Evaluating Multi-threading in the Prototype XMT Environment

Dorit Naishlos[1] *, 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
Tel: 301-405-8010, Fax: 301-405-6707
{dorit, jnuzman, tseng, vishkin}@cs.umd.edu

Abstract

XMT is a multi-threaded programming model designed to exploit explicit specification of parallel threads. Its main features are a simple thread execution model and an efficient prefix-sum instruction for synchronizing shared data accesses. This paper presents and evaluates the performance of multi-threading in the XMT programming environment. A prototype XMT compiler converts parallel regions into procedure calls, which are then executed efficiently in XMT hardware. An architecture simulator similar to SimpleScalar is used to evaluate the performance of the XMT system for twelve benchmark codes. 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.

Keywords

Parallel programming, compilers, processor architectures.

1. Introduction

Conditional branches, variable memory access latencies, and other barriers to instruction-level parallelism prevent computers from fully exploiting the large number of transistors available in modern processors. XMT, an explicit multi-threading computation framework, attempts to overcome these obstacles by providing efficient hardware support for fine-grained parallel programs.

The basic premise behind XMT is that instead of forcing the hardware to find instruction-level parallelism at run-time, the instruction set architecture should provide programmers (or the compiler) with the ability to explicitly specify parallelism when it is available. In addition, the XMT architecture attempts to provide more uniform memory access latencies, taking advantage of faster on-chip communication times. 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.

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 [VDB+98] [DV99]. In this paper, we evaluate XMT for the first time as a complete environment, including compiler and hardware simulator.

The main contributions of this paper are as follows:

- We present details of a prototype implementation of XMT, including compiler and code shape.
- We experimentally validate the efficiency of XMT using a much larger number of benchmark codes than before.
- We show a number of compiler optimizations for improving the performance of XMT programs.

We begin in Section 2 by reviewing the XMT multi-threaded programming model [VDB+98]. The model is further enhanced with updates to allow nested forking

* Supported by NSF grant 9820955

[Vishkin00]. Section 3 reviews the XMT architecture, especially features which support multi-threading. Section 4 presents the prototype XMT compiler and code generation model. Section 5 describes our XMT evaluation environment, including a behavioral simulator. Section 6 evaluates the efficiency of the XMT multi-threaded system implementation. To demonstrate the competitiveness of fine-grained parallel programs, we compare speedups versus efficient serial programs. Section 7 presents a comparison with related work. Section 8 concludes.

2. XMT programming model

The XMT programming model has a number of key features:

- Explicit spawn-join parallel regions
- Threads run to completion (do not busy-wait)
- Shared accesses synchronized with prefix-sum instruction
- Uniform accesses to shared memory

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. At run-time, different threads may have different lengths, based on control flow decisions made at run time. The arbitrary CRCW aspect dictates that concurrent writes to the same memory location result in an arbitrary one committing. No assumption can 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. An 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 is available in the literature.

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 R gets 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 value in its local storage R. The parallel prefix-sum command can be used for implementing efficient and scalable (i) load balancing (parallel implementation of queue/stack), and (ii) inter-thread synchronization.

The XMT high-level language is an extension of standard C. A parallel region is delineated by spawn and join statements. 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 (other TIDs will follow consecutively). Prefix-sum takes the form of the ps function. It adds its second argument (the increment) to the first argument (the base) and returns the original value of the base.

Consider the following small example of an XMT 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 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 [VDB+98] 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, [Vishkin00] 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. A parallel region ends after all the threads, whether specified by the original spawn or initiated by subsequent forks, have been emulated. The fork extension allows the programmer to approach many problems in a more asynchronous and dynamic manner.

The fork is implemented by using a parallel prefix-sum to increment the number of virtual threads to be executed. The caller can use the result of the prefix-sum to set-up initialization for the new thread. In XMT C, *fspawn* is used when forking may be necessary, and *xfork* performs the fork operation.

3. XMT architecture

The XMT programming model allows the programmer to specify an arbitrary degree of parallelism in a code. Clearly, real hardware has finite execution resources so all

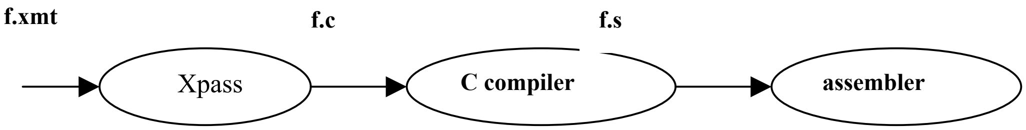

Figure 1: XMT compilation system

threads can't run 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 that ID. All TCUs repeat the process until all the virtual threads have been completed.

