FPGA-Based Prototype of a PRAM-On-Chip Processor

Xingzhi Wen , Uzi Vishkin
University of Maryland Institute for Advanced Computer Studies (UMIACS)
Electrical and Computer Engineering, University of Maryland
College Park, Maryland, USA
hsmoon@umd.edu, vishkin@umd.edu

ABSTRACT

PRAM (Parallel Random Access Model) has been widely regarded a desirable parallel machine model for many years, but it is also believed to be "impossible in reality." As the new billion-transistor processor era begins, the eXplicit Multi-Threading (XMT) PRAM-On-Chip project is attempting to design an on-chip parallel processor that efficiently supports PRAM algorithms. This paper presents the first prototype of the XMT architecture that incorporates 64 simple in-order processors operating at 75MHz. The micro-architecture of the prototype is described and the performance is studied with respect to some micro-benchmarks. Using cycle accurate emulation, the projected performance of an 800MHz XMT ASIC processor is compared with AMD Opteron 2.6GHz, which uses similar area as would a 64-processor ASIC version of the XMT prototype. The results suggest that an only 800MHz XMT ASIC system outperforms AMD Opteron 2.6GHz, with speedups ranging between 1.57 and 8.56.

Categories and Subject Descriptors

C.1.4 [**Parallel Architectures**]

General Terms

Algorithms Design Performance

Keywords

Parallel Algorithms, PRAM, On-chip parallel processor, Ease-of-programming, Explicit multi-threading, XMT

1. INTRODUCTION

The eXplicit Multi-Threading[1] (XMT) on-chip general-purpose computer architecture is aimed at the classic goal of reducing single task completion time. XMT is a parallel algorithmic architecture in the sense that: (i) it seeks to provide good performance for parallel programs derived from Parallel Random Access Machine/Model (PRAM) algorithms, and (ii) it offers methodology for advancing from PRAM algorithms to XMT programs, along with a performance metric and its empirical validation [27]. Ease of parallel programming is now widely recognized as the main stumbling block for extending commodity computer performance growth (e.g., using multi-cores). XMT provides a unique answer to this challenge. A 64-processor, 75MHz computer based on field-programmable gate array (FPGA) technology was built at the University of Maryland (UMD). A brief announcement [29], which reported this first commitment to silicon of XMT, was a preamble to the current paper. Six additional kernel benchmarks are added to the test suite and the performance of an 800MHz XMT ASIC version is projected using cycle accurate emulation. The XMT concept was introduced in [28]. An architecture simulator and speed-up results on several kernels were reported in [21]. The new computer is a significant milestone for the broad PRAM-On-Chip project at UMD. In fact, contributions in the current paper include several stages since SPAA'01 [21]: completion of the design using a hardware description language (HDL), synthesis into gate level netlist, as well as validation of the design in real hardware.

Discussion of the broader goals of XMT are deferred to the closing section at the end of this article. These goals are to address the current need for a general-purpose on-chip parallel computer architecture, which: (i) is easy to program; (ii) gives good performance with any amount of parallelism provided by the algorithm; namely, up-and down-scalability including backwards compatibility on serial code; (iii) supports application programming (in standard application languages, such as VHDL/Verilog, OpenGL, MATLAB); and (iv) fits current chip technology and scales with it.

PRAM

The PRAM virtual model of computation is a generalization of the Random Access Machine (RAM) model, the basic sequential model exposed to programmers in traditional programming languages, that assumes that any memory access or any (logic, or arithmetic) operation takes unit time. The PRAM assumes that any number of concurrent accesses to a shared memory take the same time as a single access. In the Arbitrary Concurrent-Read Concurrent-Write (CRCW) PRAM concurrent access to the same memory location for reads or writes are allowed. Reads are resolved

[1]Partially supported by NSF grant CCF-0325393. The XMT home page is at: umiacs.umd.edu/users/vishkin/XMT

CF'08, May 5–7, 2008, Ischia, Italy.

before writes and an arbitrary write unknown in advance succeeds. Design of an efficient parallel algorithm for the Arbitrary CRCW PRAM model would seek to optimize the total number of operations the algorithms perform ("work") and its parallel time ("depth") assuming unlimited hardware. Given such an algorithm, an XMT program is written in XMTC, which is a modest single-program multiple-data (SPMD) multi-threaded extension of C that includes 3 commands: Spawn, Join and PS, for Prefix-Sum; PS is a Fetch-and-Add/Increment-like command. The program seeks to optimize: (i) the length of the (longest) sequence of round trips to memory (LSRTM), (ii) queuing delay to the same shared memory location (known as QRQW), and (iii) work and depth (as per the PRAM model). Optimizing these ingredients is a responsibility shared in a subtle way between the architecture, the compiler, and the programmer/algorithm designer, where the latter (the role of the programmer) is expected to decline as XMT matures. See also [27]. For example, the XMT memory architecture requires a separate round-trip to the first level of the memory hierarchy (MH) over the interconnection network for each and every memory access; this happens unless something (e.g., prefetch) is done to avoid it; and our LSRTM metric accounts for that.

The well-developed PRAM algorithmic theory is second in magnitude only to its serial counterpart, well ahead of any other parallel approach. Circa 1990 popular serial algorithms textbooks already devoted a big chapter to PRAM algorithms. For many years, theorists (UV included) also claimed that the PRAM theory is useful. However, the PRAM was generally deemed useless (e.g., see the 1993 LOGP paper [7]). From the mid-1990s, PRAM research was reduced to a trickle and most researchers abandoned it, and some later book editions discarded their PRAM chapters. The 1998 state-of-the-art is reported in the parallel computer architecture book by Culler and Singh [8] ".. breakthrough may come from architecture if we can truly design a machine that can look to the programmer like a PRAM". We are now a step closer as hardware replaces a simulator. Advancing the PRAM implementability from impossible to available, in practice as well as in perception, is a strategic objective of our overall work. The new computer provides freedom and opportunity to pursue PRAM-related research, development and education without waiting for vendors to make the first move. The new XMT computer is 11-12K times faster than our XMT cycle-accurate simulator (46 minutes replace 1 year): heavier programs and applications and larger inputs to study scalability can now be run.

We finish this introduction with an interesting deployment example that enforces the magnitude of the adoption promise of a PRAM-related machine.

Informal PRAM/XMT Pilot with High School Students

As PRAM algorithms are based on first principles that require relatively little background, a full day (300-minute) PRAM/XMT tutorial was offered to a dozen high-school students in September 2007. Followed up with only a weekly office-hour by an undergraduate assistant, some strong students have been able to complete 5 of 6 assignments given in a graduate course on parallel algorithms. This informal course was part of a computer club that met after 8 hours of regular classes (i.e., no academic credit). This success story suggests that since high school students from more diverse technical backgrounds and abilities can study this material, hopefully everybody who can study standard serial programming, should be able to study this material if taught in a regular for-credit class by a K-12 teaching professional.

