Submitted to The 6th Workshop on High-Level Parallel Programming Models and Supportive Environments (HIPS-6), April 2001.

Evaluating the XMT Parallel Programming Model

Dorit Naishlos, Joseph Nuzman, Chau-Wen Tseng, Uzi Vishkin

Dept of Computer Science, University of Maryland, College Park, MD 20742

Tel: 301-405-8010, Fax: 301-405-6707

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

ABSTRACT

Explicit-multithreading (XMT) is a parallel programming model designed for exploiting on-chip parallelism. Its features include a simple thread execution model and an efficient prefix-sum instruction for synchronizing shared data accesses. By taking advantage of low-overhead parallel threads and high on-chip memory bandwidth, the XMT model tries to reduce the burden on programmers by obviating the need for explicit task assignment and thread coarsening.

This paper presents features of the XMT programming model, and evaluates their utility through experiments on a prototype XMT compiler and architecture simulator. We find the lack of explicit task assignment has slight effects on performance for the XMT architecture. Despite low thread overhead, thread coarsening is still necessary to some extent, but can usually be automatically applied by the XMT compiler. The prefix-sum instruction provides more scalable synchronization than traditional locks, and the simple run-until-completion thread execution model (no busy-waits) does not impair performance. Finally, the combination of features in XMT can encourage simpler parallel algorithms that may be more efficient than more traditional complex approaches.

1. Introduction

When discussing parallel programming models, the parallel computing community usually considers two models: message-passing and shared-memory. Both models usually require domain partitioning and load balancing. For dynamic, adaptive applications this effort can amount to 25% of the entire code and become a significant source of overhead [Henty00], [SSC+00]. Message-passing in addition requires distributing data structures across processors and explicitly handling inter-processor communication. Performance also decreases for fine-grained parallelism under both models, as the effects of synchronization and communication overhead become a bigger factor.

Many of these issues, however, are 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 Explicit Multi-threading (XMT) programming model. XMT is intended to provide a parallel programming model which is simpler to use, yet efficiently exploits on-chip parallelism

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]. A more recent paper describes the prototype XMT programming environment, including the XMT compiler and simulator [N+00]. In this paper, we describe features of the XMT programming model that were designed for exploiting on-chip parallelism, and evaluate their impact on both programmability and performance using the XMT programming environment.

The main contributions of this paper are as follows:

- We discuss and evaluate features of XMT designed to exploit on-chip parallelism.
- We examine their effect on programmability for several interesting application kernels.
- We experimentally evaluate the impact of XMT features on performance using the XMT compiler and simulator.

We begin by reviewing the XMT multi-threaded programming model. We then briefly discuss the XMT architecture and environment, including a compiler and behavioral simulator. We examine the impact of each feature of XMT on programmability and performance. Finally, we present a comparison with related work and conclude.

2. XMT Programming Model

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.

The user-level XMT language is an extension of standard C. The following example XMT program copies all non-zero values of array *A* to *B* in an arbitrary order:

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

The programming model has a number of key features:

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

- Dynamic *forking* of additional virtual threads

A parallel region is delineated by *spawn* and *join* statements. Every thread executing the parallel code is assigned a unique thread ID, designated TID. Shared accesses are synchronized with a prefix-sum instruction (*ps*), similar to an atomic fetch-and-increment. It can be combined by the hardware to form a multi-operand prefix-sum operation. For both simplicity and efficiency, threads always run to completion without busy-waiting. XMT does not allow for nested initiation of a spawn within a parallel spawn region [VDB+98], but a thread can perform a fork operation to introduce a new virtual thread as work is discovered [Vishkin00].

3. XMT Environment

The XMT environment consists of a prototype XMT compiler and behavioral simulator. The XMT compiler consists of two passes. The front end is a translator based on the *SUIF* compiler system [Wilson94] that converts XMT constructs into regular C code with assembly templates. It also detects all parallel regions delineated by spawn-join statements and transforms them into parallel function calls. The back end is based on GNU's *gcc* and builds an executable for the C code output by the front end [N+00].

The XMT behavioral simulator is comparable to SimpleScalar [BA97]. The fundamental units of execution for the simulated machine are the multiple thread control units (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 [BNF+99].