This functionality is enabled by support at the instruction set level. With our architecture, all TCUs independently execute a serial program. Each accepts the standard MIPS instructions, and possesses a standard set of MIPS registers locally. The expanded ISA includes a set of specialized global registers, called prefix-sum registers (PR), and a few additional instructions

Four new instructions are used for thread management. A spawn instruction interrupts all TCUs and broadcasts a new PC at which all TCUs will start. The next three instructions operate on the PR registers: pinc performs a parallel prefix-sum with value 1, pread performs a parallel read (prefix-sum with value 0) of a 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, is not performed in parallel (ie. concurrent psm's will be queued.) This is equivalent to a fetch-and-increment.

Additional instructions exist to support the nested forking mechanism. A new thread to be forked likely 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 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.

4. XMT compiler

This section describes how the XMT compiler generates a C source from an XMT program. Parallel execution in the XMT architecture requires handling the following issues:

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 one (i.e. - less than the spawn size).
3. Transition back to serial mode: detect when all threads have terminated, and resume serial mode.

In previous presentations, 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, the front end (Xpass) and the back end (gcc). Figure 1 presents the XMT compilation process. The front end (Xpass) is a source-to-source translator based on SUIF [Wilson94]. This phase converts the XMT code with its parallel constructs into regular C code with specialized assembly templates for run-time threading support.

The general scheme used by Xpass is based on transforming parallel codes into parallel procedures. The compiler transforms the parallel region (the code in the spawn-join block) into the body of the procedure. When the procedure is called, the processing units are awakened, and each starts to execute the procedure body, which emulates the threads on each TCU. Figure 2 presents a high level example of the transformations performed by our compiler. Producing this structure involves two tasks:

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

Original XMT-C program

```
main() {

        spawn(num_threads, offset);
        {
                int TID;

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

}
```

Transformed to

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

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

                **THREAD-CODE**

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

Figure 2: XMT code shape

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.

Xpass is preceded by a number of SUIF passes, and may be followed by a number of future XMT optimizations passes. The back end 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.

5. XMT evaluation environment

A behavioral simulator, comparable to SimpleScalar [BA97], 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, as well as memory access and prefix-sum resources. The clusters can be replicated repeatedly on a given chip. More details about the simulated architecture is described elsewhere [BNF+99]. Unlike previous designs, the simulated architecture does not have hard-wired thread management, and uses a banked memory rather than a monolithic memory.

For our experiments, we specify 8 TCUs in each cluster. Each cluster contains 4 integer ALUs, 2 integer multiply/divide units, 2 floating point ALUs, 2 floating point multiply/divide units, and 2 branch units. All functional unit latencies are set to the SimpleScalar sim-outorder defaults. Each cluster has a L1 cache of 8 KB, and a shared, banked L2 cache of 1 MB. The number of banks is chosen to be twice the number of clusters. A penalty of 4 cycles is charged each way for intercluster communication.

Configurations are simulated with 1, 4, 16, 64, and 256 TCUs. (The 1 and 4 TCU configurations obviously have less 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, but the details of these experiments are beyond the scope of this document. 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.

For our evaluation, we used a set of twelve benchmark codes taken from a variety of application areas. Their characteristics are shown in Table 1.

6. XMT evaluation

This section evaluates the efficiency of our implementation by examining 1) the speedups that we obtain relative to serial programs; 2) the overheads that the parallel constructs incur; 3) the load balance and memory behavior. We demonstrate the effect of different