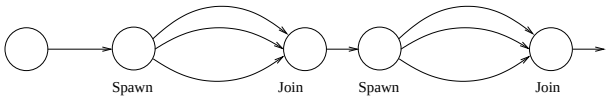


Figure 1: Serial and parallel execution modes

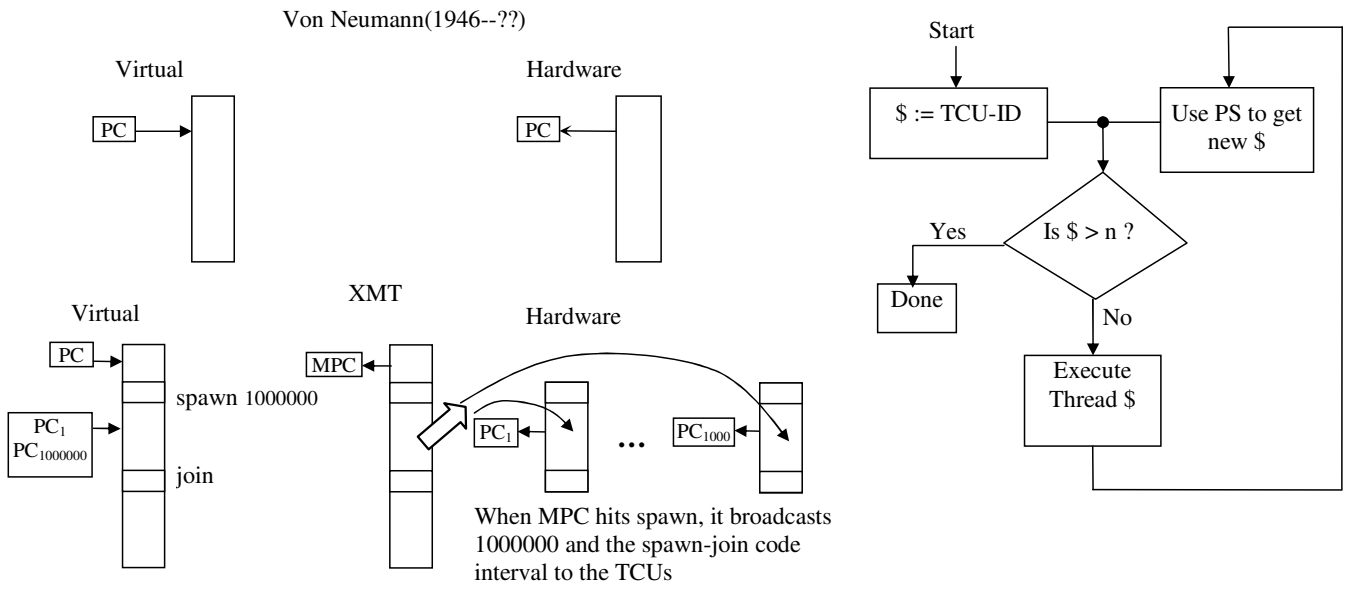
In section 2, background on XMT is briefly reviewed. In section 3, the basic microarchitecture of the prototype as well as 3 enhancements are presented. In section 4, Some performance results are presented and discussed. Related work is discussed in section 5 and section 6 concludes the paper.

2. BACKGROUND ON THE XMT FRAMEWORK

As space limitations prevent us from reviewing both the broad concepts of the XMT and our contribution, we must refer the readers to [21, 28]. However, a brief review of some basic concepts are presented in this section to make this paper as self contained as possible.

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 that has two executing modes: serial and parallel. The two instructions, *spawn* and *join*, specify the beginning and end of a parallel section (executed in parallel), respectively. See Fig 1. An arbitrary number of virtual threads, initiated by a spawn and terminated by a join, share the same code [25]. 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. An algorithm designed with this property in mind permits each thread to progress at its own speed from its initiating spawn to its terminating join, without ever having to wait for other threads; that is, no thread busy-waits for another thread. The implied "independence of order semantics" (IOS) allows XMT to have a shared memory with a relatively weak coherence model. An advantage of using this easier to implement SPMD model is that it is also an extension of the classical PRAM model, for which a vast body of parallel algorithms is available in the literature. 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 [11]) 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 very fast 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 XMTC high-level lan-



(a) Program counter + stored program

(b) TCU program snippet

Figure 2: XMT execution model

guage is an extension of standard C. The extensions are described individually in [3]. A parallel region is delineated by spawn and join statements. Synchronization is achieved through the prefix-sum and join commands. Every thread executing the parallel code is assigned a unique thread ID, designated $. 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 XMTC 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 XMTC) to get a unique index into B where it can place its value.

```
psBaseReg x =0;
spawn(0,n-1){
    int e;
    e = 1;
    if(A[$]) != 0){
        ps(e,x);
        B[e] = A[$];
    }
}
```

2.2 The XMT architecture

Perhaps the most important distinguishing characteristics of an XMT architecture are low-overhead mechanisms for the management of parallelism. New elements not present in standard microprocessor design are introduced for the purpose of supporting the parallel programming model. The XMT programming model allows programmers to specify an arbitrary degree of parallelism in their code. Clearly, real hardware has finite execution resources, so in general all threads cannot execute simultaneously. In an XMT machine, a thread control unit (TCU) executes an individual virtual thread. Upon termination of a virtual thread, the TCU performs a prefix-sum operation in order to receive a new (virtual) thread ID. The TCU will then emulate the thread with that new ID. All TCUs repeat the process until all the virtual threads have been completed. A master thread control units (MTCU) orchestrates the TCUs. Figure 2 illustrates this: (i) through a comparison with the von-Neumann stored program and program counter apparatus (2 (a)), and (ii) through a snippet of the program of a TCU (2 (b)).

We begin with Figure 2 (a). Its upper part, entitled "von-Neumann (1946–??)," illustrates the program counter apparatus in serial machines, which has dominated general-purpose computing since 1946; it is not yet clear whether, and if yes when, its reign will end. The right hand side (of the upper part of Figure 2 (a)) depicts the hardware apparatus, where one command at a time is brought to the program counter. The left hand side (of the upper part of Figure 2 (a)) demonstrates how the programmer is often educated to think about this apparatus–"the virtual outlook". Here the program counter is the one to move; it moves from one location of the memory to another, perhaps like a "book analogy", where the finger of a reader advances from one line of the book to another. The fact that this von-Neumann apparatus has survived orders of magnitude improvements in speed since the 1940s makes it a remarkable "Darwinistic success story". For this reason we sought to upgrade, rather than replace in a disruptive manner, this successful apparatus.

The lower part of Figure 2 (a), entitled "XMT", illustrates the new apparatus. The left hand side (of the lower part of Figure 2 (a)) depicts the virtual description. There is still one computer program as in the von-Neumann apparatus. In the above book analogy, one finger (marked as PC, for program counter) moves from one line of the book to another, until it reaches a special command called Spawn. The Spawn command specifies a number of "threads" which can

be performed in parallel. Since we discuss now the virtual side, any number of threads can be specified. Figure 2 (a) mentions 1000000 threads. The virtual threads, initiated by a Spawn and terminated by a Join, share the same code. At run-time, different threads may have different lengths, based on individual control flow decisions. The programmer's understanding will be that each of the threads can progress (guided by one finger per thread) from the Spawn command to a subsequent Join command at its own speed. At the Join, the thread expires. Once all the virtual threads expire, finger marked PC continues. The main difference in the hardware description on the right hand side (of the lower part of Figure 2 (a)), is that the number of program counters is fixed (the figure mentions 1000), and does not change as a function of the Spawn command at hand. The program counter of the MTCU (denoted MPC) executes the serial code, prior to the Spawn command. The program counter of the MTCU executes a Spawn command and then broadcasts the following instructions until a Join instruction to the other program counters. The program counters start by executing the first 1000 among the 1000000 threads, one thread each. When a program counter completes its thread, it starts executing one of the yet-to-be-executed threads. This is done until all of the 1000000 threads finish.