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 inter cluster communication.

4. XMT Programming and Performance Features

Traditional shared memory programming consists of assigning chunks of work to processes, usually as coarse-grained as possible, while locks and barriers are used for synchronization. The following are the main programming concepts that distinguish XMT from traditional parallel programming:

1) No task assignment.

2) Fine-grained parallelism.

3) No busy-wait.

In this section we examine each of the above features, and how they affect the way programs are written in XMT. Furthermore, we study the effects of these features on performance.

4.1 No Task Assignment

XMT relieves the programmer from the task of assigning work to processors. The programmer can think in terms of virtual threads, without worrying about low-level considerations such as the number of processing units and load balance between them. Instead of directly spawning a thread for each block of work, traditional parallel programming puts a lot of effort into the task of grouping blocks of work together in a good way (with respect load-balance and locality). This effort is translated to programmer time, line count, and code complexity. Sometimes, a program benefits from grouping because it may allow saving duplicate work. If the blocks of work are very small, this grouping serves as thread coarsening. However, in some cases, by incurring extra work, it can even result in a less efficient program.

We evaluate the effects of task composition on each of three programs by comparing two versions – XMT and Traditional. The traditional version always spawns exactly one thread for each TCU, and a loop is added to the thread body to span through all the blocks of work that are assigned to the thread:

XMT	Traditional
`spawn(N*N,0);`	`spawn(tcus,0);`
`{`	`{`
	`lb =N*N*TID/tcus;`
	`ub =N*N*(TID+1)/tcus;`
	`for(m=lb;m<ub;m++){`
`x = C[0][TID];`	`x = C[0][m];`
`i = TID/N; j = TID%N;`	`i = m/N; j = m%N;`
`for (k=0; k<N; k++){`	`for (k=0;k<N;k++){`
`x +=`	`x +=`
`A[i][k]*B[k][j];`	`A[i][k]*B[k][j];`
`}`	`}`
`C[0][TID] = x;`	`C[0][m] = x;`
	`}`
`}`	`}`
`join();`	`join();`

Note that even when the entire task of assignment involves only the few source lines (in bold font), we may still get up to 30% increase in length of code, in some cases accompanied with a decrease in performance.

We now examine the experimental results for the three programs (Figure 1). For both matrix multiplication and 2-D image convolution, the XMT version spawns a thread for each entry, while the traditional version clusters the entries, and spawns a thread for each cluster. A more irregular computation, the third program finds maximum paths in a DAG. The XMT version spawns a thread for each node, while in the traditional version each thread handles a cluster of nodes. (Note that we choose to show here the "xmt-sync" variant, as it is the closest to traditional version, though not the most efficient. Details on this and other DAG implementations appear in section 4.3.)

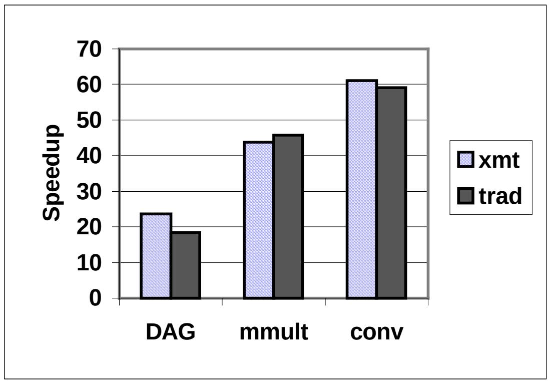

Figure 1 - No task assignment (64 TCUs)

For mmult and convolution, both versions achieve similar speedups. The traditional mmult is able to amortize some duplicate work, while the XMT version of convolution takes the lead by avoiding some task assignment overhead. For DAG, load balancing issues come into play, penalizing static assignment in the traditional version. For 16 TCUs, costs due to load imbalance constitute 35% of the traditional program running time versus 16% for XMT.

4.2 Fine-grained Parallelism

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. The XMT compiler detects such cases, and automatically transforms them such that fewer but longer threads are used.