Domain	Program	Description	Source	Input Data Set	# of CPU Cycles
Scientific Computations	jacobi	2D PDE kernel		512 x 512	7739190
	tomcatv	Mesh generation program	SPEC95	64 x 64	209250860
Linear Algebra	mmult	Matrix multiplication	Livermore loops	300 x 300	529615119
	Dot	Inner product	Livermore loops	2 x 64K	2064536
Database	Dbscan	SQL Select query on a non-indexed attributes relation.	[AUS98]	2100000 nodes	128975233
	Dbtree	A batch of indexed-tree searches.	MySQL	131072 nodes	3504809
Image processing	Convolution	Image Convolution	[AUS98]	128 x 128	64228266
	Perimeter	Compute the total perimeter of a region in a binary image represented by a quadtree	Olden	128 x 128 pixel quadtree	1632686
Sorting algorithms	Quicksort	Recursive sort using pivots		16K nodes	7645175
	Radixsort	Integer sort into buckets		16K nodes	2112779
Misc	Treeadd	Summation of binary tree nodes	Olden	64K nodes	5209216
	DAG	Find maximum path in a DAG.		1024 nodes, 131157 edges	1140753

Table 1: Benchmark Programs

programming styles, scalability with the number of TCUs, and the effect of certain compiler optimizations on these parameters.

6.1 Speedups

In this section we examine the performance of XMT on a variety of realistic applications. The speedups that XMT programs obtain over serial programs demonstrate the applicability of the XMT framework and its efficiency across a variety of problem domains. Table 1 summarizes the benchmarks we use.

The speedups we present in Figure 3 are relative to the best serial version for each application. The upper graph displays applications that are considered to be relatively parallelizable. The lower graph shows results for programs that have resisted parallel solutions due to the dynamic, irregular access patterns of the computation.

Radix and dag are both examples of programs that are known to be very problematic with regard to obtaining speedups by parallelization. Both require a lot of all-to-all communication under other programming models. SPLASH-2 reports very low speedups on their shared memory multiprocessor, "due to a parallel prefix computation in each phase that can not be completely parallelized" [WOT+95]. To maximize scalability, our 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.

The parallelism present in the dag computation is limited by the input graph. The unpredictable nature of the computation lends itself to a solution involving the nested fork operation.

Perimeter and treeadd both involve traversing a tree from the root down, forking threads a long 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, fine-grained fashion until sufficient partitions have been created. We then switch to handling all the partitions in parallel. The first part involves a lot of spawning and joining, where as the second part is a single spawn that forks threads as new partitions are created. Results show speedups topping out at 64 TCUs.

The remaining programs allow a very simple parallelization scheme: mmult, jacobi and convolution all update array entries independently of one another, and so the parallelism is straight forward and involves minimal extra overhead. This is also true for dbscan where each thread examines a single entry in the relation, and in dbtree, where each thread handles a single transaction. Note that these programs directly spawn a thread for each unit of work, while traditional parallel programming uses a more coarse-grained task distribution.

Tomcatv involves updating matrix columns independently, where each step of the computation involves 6 spawn-join blocks. The problem size (64 columns) limits the available parallelism for this scheme. Dot uses an array reduction scheme to attack the dot product problem.

