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

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

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.

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 [12] [20] prevent computers from fully exploiting the available on-chip resources. An alternative source of parallelism is at the level of coarse-grained multithreading, and has been traditionally expressed under the shared memory or the message-passing programming models, either directly by programmers, or with the aid of parallelizing compilers.

Two interesting representative design points for multi-threaded single-chip architectures are: chip multiprocessors (CMP) such as the Stanford Hydra [11], and Simultaneous Multithreading (SMT) [17]. 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 [10]. 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 off-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

[*] Supported by NSF grant 9820955

[†] Current address: IBM Research Lab in Haifa, Matam, Haifa 30195, Israel

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 [22], 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.

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 in desktop applications.

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 [18] [7]. 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 [13], [14]. 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 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. 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. 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.

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 [18] 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, [19] 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.

MIT's Cilk [9] 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.

2.2 The XMT Architecture

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.

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 *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 pincs from different TCUs are grouped, and the prefix-sum is computed and broadcast back to the TCUs. This process is pipelined and completes within a constant number of cycles.

The ISA also includes instructions for parallel read of a PR register (prefix-sum with value 0) and for initialization of 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 (i.e., concurrent psm's will be queued). This is equivalent to a fetch-and-increment [8] primitive (cf. [2]). Additional instructions exist to support the nested forking mechanism [14].

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, 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 resources. The clusters can be replicated repeatedly on a given chip. More details about the simulated architecture is described elsewhere [5]. Unlike previous designs, the simulated architecture does not have hard-wired thread management, and uses a banked memory rather than a monolithic memory.

3. THE XMT COMPILER

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

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

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

Above XMT-C program is 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 1: XMT code shape

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 [21]. 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.

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

4. EXPERIMENTAL METHODOLOGY

A behavioral simulator, comparable to SimpleScalar [4], has been developed for an XMT 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 ALUs, 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 ALU ops take 12, 4 and 2 cycles respectively, and square root takes 24 cycles. 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. The L2 cache latency is 6 cycles and memory latency is 25 cycles. A penalty of 4 cycles is charged each way for inter-cluster communication.

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 [15]. 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.

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.

We examined the costs that the different kinds of overheads incur, and observed several trends. Overheads are generally very low. This allows XMT to obtain good speedups even for very small problem sizes, and very fine-grained parallelism. Figure 2 reports spawn-block overhead costs for a matrix multiplication program. 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

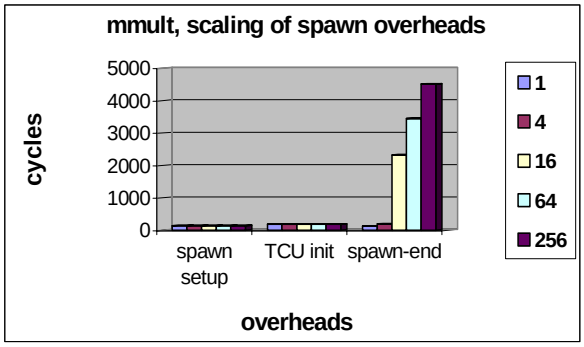

Figure 2: spawn block overheads.

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 less than 0.01% of the entire execution time of the program.

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 3 shows overhead breakdowns 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 (figure 9).

