threads with the ability to fork additional threads

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

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

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

pinc [assembly] Parallel prefix-sum to register, with increment value of 1

```
pinc     PRi, $r
```

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

pread [assembly] Parallel read of a PR register (parallel prefix-sum with value 0)

```
pread    PRi, $r
```

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

ps [C] Prefix-sum function (atomic fetch-and-add)

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

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

psalloc [assembly] Allocate a value from a PR register, without updating the register

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

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

pscommit [assembly] See *psalloc*.

pset [assembly] Initialize a parallel register

```
pset     PRi, $r
```

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

psm [assembly] Prefix-sum to memory, with an arbitrary increment value

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

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

spawn [C] Spawn of virtual threads

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

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

spawn [assembly] Spawn operation, all TCUs are interrupted and execute at this `pc`

```
spawn
```

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

suspend [assembly] Suspend TCU until PR register reaches threshold

```
suspend  PRi, $r
```

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

xfork [C] Request an additional thread be created

```
xfork ( );
```

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

xfork_and_init [C] Request an additional thread be created, and provide initial data

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

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