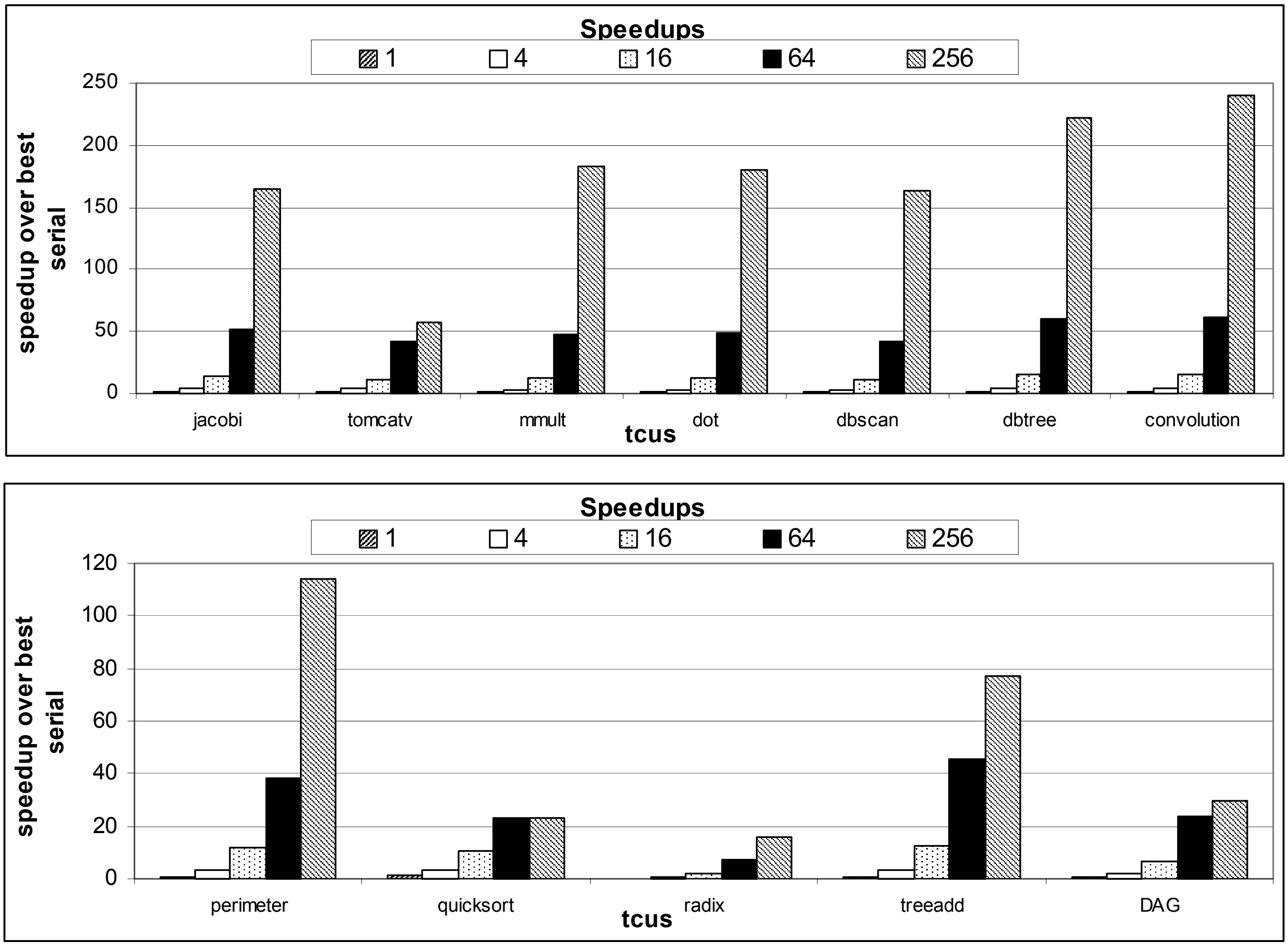

Figure 3: Speedups on XMT simulator

In summary, the results above demonstrate that XMT programs are able to obtain good speedups. Programs that achieve lower speedups do so usually as a result of one of the following: 1) The program doesn't perform a lot of computation (a very small problem size was used); 2) The parallel algorithm involves a lot of work compared to the serial one; 3) An extremely fine-grained parallelism is used. The last case suffers from the overheads that are involved in creating a thread. These overheads are generally very low, but become significant in very fine-grained programs. The next section discusses this issue in more detail, and presents an optimization designed to overcome this problem, allowing efficient fine-grained programming.

6.2 Thread overhead and coarsening

Setting up a parallel region and managing the threads incur an overhead. We can break down this cost to the different elements that it consists of:

- Spawn Set-up: setting up the environment, broadcasting data.
- TCU-Init: initializing the TCUs context;
- Thread Overhead: emulating threads on each TCU - obtain a thread ID and verify that it is less than the spawn size.
- Load Imbalance: idling at the end of a spawn until all threads complete, then transitioning back to serial mode.

We examine the costs that the different kinds of overheads incur. Several trends are evident. 1) overheads are generally very low. In particular, even for very small problem sizes, and very fine-grained parallelism – the system obtains good speedups, which are further improved by our optimizations. 2) Setting up the parallel region is a cheap operation - the Spawn Set-up 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 lots of spawns and joins still perform well. 3) The most dominant overhead is the one charged to thread creation. We