To evaluate the effects of granularity on performance, we use the following three programs: LU, a linear algebra program that computes lower-upper matrix factorization. Jacobi, a 2-D PDE kernel, and dbscan – a database kernel that emulates an SQL query on a non-indexed attributes relation. We compare several versions for each: 1) fine-grained: each thread handles one entry, 2) by-row: each thread computes an entire row, 3) clustered: the fine-grained version coarsened by the XMT compiler, 4) traditional: by-row, with the work assigned to processors (as described in the section above). (For dbscan we compare only 3 versions - fine-grained, clustered, and traditional).

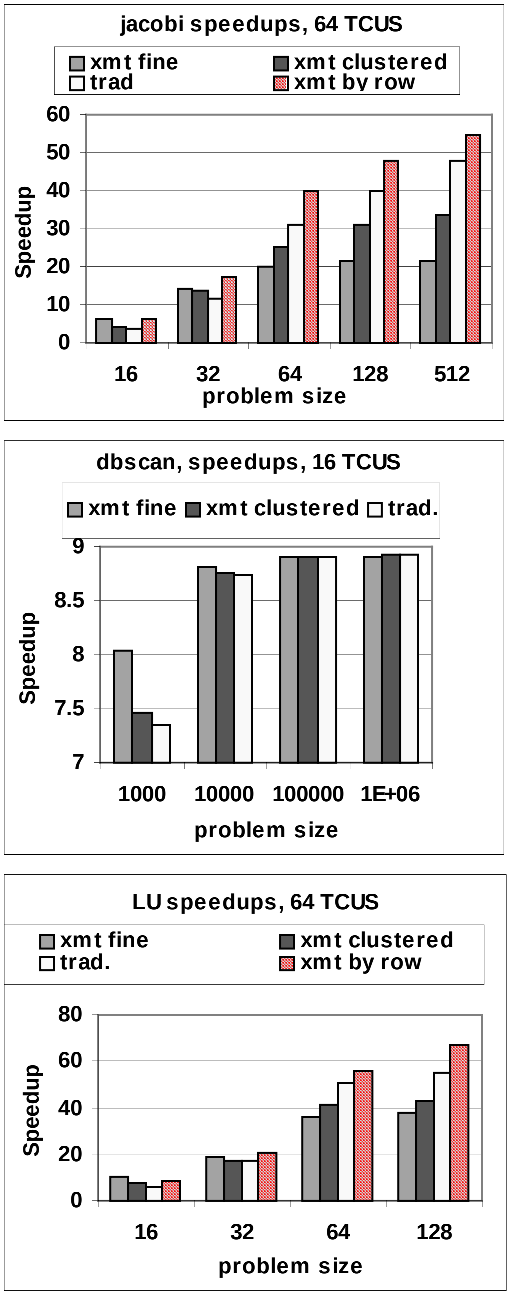

Figure 2 - Fine-grained parallelism

All three programs demonstrate the same behavior (Figure 2). For the smallest problem sizes, the best speedups are achieved by the fine-grained version, where the coarsest version (traditional) gets the lowest speedups. As the problem size increases, the coarser versions beat the fine-grained one, while the by-row version is the fastest version. Though the advantage that the fine-grained versions demonstrate for the smaller sizes is not decisive, it suggests that in the general case, algorithms for more irregular applications may benefit. As an example, in section 4.4 we describe how our implementations of radixsort and quicksort take advantage of fine-grained parallelism. For more regular applications however, the XMT compiler can make fine-grained programs competitive with the coarse-grained versions by automatically clustering the parallel regions. This allows the programmer to ignore granularity considerations, and directly write the easier fine-grained version.

4.3 Synchronization: Scalable Mechanisms and Asynchronous Algorithms

Traditional parallel programs typically use locks and barriers for synchronization. As we will demonstrate, these synchronization mechanisms can be efficiently supported in XMT. However, typically XMT addresses synchronization in a manner that minimizes busy-waiting. In many cases, the parallel prefix sum operation can be used instead of a lock, and the join serves as a barrier. Alternatively, an XMT methodology can often suggest an entirely different algorithm. We examine these alternatives with two representative programs, dot and DAG. We show results for relatively small input sizes, where the relative costs of the various techniques are easily seen. While differences tend to be less dramatic with larger workloads, the same trends are evident.