Figure 2 (b) illustrates the program of a TCU. Suppose that n = 1000000 threads are to be executed as a result of a Spawn command. The figure assumes that n, and the SPMD code, were broadcast to all TCUs. TCU i starts by executing the respective virtual thread i, but only if i is not larger than n. Upon finishing the execution of a virtual thread, the TCU uses a prefix-sum computation to obtain the ID of the next virtual thread it should execute, and proceeds to execute it if that ID is not larger than n. Note that the only communication among TCUs above was through the prefix-sum computation. An extension of the architecture that allows some nesting of spawn commands (using an sspawn command, noted later) is not reviewed here.

3. MICROARCHITECTURE OF THE PROTOTYPE

3.1 Overview of the prototype

The prototype includes a master thread control unit (MTCU), 4 clusters comprising TCUs and functional units, an interconnection network, 8 on-chip cache modules, a memory controller (MC), a global register file (GRF) and a prefix-sum unit. Figure 3 depicts the block diagram of the XMT FPGA prototype as well as the partitioning for 3 FPGA chips.

The master TCU (MTCU) executes the serial portion of the program and handles the special XMT instructions such as *spawn* and *join* instruction. The MTCU broadcasts the instructions in a parallel section to all clusters where they are copied to a local instruction buffer (see figure 4) and later fetched by TCUs inside clusters. The Master TCU has its own cache, L0, that is only active during serial mode and applies write through. When the XMT processor enters the parallel mode, the Master TCU discards its local cache. The overhead of the flushing L0 cache is trivial since the write through mechanism is chosen. When XMT operates in serial mode, L0 cache is the first level cache of the MTCU and parallel memory modules provide the next level of the memory

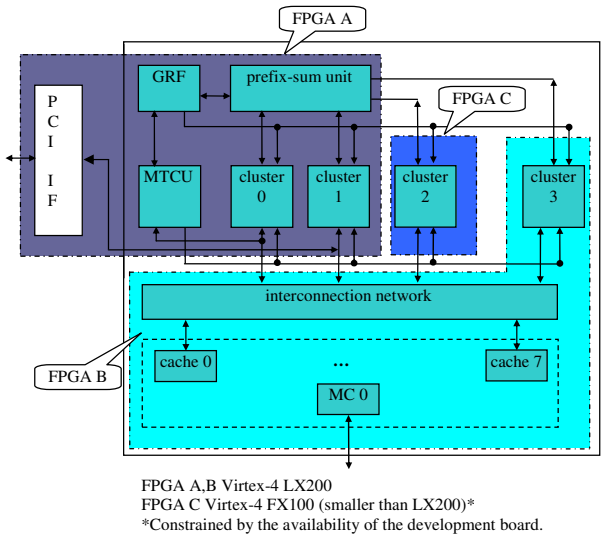


Figure 3: Block diagram of the prototype

hierarchy, which is similar to a multilevel cache hierarchy in an advanced uniprocessor.

A cluster is a group of 16 TCUs and accompanying functional units. The block diagram of a cluster is shown on figure 4. A TCU can execute a thread in the parallel mode. TCUs have their own local registers and they are simple in-order pipelines including fetch, decode execute/memory access and write back stages. The TCUs have a very simple structure and do not aggressively pursue optimal performance. Given the limited chip area, the overall performance of the XMT is likely better when it has a larger number of simple TCUs than fewer but more advanced TCUs, because of the well known diminishing return of many instruction level parallelism (ILP) techniques. However, the XMT concept does not prevent TCUs from introducing any advanced techniques, since the thread level parallelism (TLP) that XMT is taking advantage of is orthogonal to ILP. Similar to a simultaneous multithreaded (SMT) processor, TCUs share some functional units: Multiplication/Division (M/D) unit, read-only buffer and interconnection network port. If several TCUs assigned to a functional unit seek to use it, proper arbitration is used to queue all requests. The read-only buffers in the clusters are hardware/software co-managed temporal storage for TCUs and we will discuss this in the next section. The cluster has one load/store port to the interconnection network, which is shared by all TCUs inside the cluster. The store counter is used to flush the store operations by counting the number of pending stores.

As a key feature of the XMT processor, prefix-sum operations must be executed very efficiently. The hardware implementation of the prefix-sum unit [26] can accept binary input from multiple TCUs and the execution time does not depend on the number of TCUs that are sending requests to it. PS_TCU module in the cluster combines all requests from TCUs and sends one request to the global prefix-sum unit. It is also responsible for distributing the results from the prefix-sum unit to the individual TCUs.

There are 8 independent shared cache modules and they are connected to clusters by an interconnection network. The address space is evenly divided among these cache mod-

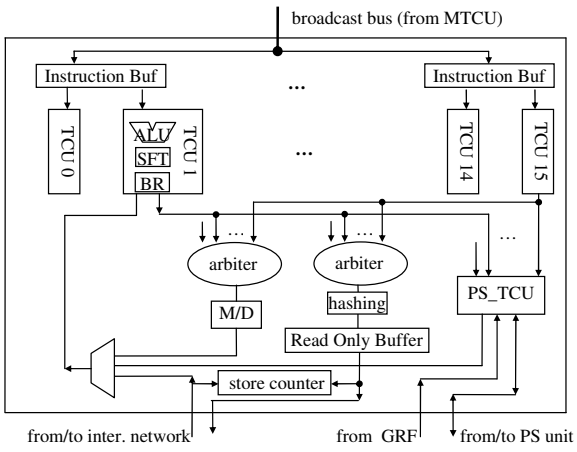


Figure 4: Inside a cluster

ules. To avoid an unbalanced load in different cache modules for certain patterns of memory access, hashing is used in mapping between memory address and cache modules. The parallel caches are used primarily for data, since the instructions for regular TCUs are broadcasted by the MTCU and stored in the instruction buffer.

The interconnection network is a very important component of the XMT processor and needs to provide high bandwidth low latency communication between clusters and cache modules. In this work, we incorporated a behavioral model of the interconnection network presented in [5]. (A recent refinement of the interconnection network that provides improved scalability is presented in [6].) For the prototype, since the number of ports is quite small(4×8), FPGA tools were able to generate an acceptable physical network from the RTL description.

3.2 Features of the memory hierarchy in XMT

We chose not to deploy private caches in TCUs/clusters because of the following reasons. Scalable cache coherence protocols are very complicated for hardware implementation[8] and inefficient for certain types of memory access patterns, typically from fine-grained parallelism. For example, false cache line sharing, which occurs when processors write to a shared cache line but not at the same location, is very inefficient in a cache coherent multiprocessor system. For the fine-grained parallelism, the cache coherent private cache is also not efficient in terms of power, due to the large granularity of the data movement between caches, extra cache coherence message exchange and complicated hardware. The con of our approach is the relative long latency in memory accesses that require round trip to shared parallel cache through an interconnection network. The concept of the length of the sequence of round trips to memory (LSRTM) [27] accounts for the long memory access latency for the performance of an algorithm. The task of optimizing the LSRTM is shared by the programmer, compiler and XMT hardware architecture. In this section, we will present the architecture part of optimization that helps in improving LSRTM. The optimization is based on the observation that the hardware/software co-managed prefetch buffer and the read-only buffer can also take advantage of spatial and temporal locality in a way similar to a local private cache.