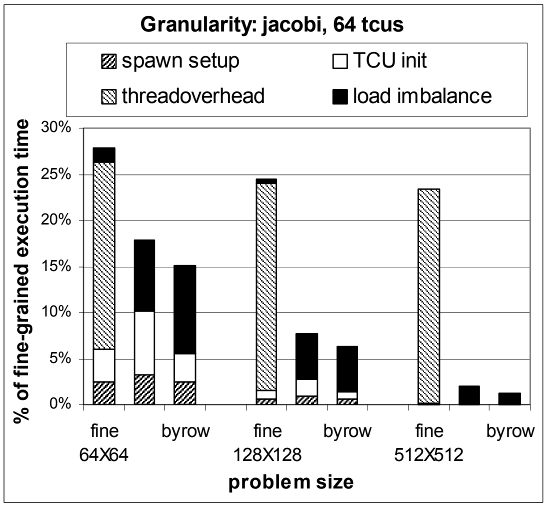

Figure 4: Thread overhead and coarsening

therefore concentrate on optimizations that aim to reduce this overhead. Furthermore, as the number of TCUs increases, the opportunity cost of idle TCUs at the end of the parallel region becomes more significant. Our optimization tries to approach this problem by adjusting for the number of TCUs. 4) The thread structure of the parallel algorithm greatly affects the overhead distribution. Tradeoffs between different choices made by the programmer are clearly illustrated by the overhead distribution.

We use the jacobi kernel to demonstrate how our light-weight mechanisms allow even very fine-grained programs to obtain speedups. Jacobi computes matrix elements independently of one another, and thus allows handling all the entries in parallel. The parallel version consists of one spawn block. For a problem size N, N square threads are spawned. The body of a thread consists of the following 3 source lines:

```
i = TID/size + 1;
j = (TID%size) + 2;
A[i][j]=B[i-1][j] + . . .;
```

The above is translated to 29 assembly instructions, that take between 44 to 66 cycles. The overhead involved in obtaining a new thread takes 20 cycles - constituting a significant portion of the actual work the thread performs. We examined a different parallel version of jacobi, in which we spawn only N threads, where each one operates on an entire row of the matrix. Here load balancing becomes the dominant factor, and overall, the coarser-grained version performs better.

Thread coarsening is a common technique to improve the performance of multi-threaded programs. However, when programming in XMT, we would like the programmer to be relieved of this consideration – in fact, the programmer is encouraged to express as much parallelism as possible, as fine- grained as it may be. Cases in which the low overhead thread constructs are not efficient enough (as is the case for jacobi) are automatically detected by our compiler and optimized. The optimization changes the thread structure of the spawn block - instead of having many short threads, the spawn-block is transformed to have fewer but longer threads ("clustered-threads").

Figure 4 presents the overhead breakdown with different decomposition granularities. We report results for three problem sizes, comparing three versions for each: 1) "fine" – the original fine-grained program; 2) "cl" – the fine-grained version automatically coarsened by the XMT compiler; 3) "byrow" – the coarse-grained version. Observe that fine pays a heavy penalty for thread overhead, while byrow's most significant cost is due to load imbalance. The relative cost of thread overhead increases as the problem size becomes larger. The clustered version is able to eliminate most of the thread overhead, making it competitive with the coarse-grained program. This optimization can be tuned to reduce the load imbalance cost by reserving unclustered threads at the tail of a spawn, at the expense of a minimal increase in thread overheads.

6.3 Nested fork evaluation

To demonstrate the potential of the thread forking capability, we highlight the difference between synchronous and asynchronous programming styles on the dag computation. The synchronous version uses frequent spawns and joins, while the asynchronous forks new threads to explore nodes as they are discovered. Figure 5 shows overhead breakdowns and speedups for both versions on two different graph sizes. 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.

6.4 Memory access costs

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 all the programs tested. As an example, Figure 6 shows the breakdown of TCU time between active processing (CPU), memory stalls, and idling for dbtree. As the number of TCUs increases, the memory stall share does not excessively increase. (Also note idle time becomes a larger share of TCU time in higher TCU configurations, as may be expected.) We see this trend