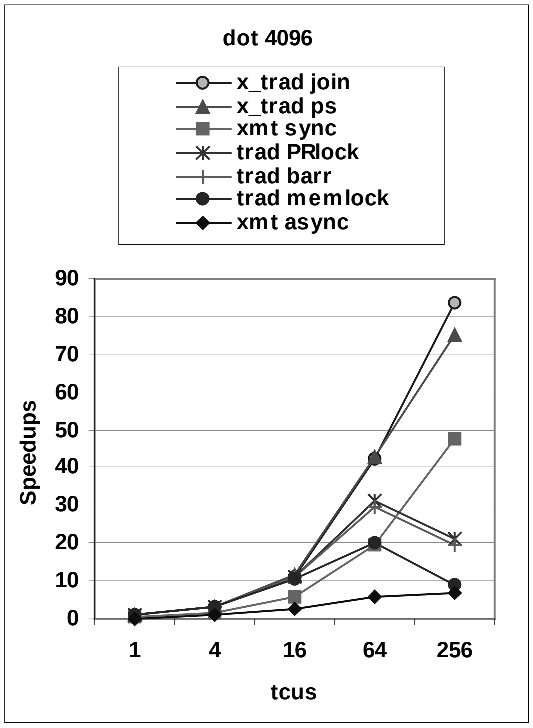

Figure 3 - Synchronization Options in Dot

(4096 elements)

Dot product is an example of a common reduction task. We compare the following versions for dot:

1) XMT-algorithmic-style programs, using a binary tree structure for no-busy-wait synchronization [Vishkin00]. We wrote two programs in this style, one propagates values up the tree in a synchronous fashion, involving a spawn and join for each layer of the tree ("xmt-sync"). The other version propagates the values in an asynchronous fashion, involving only one spawn-join block, within which the threads advance as far as they can ("xmt-async").

2) Traditional style programs, in the sense that the problem is decomposed in a coarse-grained fashion between the processing units. However, XMT utilities are used for synchronization. In the "x_trad-ps" version, the TCUs update the global dot product atomically with their portion of the computation using the parallel prefix–sum mechanism, instead of locking. In the "x_trad-join" version, after all the TCUs have completed computing their portion, they join, instead of using a barrier, and the global dot product is computed serially after the join by a single TCU.

3) Traditional style programs, using busy-wait synchronization. The "trad_PRlock" uses the most efficient but somewhat specialized parallel prefix-sum operation, while "trad_memlock" uses a less scalable but more general fetch-and-add mechanism. "trad-barr" uses a barrier.

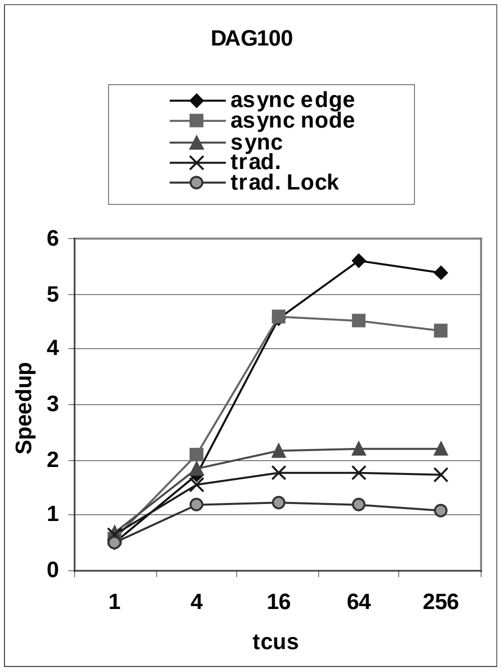

Figure 4 - Synchronization Options in DAG

(100 nodes, 473 edges)

The general trend is that programs using non-XMT synchronization scale poorly with the number of TCUs compared to the others (Figure 3). The exception is the "xmt-async" version due to the amount of storage and extra work that it involves.