Value broadcasting

Consider the following implementation problem in a parallel algorithm. Suppose that all, or nearly all, parallel threads of an XMT program use a certain variable. Without giving special attention to this case, each thread (or at least the first thread in each cluster) will need to read the variable through the interconnection network. Furthermore, the read requests will be queued at the memory module and handled one at a time, thus significantly increasing the implementation overhead of a parallel algorithm. A value broadcasting mechanism is introduced to reduce the implementation time of such concurrent reads. For value broadcasting, the MTCU reads the value from memory in serial mode and stores it to a register, then the value is broadcasted to the TCUs in the form of a load-immediate instruction during instruction broadcasting. Each TCUs can now derive the value from the load-immediate instruction, as opposed to issuing a read request to the shared cache.

Prefetching

In the XMT FPGA prototype, there is no local private cache for TCUs/clusters, but each TCU has 4 prefetch buffers that are controlled by prefetch instructions. A prefetch instruction is essentially a non-blocking read instruction, which will bring the data to one of the prefetch buffers in the TCU. The following read operation for the same memory location will get the value from the prefetch buffer. When a program exhibits a certain memory access pattern that can be determined in compiling time, proper prefetch instruction can be inserted and memory access operation will be overlapped with the execution of other instructions. Therefore, the XMT system can also take advantage of spatial locality in the memory access as a local cache can do. Compared to the conventional cache, a prefetch instruction adds extra overhead to the processor, but it can be more effective in using crucial interconnection bandwidth, because it brings in only necessary data.

Read-only buffer

The TCUs in the same cluster share one interconnection port. When multiple TCUs try to read the same memory location, it is desirable to send one combined request from a cluster to the cache module to reduce load on both the network and cache module. Broadcasting may help solving this problem, but because the broadcasting is done through registers, the number of values that can be broadcasted is limited. Read-only buffer is a hardware/software co-managed storage in the cluster, which provides request combination as well as limited temporal locality. The compiler is supposed to use two different kinds of read instructions, depending on whether the data is safe to be stored in the read-only buffer. In general, any memory locations not written in a particular spawn-join block, which may be referenced multiple times from TCUs, are good candidates for storing in the read-only buffer. The read-only buffer stores the value as well as the address and the following read operations that reference the same address will get the value from the read-only buffer.

3.3 Specifications of the XMT FPGA prototype

The XMT FPGA prototype system consists of 3 FPGA chips: 2 Virtex-4 LX200 and 1 Virtex-4 FX100. PCI is used as the interface. The FPGA board is purchased from a

Table 1: specifications of the XMT FPGA prototype

Clock rate	75MHz	Number of TCU cluster	4
Memory size	1GB DDR2	Number of TCU per cluster	16
DRAM. data rate	2.4GB/s	Number of shared cache modules	8
MTCU local cache	8KB	Size of each shared cache module	32KB
Shared cache miss penalty	26∼ cycles	MTCU mem. access local miss, shared cache hit	25 cycle
TCU shared cache access hit	31 cycles	MTCU mem. access local hit	1 cycle
TCU ps operation	10∼25 cycles	MTCU,TCU ALU operation	1 cycles
MTCU,TCU SHIFT operation	2 cycles	MTCU,TCU BRANCH penalty	4 cycles
MTCU multiplication	6 cycles	MTCU division	36 cycles
TCU multiplication, division sharing overhead	4 cycles	Number of multiplication/division unit per cluster	1
Number of ALU,BRANCH,SHIFT unit per cluster	16	Size of each instruction buffer in TCUs	4KB

third party company and our selection is constrained by the availability of development board that can be used for XMT prototyping. Detailed specifications of the XMT FPGA prototype system are listed in table 1.

3.4 Envisioned XMT processor

The XMT FPGA prototype is a scaled-down version of an envisioned XMT processor, which is shown in figure 5. We aspire to have in the not-too-far future a XMT processor that has 1024 TCUs grouped into 64 clusters and 64 on-chip memory modules. Each memory module consists of two levels of caches and multiple memory access ports shared by multiple L2 cache modules. The MTCU has local instruction and data caches for better backwards compatibility with serial programs.

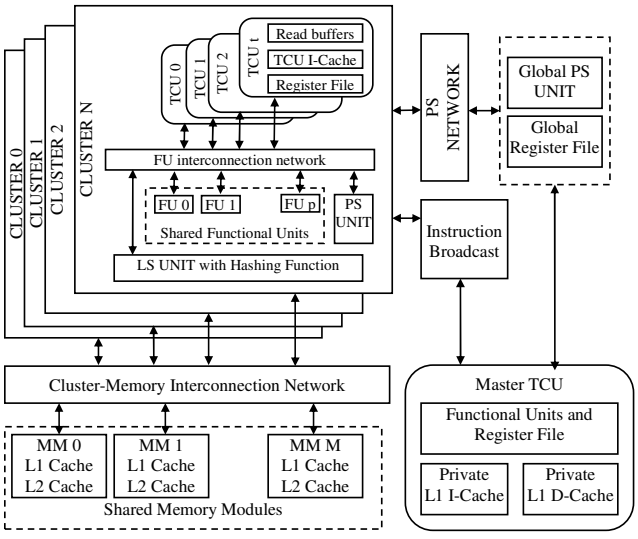


Figure 5: An envisioned XMT processor

The XMT FPGA prototype supports a subset of MIPS I ISA as well as a few XMT specific instructions. Most of MIPS I instructions, except for floating point, byte and halfword load-store instructions, are supported. The XMT specific instructions include spawn, join, sspawn (for single spawn: generate an additional thread while in parallel mode), ps, psm, and instructions for broadcasting, prefetching, and read-only buffers.

4. PERFORMANCE EVALUATION OF THE XMT FPGA PROTOTYPE

4.1 Kernel benchmarks

The prototype is not a full system and the applications we can test on it are quite limited. First, it does not support floating point operations. This puts many real applications and benchmarks, such as SPLASH, out of consideration. Another limitation of the current prototype is that the parallel section cannot have any function call, because TCUs do not have an instruction cache and the size of the instruction buffer is quite limited. A third and more important limitation is that, the prototype is not ready for an OS. These limitations are only for this prototype and will be eliminated in the future. Then we will be able to test the system with broader applications. With the above limitations, the following 8 kernel applications are chosen to benchmark the performance of the prototype. Fortunately, with integer programs, we are able to test the performance of the memory system, which is the interesting part of the XMT architecture. The input sizes are listed in table 2. For each kernel, we tested 2 input sizes, "large" and "small". The memory size used by each program is also listed.

The 8 kernel benchmarks are matrix multiplication(mmul), quicksort(qsort), breadth-first search(BFS), finding longest path in a directed acyclic graph(DAG), array summation(add), array compaction(comp), key search in binary search tree (BST) and convolution(conv). mmul is to calculate the product of two dense integer matrices. A known parallel quicksort method [15] was used for XMT. BFS and DAG are representative graph applications and the memory access pattern is irregular and known to be difficult to parallelize. Array summation is calculating the sum of an integer array and array compaction is described in Section 2.1. The BST problem is searching for a set of keys in a balanced binary search tree. In conv, a two dimensional image array of X , and another filter array of F are given and a filtered array Y is calculated with the dot product of F and sub-array of X.