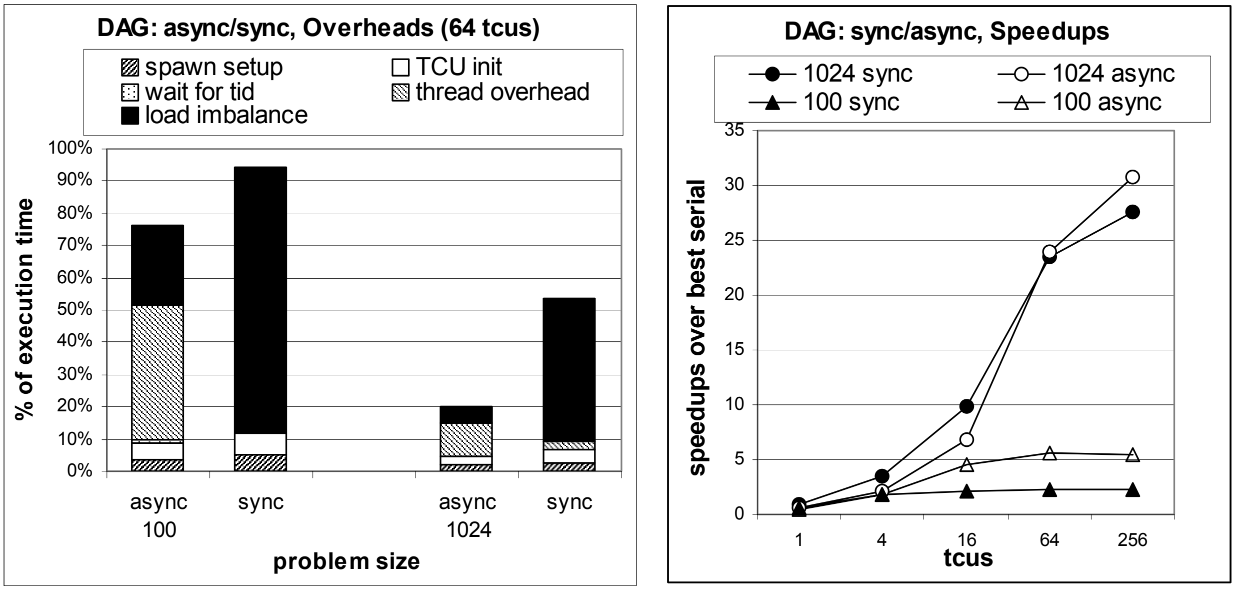

Figure 5: Fork versus synchronous

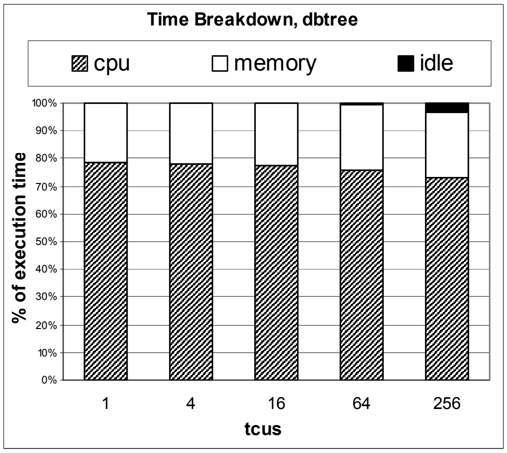

Figure 6: Memory and load balance, dbtree

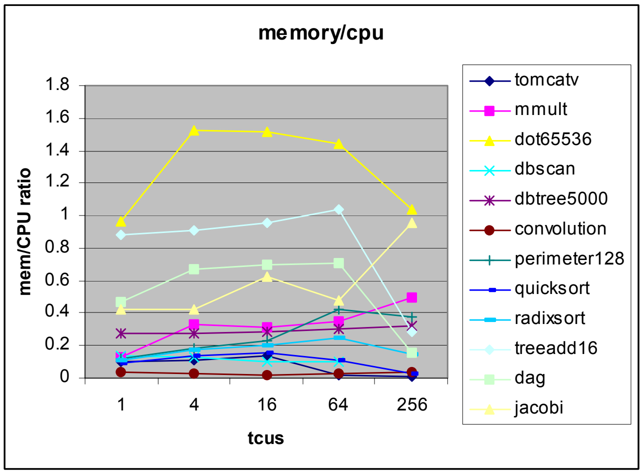

Figure 7: Memory/cpu

quite consistently across a broad range of benchmarks in Figure 7. This can be attributed to the XMT architecture design, which relies on a high-bandwidth, scalable on-chip memory system.

7. Related work

XMT has tried to build on available technologies to the extent possible. The relaxation in the synchrony of PRAM algorithms is related to the work of [CZ89] on asynchronous PRAMs. Basic insights concerning the use of a prefix-sum like primitive go back to the Fetch-and-Add or Fetch-and-Increment [FG91] primitives (cf. [AG94]).