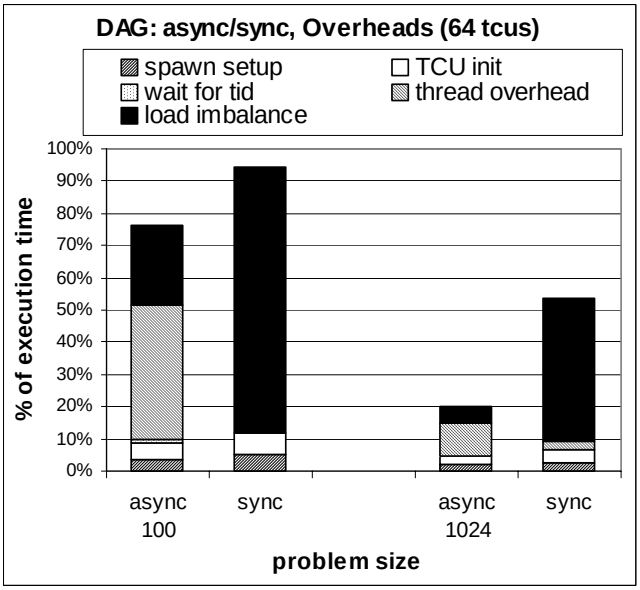

Figure 3: Fork versus synchronous programming

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. As an example, Figure 4 shows the breakdown of TCU time between active processing (CPU), memory stalls, and idling for 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. This can be attributed to the XMT architecture design, which relies on a high-bandwidth, scalable on-chip memory system.

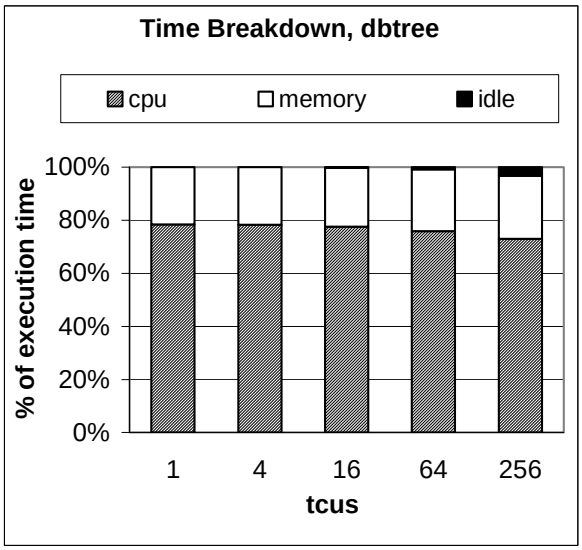

Figure 4: Memory stall behavior, dbtree.

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 5 reports results for the dag program running on 16 TCUs, showing execution time breakdown (number of cycles) for each TCU.

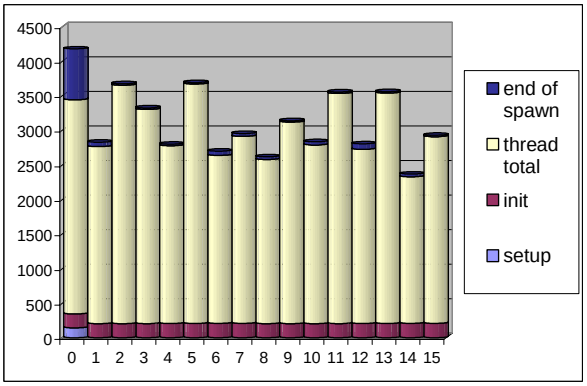

Figure 5: Load Balance in dag.

We observe that the disparity between TCU work loads 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 (range from 96 to 516 cycles)).

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.

The results shown in figure 6 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.

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 to exploit spatial locality, and reduce 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 7 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