As explained above, the compiler is responsible for taking

Table 2: input size of the benchmarks

App.	Large size			Small size		
	input size	memory usage		input size	memory usage	
		parallel	serial		parallel	serial
mmul	2000x2000	48MB	48MB	128x128	192KB	192KB
qsort	20 million	360MB	200MB	100 thousand	1.8MB	1MB
BFS	V=1M, E=10M	220MB	100MB	V=100K, E=1M	21.6MB	9.6MB
DAG	V=1M, E=17M	368MB	160MB	V=50K, E=600K	13.4MB	6.0MB
add	50 million	200MB	200MB	3 million	12MB	12MB
comp	20 million	208MB	208MB	2 million	20.8MB	20.8MB
BST	16.8M nodes, 512K keys	205MB	205MB	2.1M nodes, 16K keys	25.3MB	25.3MB
conv	image:1000x1000 filter:32x32	8MB	8MB	image:200x200 filter:16x16	320KB	320KB

advantage of the 3 features of the XMT FPGA prototype: value broadcasting, hardware/software co-managed prefetch buffer, and read-only buffer. Our compiler is under development and is not yet mature enough to effectively use these features, especially the last two. To test XMT FPGA prototype properly, we manually optimized the compiler produced assembly code. (i) For read operations in a loop, a prefetch instruction for the next iteration is inserted for each read operation. (ii)For other read operations, prefetch instructions are inserted as soon as the address is available. (iii) Normal read instructions are replaced with XMT-specific read instructions that make use of the read-only buffers, if the value could be guaranteed not to change during the current parallel mode. These manual optimizations will be integrated into the XMTC compiler soon.

4.2 Performance evaluation

In this section, we will report some data collected from the XMT FPGA prototype.

Speedup of parallel program in XMT

Both serial and parallel versions of the 8 benchmarks are executed on the FPGA system. The speedups of parallel over serial programs are shown in table 3. Generally, speedup is upper bounded by 64, since at most 64 threads are active in the parallel programs.

Two computational benchmarks, mmul and conv, achieved the highest speedups. This is because the program is highly CPU-bound and both have very regular memory access patterns, which makes it easy to take advantage of the software prefetch command. On the other hand, quicksort, addition and compaction problems are memory-bound programs and provided lower speedups. Note that, while the FPGA computer has 64 TCUs, they are grouped in 4 clusters and all 16 TCUs inside a cluster share one load/store port of the cluster. On the cache side, there are 8 parallel cache modules, so on average, 8 TCUs share one cache module. More importantly, the FPGA computer has only one off-chip DRAM channel shared by all 8 on-chip cache modules. For memory bound applications, these shared resource can become a bottleneck and limits the speedup numbers. The two graph related applications, BFS and DAG, are fine-grained and their memory access patterns are irregular. The XMT FPGA has speedups of 15.7∼22.0 for these two benchmarks. Note that these two programs are known to be very difficult to parallelize for traditional coarse grain parallel computers.

The BST program showed relatively low speedup of 7.00 in the large input set and 10.0 in the small input set. Recall that the BST is a balanced binary tree and a search proceeds from the root of the tree and advances towards leaves until the key is found or a leaf is reached. The upper part of the tree, especially the root, is repeatedly accessed by all threads and this will add queuing delay to the TCUs. Unlike the local cache in the MTCU, which is used only by one thread, the read-only buffer is shared by all 16 TCUs in a cluster, meaning 512 bytes per thread, which is extremely small to take advantage of temporal locality. The increased speedup in the small input set confirms the above explanation. In the program data structure the pointer to the left child is always following the parent node. In the serial program, the local cache in MTCU is taking advantage of this spatial locality automatically. However, in the parallel program, explicit prefetch commands need to be inserted, but we did not do it, because we are not sure yet that the compiler can figure out this locality.

Cache hit rate

The size of the cache in the prototype is small, 8KB local cache for the MTCU and a total of 256KB parallel cache for the TCUs or only 4KB per TCU. This is due to the limited "block RAMs", the built-in memory in Xilinx FPGAs. The cache hit rates of the large input size are listed in table 4. Cache hit rate is calculated for read operation only(write is ignored). In serial execution, cache hit rate of local cache is reported while that of parallel shared cache is reported for parallel execution. Table 4 shows that the hit rate in parallel execution is lower than that of serial execution in general and it is very low for BFS, DAG, qsort and BST. This suggests that the cache size may need to be increased in future XMT processors.

Throughputs of the load/store unit and interconnection network

XMT is a shared memory architecture and all TCUs access the shared memory space through the interconnection network. It is interesting to study how the interconnection network is used in these benchmarks. Grouping 16 TCUs into a cluster and assigning one interconnection network port to a cluster lead to efficient utilization of the interconnection network. Note that the cost of the interconnection network in terms of area, and to a lesser extent delay, is increasing significantly as the number of ports increases. For all bench-

Table 3: Speedups of the benchmarks (parallel Vs. serial in XMT)

Input	mmul	qsort	BFS	DAG	add	comp	BST	conv
Large	35.7	20.8	15.7	19.6	26.0	15.3	7.00	38.9
Small	45.9	16.8	18.1	22.0	24.0	23.8	10.0	36.9

Table 4: Cache hit rate

execution	mmul	qsort	BFS	DAG	add	comp	BST	conv
serial	0.688	0.953	0.681	0.587	0.874	0.993	0.794	0.918
parallel	0.75	0.57	0.31	0.50	0.88	0.78	0.43	0.99

marks, we calculated the total number of packets delivered by the interconnection network as well as the number of requests processed by the load store (LS) units inside the clusters. Two factors cause the LS unit and interconnection network to have different utilization rates. The LS unit will generate two packets for a write/PSM and thus results in a higher load in the interconnection network. On the other hand, some requests from TCUs can be processed by the read-only buffer without ever sending a packet over the interconnection network; this would reduce the relative load of the interconnection network. The number of packets that will be sent and received through the interconnection network is listed in table 5. A DRAM prefetch instruction from a TCU brings the data from DRAM to the shared cache, but the TCU will not receive any response.

Table 5: Number of packets per request

Request	send	receive
Read	1	1
Write/PSM	2	1
DRAM prefetch	1	0

The bandwidth utilization of the interconnection network and normalized throughput of the LS unit are shown in Figure 6. The number of total requests processed by the LS units is divided by the total number of the parallel cycles and the number of LS units in the XMT processor, which is 4 in this prototype. The total number of packets transferred from clusters to the cache modules are divided by the number of cycles in parallel mode and the number of clusters, 4.