This trend in synchronization primitive scalability is further reinforced when examining the performance of different versions for the DAG program. In addition, the irregular nature of the computation makes it a good candidate for a less synchronous programming style. Especially with sparser graphs, asynchronous programs should excel by enabling parallelism as soon as it is discovered. We compare the following versions [Vishkin00]:

1) XMT style, synchronous fashion: "sync". A thread is spawned for each node with in-degree 0. Each thread steps through the outgoing edges of the node, and decrements the in-degree of the sink nodes. After a join, threads are spawned for newly in-degree 0 nodes. The process is repeated until all nodes are processed.
2) Traditional style. These versions are based on the "sync" version, adding the necessary decomposition. One program that uses the prefix-sum as a synchronization mechanism ("trad"), and another that uses locks instead ("trad-lock").
3) XMT style, asynchronous fashion: "async-node" and "async-edge". The async-node spawns a thread for every node as in "sync". Whenever a thread decrements the in-degree of a sink node to zero, it forks a thread to process that node. Async-edge is similar, but a thread processing a node forks a thread to process every outgoing edge. Async-edge is the finest-grain, least-synchronous of the versions.

As expected, the more synchronous the algorithm is, the worse it scales with the number of TCUs. This trend can be seen by comparing the relative performance of async-edge, async-node, and sync (Figure 4).

We again see the effect of synchronization mechanism overhead on scalability. The relative performance of trad and trad-lock illustrates the superiority of prefix-sum to traditional busy-wait locking.

4.4 Two Case Studies

In this section we provide two examples of how the combination of features in XMT enables new approaches that outperform more traditional algorithms.

4.4.1 Radix

This integer sort consists of the stable radix sort routine that is applied iteratively to each "digit" of the input until fully sorted. For every value a digit might take, we say there is a bin for the key to go in. By counting the number of keys for each bin, we can determine a final rank for each bin.

SPLASH radix uses the common parallel algorithm of [B+91]. It operates with P processors on N input keys. Each processor is assigned a continuous partition of N/P keys, and locally computes bin counts from its own partition. Since each processor has its own set of bin counters, a global ranking operation must be performed. A binary tree of bin counter arrays is formed. Each processor ranks its bins locally and then either sums results from other processors further up the tree, or waits for partial sums to be propagated back down the tree. Finally, with globally ranked bins, each processor can copy each of its keys in sequence to the output array.

One of the hurdles to scalability is that each processor needs its own set of bin counters. The more processors that are applied to a problem, the more time must be spent doing global ranking. This effectively limits the P for the SPLASH radix. In an XMT style program, significantly more parallelism can be expressed.

Due to the need to preserve equal-key order, we are limited to the P sequential threads to serially copy keys within individual partitions. However, the bin counting step can be much more fine-grained. Provided that threads cope with simultaneous access to the same bin counter (e.g. with the prefix-sum primitive), it is possible to spawn a separate thread for each key of input.

The global ranking step can also be finer-grained. SPLASH operates with a granularity of an entire bin counter array. The XMT approach does not insist that the programmer wrestle with data locality. In this light, the global ranking step can be considered to be a single, large prefix-sum operation, where the inputs are the interleaved bin counters from all the processors. It is then a simple matter to apply a fine-grained binary-tree parallel prefix algorithm.

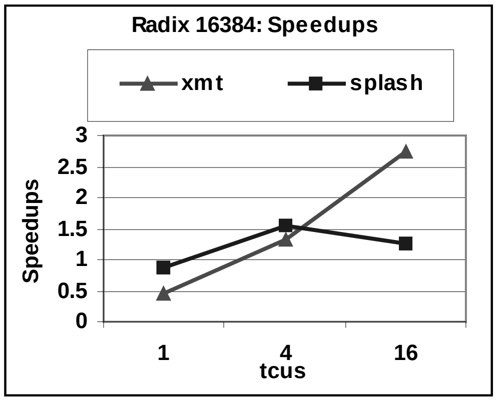

Figure 5 - XMT vs Splash

There is certainly a cost to the more fine-grained algorithm. Without automated coarsening of the XMT threads, the coarser-threaded bin counting clearly has the work advantage. With 1 TCU, the XMT algorithm performs nearly twice the work of SPLASH (Figure 5). However, with 16 TCUs, the SPLASH algorithm is already doing worse than it did with 4 TCUs. In contrast, the finer-grained program easily scales to 16 TCUs. (Note that these results are for an input of 16K keys, while SPLASH defaults to 256K keys.)