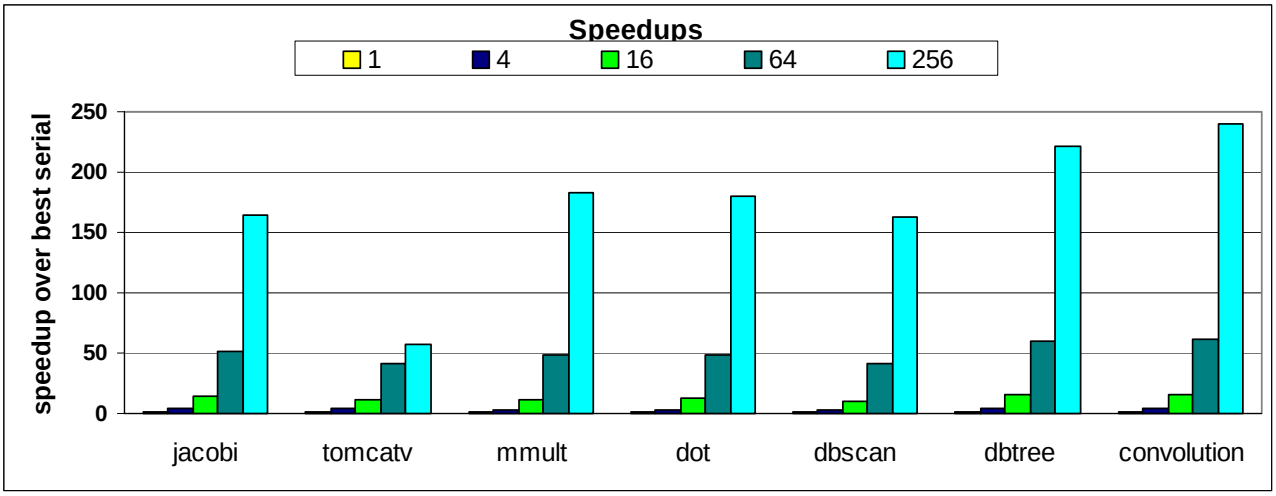

Figure 6: Scaling of speedups

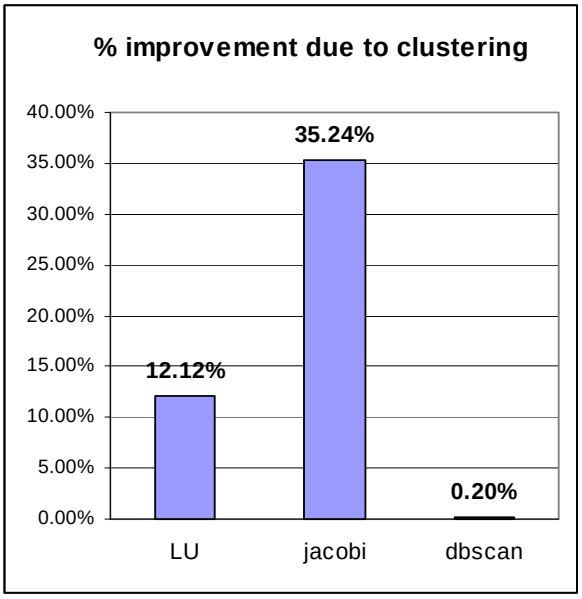

Figure 7: Impact of clustering.

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 inhibit the conservative compilation scheme from extracting computation out of the cluster loop. Consequently, clustering does not improve performance for dbscan.

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 [14]).

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 (figure 8) the following versions for each: 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

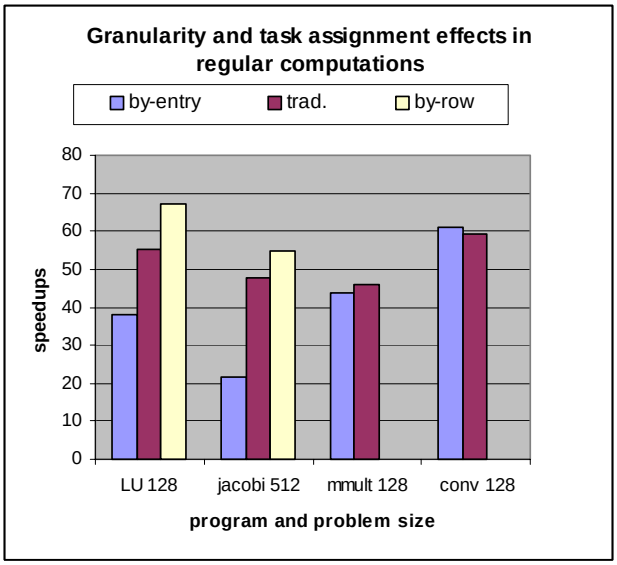

Figure 8: Granularity and Task assignment.