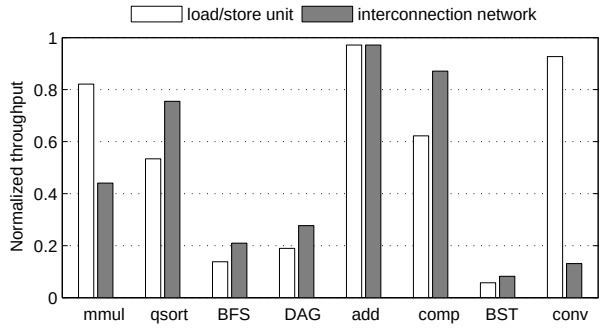


Figure 6: Normalized throughput

Four benchmarks: mmul, qsort, add and comp showed a high utilization rate of LS unit or interconnection network usage. For mmul and conv, where the read-only buffer is extensively used, the LS unit is almost fully loaded but the network has much less traffic. For other benchmarks, where the read-only buffer is not used extensively, the interconnection network is more crowded than the LS units, because an LS unit uses one cycle to process a write operation that will then require two packets(address and data), and, therefore two cycles in the interconnection network. The 3 benchmarks BFS, DAG and BST used a very small portion of the bandwidth available. This is because of the long latency in cache accesses due to the low hit rate and extensively long queue in the cache module (BST).

4.3 Performance of an envisioned XMT ASIC processor

In previous sections, various aspects of the performance of the XMT FPGA computer are measured and evaluated, but we are not ready to compare the wall clock execution time of the benchmarks between XMT FPGA prototype and any existing processors because the clock rate of the XMT FPGA prototype is too low. It is clear that XMT processor with ASIC implementation can operate at a much higher clock rate, and thus achieve much better performance in terms of wall clock time. In this section, performance of an arbitrary XMT ASIC processor with a higher, yet quite modest, clock rate is projected and compared against an AMD Opteron processor.

An arbitrary XMT processor with 800MHz internal clock, and 400MHz DDR2 DRAM memory is chosen for the performance evaluation of a XMT ASIC processor. Both clock rates are reasonable and are in fact a bit conservative, considering the fact that 400MHz DDR2 DRAM (PC2-6400) is commercially available and MIPS32® $74K^{TM}$ family cores operate at 1GHz.

If all components of the XMT FPGA computer are accelerated in the same ratio, the cycle count will not change and the wall clock time will decrease at the same ratio and we can easily project the performance of the XMT processor with a higher clock rate. However, due to the dynamic behavior of the DRAMs and some timing requirements, the cycle count of the DRAM operations will change dramatically in different clock rate thus this simple projection method cannot be used.

To enable the projection, we need to design a low clock rate system that behaves exactly as in the high clock rate system in terms of cycle count. In other words, if the low

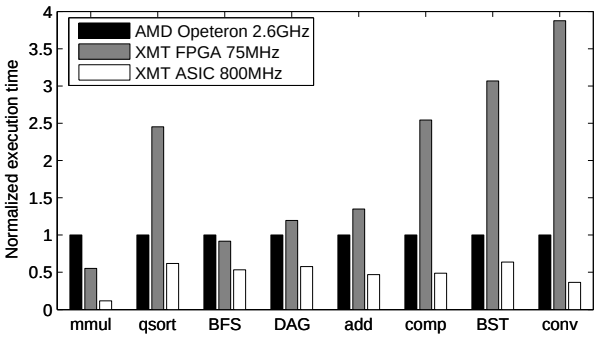


Figure 7: Normalized execution time

clock rate system is cycle accurate for the high clock system, the cycle count can be used for evaluating the high clock rate system. The following 3 limitations are applied to get a cycle accurate system: (i) The timing constraints are converted to the number of cycles in 800MHz and proper delays are added. (ii) The data transfer bandwidth is constrained to one burst transfer per four cycles as in the 800Mhz DDR2 DRAM chip. (iii) The DDR2 command rate is also limited to one command per two cycles as in the 800Mhz DDR2 DRAM chip. Table 6 shows what is modified to emulate 800MHz XMT ASIC properly.

Table 6: Modifications for performance projection

Item	75MHz	75MHz emulating 800MHz	800MHz emulated
Read latency [a] (cycle)	3.5 [b]	24.5	24.5
Maximum DRAM command per cycle	2	0.5	0.5
Peak bandwidth	2.4GB/s	0.6GB/s	6.4GB/s

[a]Latency of DRAM access depends on many factors. We only noted latency of a read operation, under some DRAM assumptions. For those familiar with DRAMs and DRAM terminology, the assumptions are that the reading is done from a closed bank and there are no activities in other banks.

[b]The DRAM controller operates at 150MHz and one cycle in 150MHz is converted to half a cycle in 75MHz

The execution time is calculated by converting the cycle count with the period of the 800MHz clock or dividing the wall clock time by ratio of 800/75=10.67. Figure 7 shows normalized wall clock time of the 8 kernel benchmarks in an AMD Opteron 2.6GHz, XMT FPGA 75MHz and envisioned XMT ASIC 800MHz. All wall clock time is normalized to the execution time of an AMD Opteron 2.6GHz. The execution time measured in seconds are listed in the Table 7. The AMD Opteron processor is operating at 2.6GHz and has 64KB+64KB L1, 1MB L2 cache. The system used for testing had dual channel PC-3200 DDR DRAM, which provides a bandwidth of 6.4GB/s. A 64-TCU XMT processor has been implemented in a $10mm \times 10mm$ chip in 90nm technology. As this was an academic project by a team of only four students and with rather limited optimization, this area usage should be interpreted as an upper bound on the area needed. Unfortunately, we were not able to find the area information about the tested 2.6GHz AMD Opteron, but a similar configuration(same cache size) AMD Opteron used 189 mm^2 in 130nm technology[12]. Although it is not fair to compare the two numbers directly, it is reasonable to believe that the XMT ASIC uses no more silicon area than the AMD Opteron processor tested. The envisioned XMT ASIC 800MHz system outperforms an AMD Opteron for all of 8 kernel benchmarks and speedup ranges from 1.57 to 8.56. It is quite significant considering that the XMT ASIC 800MHz processor has only a total of 256 KB shared cache and operating at a much lower clock rate. The envisioned XMT ASIC 800MHz system showed significant advantage over the AMD Opteron in two CPU-bound benchmarks, mmul and conv.

5. RELATED WORK

The aspiration to approximate the theoretical performance of the PRAM is not new. Multi-chip multi-processor designs such as the NYU-Ultracomputer[11], CRAY/Tera MTA[1] and the SB-PRAM [2, 9] are representative examples. Although XMT shares some features with these 3 architectures, XMT is different from them as it is a single chip architecture. The Tera MTA and SB-PRAM execute multiple threads on a processor, but they accomplish that by switching among many threads, rather than executing multiple threads concurrently. The fetch-and-add feature in the NYU-Ultracomputer and SB-PRAM is integrated into the network, while the prefix-sum operation in XMT is a dedicated fast single functional unit.

CMP (chip multiprocessors) [14], where multiple processor cores are placed on a single die, is a popular form of on-chip parallel architecture. IBM introduced dual core Power 4 processor in 2001, now both Intel and AMD offer multicore processor products. These multicore processors [14, 19] are good for coarse-grain parallelism, such as multitasking. However, they are limited in supporting extremely fine-grained programs, like PRAM algorithms, efficiently because of the private cache in each core.

The CELL processor [17], jointly developed by Sony, Toshiba and IBM, is another interesting form of multicore processor. The explicit software-controlled memory architecture in the CELL processor poses significant challenges for programmers for both correctness and performance. The XMT processor seeks a middle ground by introducing hardware/software co-managed read-only buffer and prefetch buffer. The Niagara processor from SUN[10] and XMT share some features, like using simple in-order processor for better power and area efficiency, but Niagara is trying to achieve better overall throughput from a single chip, while XMT seeks the classic goal of shortening single task completion time using parallel computing.