4.4.2 Quicksort

Quicksort provides another simple illustration of how XMT might be able to express parallelism that is traditionally neglected. A classic example of the divide-and-conquer approach, quicksort is a recursive sort using pivots.

The traditional parallel quicksort follows this divide-and-conquer approach. After splitting a partition, a thread forks a new thread to sort the right side, and does the left side itself.

In XMT, the forking need not stop when the full machine parallelism is reached. Newly created partitions are queued up, and automatically assigned to threads without complicated programmer intervention. This dynamic load balancing is particularly useful for handling the unpredictable partitioning of quicksort.

This approach is still less than ideal. At the beginning of the program, only one thread is active. A machine can not be utilized fully until enough partitions have been created. Unfortunately, the partitions encountered at the start of the computation are the largest and therefore take the longest time to split.

A fine-grained XMT program can parallelize the partition step. A thread is spawned for each element of the partition. A prefix-sum to either a left counter or a right counter determines the output rank of the element.

Since the latter method requires synchronization at each partition, we do not wish to use this algorithm throughout the program. The optimal solution is to start with the fine-grained, synchronous parallel partitioning algorithm, and then switch to the traditional divide-and-conquer when a sufficient number of partitions are available. With a 64 TCU configuration operating on 16,384 elements, this hybrid approach is more than twice as fast as the traditional approach.

5. Related Work

Recent work on comparing different parallel programming models [Henty00], [CE00], [SSC+00], [CDS00], typically focuses on the shared-memory and message-passing programming models on multiprocessor systems. Our work attempts to examine parallel programming with respect to the different assumptions implied by an on-chip environment.

Various other projects explore on-chip parallel architectures: CMP [HNO97], Multiscalar [Franklin93], SMT [TLE+99], and Raw [WTS97]. The current paper is targeted toward exploring shared-memory parallel algorithms as applied to scalable on-chip architecture.

6. Conclusion

This paper presented features of XMT, a parallel programming model designed for exploiting on-chip parallelism. With prototype compiler and architecture simulator, we studied XMT programming in areas where parallel computing has underperformed in the past: very fine-grained parallelism; smaller problem sizes; and unpredictable, irregular computations.

XMT features encourage programmers to write high level programs with more fine-grained parallelism, without worrying about assigning threads to processors. We found the XMT architecture and compiler can usually support this simple fine-grained programming style with little loss in performance for on-chip multiprocessors.

When overheads are low enough so that parallelism can be leveraged even for small inputs, many more tasks can potentially be sped up.

The flexible programming style also encourages the development of new algorithms to take advantage of properties of on-chip parallelism. We demonstrated that XMT is especially useful for more advanced applications with dynamic, irregular access patterns.

7. References

[B+91] G. E. Blelloch et. al, “A Comparison of Sorting Algorithms for the Connection Machine CM-2,” Symposium on Parallel Algorithms and Architectures, pp. 3-16, July 1991.

[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.

[CDS00] B.L. Chamberlain, S.J. Deitz, L. Snyder, “A Comparative Study of the NAS MG Benchmark across Parallel Languages and Architectures,” Proc. of Supercomputing (SC), 2000.

[CE00] F. Cappello, D. Etiemble, “MPI versus MPI+OpenMP on the IBM SP for the NAS Benchmarks,” Proc. of Supercomputing (SC), 2000.

[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.

[Henty00] D.S.Henty, “Performance of Hybrid Message-Passing and Shared-Memory Parallelism for Discrete Element Modeling,” Proc. of Supercomputing (SC), 2000.

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

[N+00] D. Naishlos, J. Nuzman, C.-W. Tseng, and U. Vishkin, "Evaluating Multi-threading in the Prototype XMT Environment," University of Maryland Technical Report, October 2000.

[SSC+00] H. Shan, J.P. Singh, L. Oliker, R. Biswas, “A Comparison of Three Programming Models for Adaptive Aplications on the Origin2000,” Proc. of Supercomputing (SC), 2000.

[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.