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 [19]. 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 [14] 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 [16]).

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).

We demonstrate programming tradeoffs for irregular applications using the dag program (figure 9). Some trends have been observed for the other types of computations, but are evident here to a greater extent:

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.

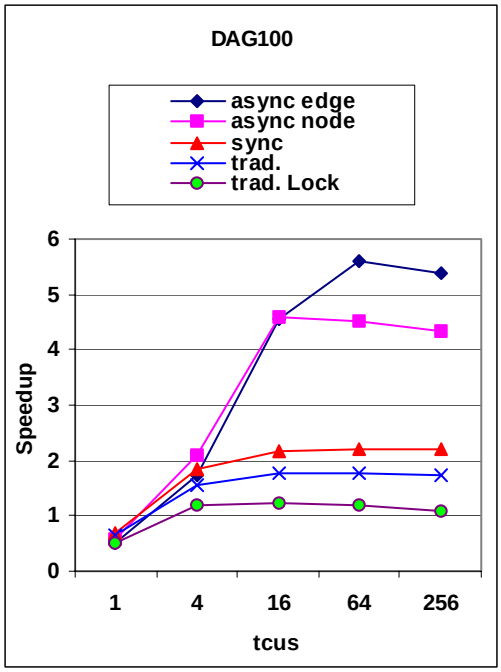

Figure 9 - Synchronization Options in dag (100 nodes, 473 edges)

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 10 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 [23]. 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.

Perimeter and treeadd, benchmarks from Olden, 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.

8. 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.

9. ACKNOWLEDGEMENT

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

10. 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,"

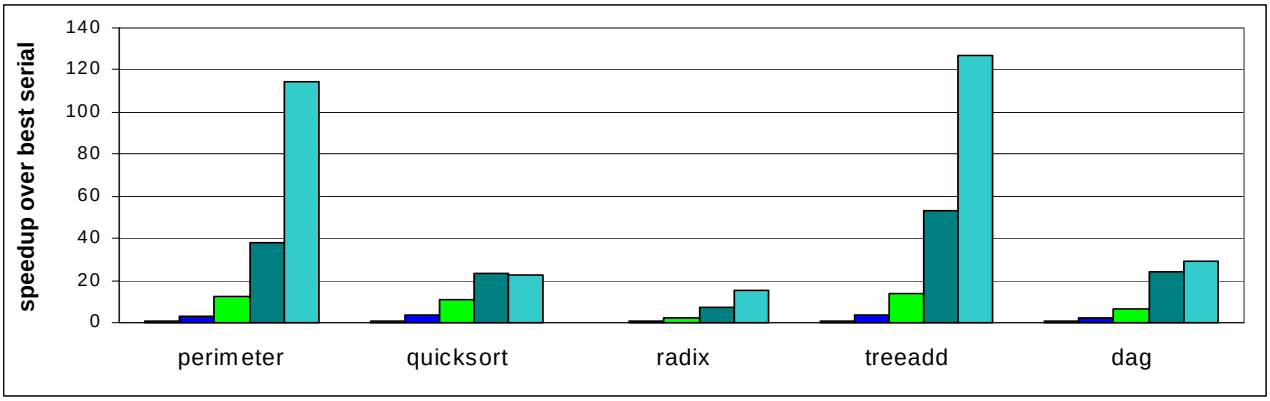

Figure 10: Speedups for irregular applications.

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] 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.

[8] 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.

[9] 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.

[10] 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.

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

[12] 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.

[13] 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 Compliation (MTEAC2000), December 2000. Best Paper Award.

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

[15] J. Nuzman, Masters thesis, University of Maryland, Department of Electrical and Computer Engineering, 2001. In preparation.

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

[17] 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.

[18] 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.

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

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

[21] 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.

[22] 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.

[23] 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.