Tile-based architectures, such as MIT's Raw, Stanford's Smart Memories and UT-Austin's TRIPS [22], also expect to scale to high levels of parallelism. XMT, unlike Raw, Smart Memories and TRIPS, provides hardware support for efficient load balancing, and better support for a shared-memory model, both of which are critical for many irregular applications. Last, but not least, none of these approaches, as well as transactional memories and general purpose use of GPUs (with their remarkable progress), can currently provide evidence that a general-purpose parallel algorith-

Table 7: Execution time in seconds

processor	mmul	qsort	BFS	DAG	add	comp	BST	conv
Opteron	117.4	2.644	0.659	2.594	0.143	0.105	0.479	1.776
XMT FPGA	64.74	6.483	0.604	3.101	0.193	0.267	1.469	6.884
XMT ASIC	13.71	1.634	0.351	1.494	0.067	0.051	0.305	0.647

mic knowledge base exists to support them, while XMT was engineered to support the PRAM.

6. CONCLUSION AND DISCUSSION

First commitment to silicon of XMT

In this paper, we presented the first commitment to silicon of the XMT architecture. This is a significant milestone and we are one step closer to a practical PRAM-on-chip processor. In addition to the brief description of the micro-architecture of the prototype, 3 features: value broadcasting, hardware/software co-managed prefetch buffer and read-only buffer are discussed. The performance of the XMT FPGA prototype is evaluated and performance of an envisioned XMT ASIC processor is projected based on cycle accurate emulation. With the same or less area budget, the 800MHz XMT ASIC processor outperforms the 2.6GHz AMD Opteron. Overall, the XMT processor is not only easy to program, but also provides very good performance in addition to very high area efficiency.

Timely Case for the Education Enterprise

We believe the education enterprise should have a special interest in our approach. As explained throughout [8], other parallel programming approaches tend to require understanding that does not only comprise the PRAM level of understanding (or cognition), but in fact, significantly exceeds it. This means that the PRAM provides a useful *common denominator* among the variety of current approaches. Our impression is that most computer science undergraduate programs maintain the old status quo, where they do not teach parallel programming and algorithms partly because the jury is still out on which approaches will emerge as winners. The unfortunate outcome is that the sole training that a 22-year graduate receives, heading for a 50 year career that is likely to be dominated by parallelism, is for programming the computers of the past. Each year that goes by without change, *yet another generation gets out of school and into the market place underprepared*, or perhaps even mis-prepared; the reason is that, for some, the biggest challenge in parallel algorithms and programming education is overcoming bad habits. If it were possible to widely distribute affordable PRAM-On-Chip machines, the education enterprise would be able to make significant progress towards mitigating the problem. As explained earlier, we believe that the change should also encompass high schools, and eventually even middle schools, or wherever young people learn their first programming language. The fact that PRAM is based on first principles makes this possible.

XMT is a Candidate for the Processor-of-the-Future

Based on the roadmap of all vendors, the processor of the future will have to be based on a general-purpose on-chip parallel computer architecture. There are some crucial properties that any candidate architecture must have. Below, we highlight several such properties, and explain, or provide evidence that XMT meets them. As noted earlier, our focus is on the market niche that aims at single-task completion time. *Property 1: ease of programming.* With few exceptions, novel parallel computer systems started with an architecture, and this led to methodologies for programming them. XMT did not follow this *build-first figure-out-how-to-program-later approach*, as its starting point was the simple PRAM parallel algorithmic thinking. In PRAM algorithmic model, the algorithm designer (programmer) is free to assume that the hardware resources (number of hardware threads) are not limited; and can create an arbitrary number of concurrent virtual threads. This provides a highly sought freedom for the programmer, as hardware specifics are abstracted away. This can be done since the hardware employs an efficient mechanism of dynamically allocating these virtual threads to available hardware threads. This freedom to express concurrency and its automatic translation to hardware provide ease of programming, and high performance. Coupled with the unmatched theoretical strength of the PRAM, this gives XMT a significant advantage. The 2005 NSF Blue-Ribbon Panel on Cyberinfrastructure reported that to many users programming existing parallel computers is as intimidating and time-consuming as programming in assembly language. To complete our discussion of ease-of-programming, we note some recent empirical evidence. Experimental validation of ease-of-programming claims requires a big investment, or otherwise it will not meet academic publication standards. The DARPA-HPCS program provided such an investment. The study reported in the journal paper [16] compared two similar groups of students solving the same problem. One group used MPI. The second group that used XMTC was able to complete the job in about half the development time of the MPI group. *Property 2: good performance with any amount, or grain, of parallelism provided by the algorithm; namely, up-and down-scalability including backwards compatibility on serial code, and fine-grained or coarse-grained parallelism.* The only issue not emphasized previously is compatibility on serial code, as provided by the MTCU and its local cache. The importance of backwards compatibility on serial code cannot be overstated, as this allows use of existing code. *Property 3: support application programming (in standard application languages, such as VHDL/Verilog, OpenGL, MATLAB).* Since code in such application languages tend to reflect parallelism allowed in the application, it is often possible to translate this parallelism into significant speedups on XMT. The paper [13] demonstrated that XMT can provide speedups of 100X given for gate-level logic simulation using VHDL. *Property 4: fit current chip technology and scales with it.* Tape-out of a 9mm by 5mm chip in 90nm comprising the interconnection network component of a 128-TCU XMT was reported in [4]. We aspire to have in the not-too-far future an XMT processor that has 1024 TCUs grouped into 64 clusters and 64 on-chip memory modules.

What one needs to learn about the XMT approach in order to start using it

We see several levels of use. The entry (or *literacy*) level requires some minimal background in C or Java programming and reading the tutorial and manual of XMT [3]. The next level, which we call *fluency* level, needs to recognize, though to a rather a limited extent, that the original intellectual basis for XMT was the PRAM theory of parallel algorithms. Below, we explain the relationship between that theory and XMT programming.

We will make the point that one does not need to learn the PRAM theory in order to use XMT. In fact, as noted earlier, we have demonstrated that most XMT programmers can generally detour the PRAM theory.

The *first thing* that one needs to learn is how the PRAM teaches to think algorithmically in parallel. This is the only PRAM-related thing one really needs to understand. PRAM algorithm description requires almost a computer-program-like level-of-detail. Having to deal upfront with lower level details is not only tedious; it often becomes an obstacle to concentrating on the big picture. To overcome that, a description methodology that allows drastic reduction in level-of-detail is needed. This methodology guides thinking about a parallel algorithm in terms of two basic features: its total number of operations (called work), and the shortest time (called depth) in which these operations could be completed, hypothetically assuming unlimited amount of hardware. Introduced by Shiloach and Vishkin in 1982 [23], what makes the methodology work is that it is a matter of skill to later fill in the details that the Work-Depth description teaches to initially suppress. Furthermore, limited training is sufficient to provide the needed skill. (It is worth noting that this work-depth methodology provides the description platform for several parallel algorithms books that appeared since the early 1990s, such as [20, 18].)

The *second thing* is how to turn this parallel algorithmic thinking into XMTC programming. One of our more recent innovations (demonstrated with high school students and through a university course offered to Freshmen that do not major in computer science) is that people can proceed from this high-level parallel algorithmic thinking directly to XMTC programming.