MIT's Cilk [FLR98] provides a multi-threaded programming interface and execution model. There are two important differences in scope. First, since Cilk is targeted at compatibility with existing SMP machines, load balancing in software is an important element of the project. XMT requires 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. 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.

Another design point for on-chip parallelism is that occupied by chip multiprocessors (CMP) [HNO97]. Research in this area has tended to focus on multiprogramming, rather than fine-grained multithreading of a single task.

Other proposed multi-threaded architectures, such as Simultaneous Multithreading (SMT) or Multiscalar [Franklin93], also feature multiple program counters and make useful points of comparison. Recent work on SMT [TLE+99] has proposed light-weight synchronization methods for multithreading. In fact the Acquire primitive is very similar to the suspend primitive presented here. The two instructions share motivations, since an XMT cluster with shared functional units is very similar in spirit to an SMT processor. Threading support in SMT is not targeted towards supporting a PRAM-style program. XMT, with, for example, the parallel prefix-sum, aspires to scale up to much higher levels of parallelism than other multithreaded architectures consider currently.

The Tera Multi-Threaded Architecture (MTA) [AC+90] supports many threads on a given processor. The processors switch between threads to hide latencies, rather than running multiple threads concurrently. 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 in desktop applications.

Tile based architectures, such as MIT's Raw [WTS97], also expect to scale to high levels of parallelism. However, the Raw utilizes a message-passing model rather than the shared-memory model of XMT.

8. Conclusion

The reliance on compiler and the extension of the architecture simulator marks progress with respect to previous XMT reports. Concretely, the work reported here allows researchers who are interested in performance hungry applications to start considering XMT. (`http://www.umiacs.umd.edu/~vishkin/XMT`)

9. References

[AC+90] R. Alverson, D. Callahan, D. Cummings, B. Koblenz, A. Porterfield, and B. Smith, "The Tera Computer System," Proc. International Conference on Supercomputing, 1990.

[AG94] G.S. Almasi A. Gottlieb. Highly Parallel Computing, Second Edition. Benjamin/Cummings, 1994.

[AUS98] A. Acharya, M. Uysal, J. Saltz. Active Disks: Programming Model, Algorithms, and Evaluation. Proc. ASPLOS'98, October 1998.

[BA97] D. Burger and T. M. Austin, "The SimpleScalar Tool Set, Version 2.0," Tech. Report CS-1342, University of Wisconsin-Madison, June 1997.

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

[CZ89] R. Cole and O. Zajicek, "The APRAM: incorporating asynchrony into the PRAM model," Proc. 1st ACM-SPAA, pp. 169-178, 1989.

[DV99] 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.

[Franklin93] M. Franklin, "The Multiscalar Architecture," Ph.D. thesis. Technical Report TR 1196, Computer Sciences Department, University of Wisconsin-Madison, December 1993.

[FG91] 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.

[FLR98] 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.

[HMT95] H. Hum, O. Macquelin, K. Theobald, X. Tian, G. Gao, P. Cupryk, N. Elmassri, L. Hendren, A. Jimenez, S. Krishnan, A. Marquez, S. Merali, S. Nemawarkar, P. Panangaden, X. Xue, and Y. Zhu. *A design study of the EARTH multiprocessor*. In Proc. Int. Conf. on Parallel Architectures and Compilation Techniques, 1995.

[HNO97] L. Hammond, B. Nayfeh, and K. Olukotun, "A Single-Chip Multiprocessor," IEEE Computer, Vol. 30, pp. 79-85, September 1997.

[TLE+99] D. Tullsen, J. Lo, S. Eggers, H. Levy, "Supporting Fine-Grained Synchronization on a Simultaneous Multi-threading Processor," Proc. of the 5th International Symposium on High Performance Computer Architecture, 1999.

[VDB+98] 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.

[Viskin00] U. Vishkin, "A No-Busy-Wait Balanced Tree Parallel Algorithmic Paradigm," Proc. 12th ACM Symposium on Parallel Algorithms and Architectures (SPAA), 2000.

[WTS97] 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.

[Wilson94] 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.

[WOT+95] 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 1995.