For most users, these two things are all they need to know. One quick way to learn this material is through the online tutorial featured on [31]. The items featured include: recorded video of a 300-minute tutorial (originally given to high-school students) along with slides, class notes, as well as a tutorial and manual for XMTC. The basic methodology, from parallel algorithmic thinking to XMTC programming, outlined above is also reviewed in [27].

Expert users (such as performance programmers), or (say compiler) researchers, may then follow-up with an in-depth study of the current paper, or other XMT papers and of course the full PRAM theory.

Postscript

A naming contest for the XMT FPGA prototype held by the University of Maryland got nearly 6000 submissions. The name Paraleap was selected. See [30].

Acknowledgement

Generous help from Aydin Balkan, Rajeev Barua, George Caragea, Mike Horak, Fuat Keceli, Gang Qu and Alexandros Tzannes, as well as from other members of the XMT team is all gratefully acknowledged. Specific major contributions include a compiler developed by Alexandros [24], and the cycle accurate simulator developed by Fuat.

7. REFERENCES

[1] R. Alverson, D. Callahan, D. Cummings, B. Koblenz, A. Porterfield, and B. Smith. The tera computer system. In *Proceedings of the 4th international conference on Supercomputing*, 1990.

[2] P. Bach, M. Braun, A. Formella, J. Friedrich, T. Grun, and C. Lichtenau. Building the 4 processor SB-PRAM prototype. In *Proceedings of the 30th Hawaii International Conference on System Sciences: Advanced Technology Track - Volume 5*, 1997.

[3] A. O. Balkan and U. Vishkin. Programmer's manual for XMTC language, XMTC compiler and XMT simulator. Technical report, University of Maryland Institute for Advanced Compuer Studies, 2005.

[4] A. O. Balkan, M. Horak, G. Qu, and U. Vishkin. Layout-accurate design and implementation of a high-throughput interconnection network for single-chip parallel processing. In *Hot Interconnects, Stanford, CA*, 2007.

[5] A. O. Balkan, G. Qu, and U. Vishkin. A mesh-of-trees interconnection network for single-chip parallel processing. In *ASAP '06: Proceedings of the IEEE 17th International Conference on Application-specific Systems, Architectures and Processors* , pages 73–80, 2006.

[6] A. O. Balkan, G. Qu, and U. Vishkin. An Area-Efficient High-Throughput Hybrid Interconnection Network for Single-Chip Parallel Processing. In *45th Design Automation Conference*, Anaheim, CA, June 8-13, 2008. To appear.

[7] D. E. Culler, R. M. Karp, D. A. Patterson, A. Sahay, K. E. Schauser, E. Santos, R. Subramonian, and T. von Eicken. LogP: Towards a realistic model of parallel computation. In *Principles Practice of Parallel Programming*, pages 1–12, 1993.

[8] D. E. Culler, J. P. Singh, and A. Gupta. *Parallel Computer Architecture: A Hardware/Software Approach.* Morgan Kaufmann Publishers, Inc, 1998.

[9] A. Formella, J. Keller, and T. Walle. Hpp: A high performance PRAM. In *Euro-Par '96: Proceedings of the Second International Euro-Par Conference on Parallel Processing-Volume II*, pages 425–434, London, UK, 1996. Springer-Verlag.

[10] L. Geppert. Sun's big splash niagara microprocessor chip. *IEEE Spectrum*, 42:56–60, 2005.

[11] A. Gottlieb, R. Grishman, C. P. Kruskal, K. P. McAuliffe, L. Rudolph, and M. Snir. The NYU Ultracomputer - designing an mimd shared memory parallel computer. *IEEE Trans. Computers*, 32(2):175–189, 1983.

[12] M. Gotuaco, P. Huebler, H. Ruelke, C. Streck, and W. Senninger. Implementation of cvd low-k dielectrics for high-volume production. *Solid State Technology*, 47:60–62, 2004.

[13] P. Gu and U. Vishkin. Case study of gate-level logic simulation on an extremely fine-grained chip multiprocesor. *Journal of Embedded Computing,*

Special Issue: Issues in Embedded Single-Chip Multicore Architectures, 2,2:181–190, 2006.

[14] L. Hammond, B. A. Nayfeh, and K. Olukotun. A single-chip multiprocessor. *Computer*, 30(9):79–85, 1997.

[15] P. Heidelberger, A. Norton, and J. T. Robinson. Parallel quicksort using fetch-and-add. *IEEE Trans. Comput.*, 39(1):133–138, 1990.

[16] L. Hochstein, V. Basili, U. Vishkin, and J. Gilbert. A pilot study to compare programming effort for two parallel programming models. *Journal of Systems and Software*, To appear.

[17] H. P. Hofstee. Power efficient processor architecture and the CELL processor. *hpca*, 00:258–262, 2005.

[18] J. JaJa. An Introduction to Parallel Algorithms. Addison-Wesley, 1992.

[19] R. Kalla, B. Sinharoy, and J. M. Tendler. Ibm power5 chip: A dual-core multithreaded processor. *IEEE Micro*, 24(2):40–47, 2004.

[20] J. Keller, C.W. Kessler and J.L. Traeff. Practical PRAM Programming. Wiley-Interscience, 2001.

[21] D. Naishlos, J. Nuzman, C.-W. Tseng, and U. Vishkin. Towards a first vertical prototyping of an extremely fine-grained parallel programming approach. *Theory of Computing Systems, Special Issue for SPAA 2001. Springer*, 36:521–552, 2003.

[22] K. Sankaralingam, R. Nagarajan, H. Liu, C. Kim, J. Huh, D. Burger, S. W. Keckler, and C. R. Moore. Exploiting ILP, TLP, and DLP with the polymorphous TRIPS architecture. *SIGARCH Comput. Archit. News*, 31(2):422–433, 2003.

[23] Y. Shiloach and U. Vishkin. An O((n**2)log n) parallel max-flow algorithm. *J. Algorithms*, 3(2):128–146, 1982.

[24] A. Tzannes, R. Barua, G. Caragea, and U. Vishkin. Issues in writing a parallel compiler starting from a serial compiler, draft. Technical report, University of Maryland Institute for Advanced Compuer Studies, 2006.

[25] U. Vishkin. Spawn-join instruction set architecture for providing explicit multi-threading (XMT). U.S. Patent 6,463,527, 2002.

[26] U. Vishkin. Prefix sums and an application thereoff. U.S. Patent 6,542,918, 2003.

[27] U. Vishkin, G. Caragea, and B. Lee. *Handbook of Parallel Computing: Models, Algorithms and Applications*, chapter Models for advancing PRAM and other algorithms into parallel programs for a PRAM-On-Chip platform. CRC press, 2008.

[28] U. Vishkin, S. Dascal, E. Berkovich, and J. Nuzman. Explicit multi-threading (XMT) bridging models for instruction parallelism. In *Proc. 10th ACM symposium on Parallel algorithms and architectures SPAA*, 1998.

[29] X. Wen and U. Vishkin. Brief announcement – PRAM-On-Chip: first commitment to silicon. In *SPAA '07: Proc. 19th ACM symposium on Parallelism in algorithms and architectures* pages 301-2.

[30] http://www.eng.umd.edu/media /pressreleases/pr112707_superwinner.html

[31] http://www.umiacs.umd.edu/users /vishkin/XMT/index.shtml#tutorial