PRAM-On-Chip: First Commitment to Silicon

Xingzhi Wen and Uzi Vishkin
University of Maryland

Categories and Subject Descriptors: C.1.4 Parallel Architectures.

Terms: Design

Keywords: Parallel algorithms, PRAM, On-chip parallel processor, Ease-of-programming, Explicit multi-threading, XMT

Introduction[1] The *eXplicit Multi-Threading (XMT)* on-chip general-purpose computer architecture is aimed at the classic goal of reducing single task completion time. It 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) a methodology for advancing from PRAM algorithms to XMT programs, along with a performance metric and its empirical validation are provided [1]. 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. This brief announcement (BA) reports first commitment to silicon of XMT. A 64-processor, 75MHz computer based on field-programmable gate array (FPGA) technology was built at the University of Maryland (UMD). XMT was introduced in SPAA'98. An architecture simulator and speed-up results on several kernels were reported in SPAA'01. The new computer is a significant milestone for the broad PRAM-On-Chip project at UMD. In fact, contributions in the current BA include several stages since SPAA'01: 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. This overall progress, its context and uses of the much faster hardware over a simulator are the focus of this BA.

The PRAM virtual model of computation 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 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 performs ("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—a Fetch-and-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*. See also [1]. 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 is unless something (e.g., prefetch) is done to avoid it; *and our LSRTM metric accounts for that.* While we took advantage of Burton Smith's latency hiding pipelining technique for code providing abundant parallelism, the LSRTM metric guided design for good performance from any amount of parallelism, even if it is rather limited. Moving data between MH levels (e.g., main memory to first-level cache) is generally orthogonal and amenable to standard caching approaches. In addition to XMTC many other application-programming interfaces (APIs) will be possible; e.g., VHDL/Verilog, MATLAB, and OpenGL.

A brain child of the SPAA community, 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 had a big chapter on PRAM algorithms. Theorists (UV included) also claimed for many years that the PRAM theory is useful. However, the PRAM was generally deemed useless (e.g., see the 1993 LOGP paper). Since the mid-1990s, PRAM research was reduced to a trickle, most of its researchers left it, and later book editions dropped their PRAM chapter. The 1998 state-of-the-art is reported in Culler-Singh's parallel computer architecture book: "*.. breakthrough may come from architecture if we can truly design a machine that can look to the programmer like a PRAM*". In 2007, we are a step closer as hardware replaces a simulator. The current paper seeks to advance the perception of PRAM implementability *from impossible to available*. The new computer provides freedom and opportunity to pursue PRAM-related research, development and education [2] 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). This *wall clock gap* is itself a significant contribution: heavier programs and applications and larger inputs to study scalability can now be run. If runs of the XMT computer grossed 18 hours for this BA, simulator results would have been ready for SPAA 2030. A programming assignment that ran over an hour on a simulator in the Spring'06 UMD Parallel Algorithms course takes under a second in Spring'07. Also, a general-purpose algorithms course with its *own* computer, programming methodology [1], programming language, and compiler is rather uncommon.

Overview of the XMT Architecture and Computer The XMT processor (see Fig 1) includes a master thread control unit (MTCU), processing clusters (C0...Cn in Fig 1) each comprising several TCUs, a high-bandwidth low-latency interconnection network, memory modules (MMs) each comprising on-chip cache and off-chip memory, a global register file (GRF) and a

[1] {hsmoon,vishkin}@umd.edu.
XMT Home page: www.umiacs.umd.edu/~vishkin/XMT
Partially supported by NSF grant CCF-0325393.

SPAA'07, June 9-11, 2007, San Diego, California, USA.
ACM 978-1-59593-667-7/07/0006.

prefix-sum unit. Fig. 1 suppresses the sharing of a memory controller by several MMs. The processor alternates between serial mode, where only the MTCU is active, and parallel mode. The MTCU has a standard private data cache used only in serial mode and a standard instruction cache. The TCUs do not have a write data cache. They and the MTCU all share the MMs. Our SPAA'01 paper describes the way in which: (i) the XMT apparatus of the program counters and stored program extends the standard von-Neumann serial apparatus, (ii) virtual threads coming from an XMTC program (these are *not OS threads*) are allocated dynamically at run time, for load balancing, to TCUs, (iii) hardware implementation of the PS operation and its coupling with a global register file (GRF), (iv) independence of order semantics (IOS) that allows a thread to advance at its own speed without busy-waiting for other concurrent threads and its tie to Arbitrary CW, and (v) a more general design ideal, called no-busy-wait finite-state-machines (NBW FSM), guides the overall design of XMT. In principle, the MTCU is an advanced serial microprocessor that can also execute XMT instructions such as spawn and join. Typical program execution flow is shown on Fig.2. The MTCU broadcasts the instructions in a parallel section, that starts with a spawn command and ends with a join command, on a bus connecting to all TCU clusters. In parallel mode a TCU can execute one thread at a time. TCUs have their own local registers and they are simple in-order pipelines including fetch, decode, execute/memory-access and write back stages. The new computer has 64 TCUs in 4 clusters of 16 TCUs each. (We aspire to have 1024 TCUs in 64 clusters in the future). A cluster has functional units shared by several TCUs and one load/store port to the interconnection network, shared by all its TCUs. The global memory address space is evenly partitioned into the MMs using a form of hashing. In particular, *the cache-coherence problem, a challenge for scalability, is eliminated*: in principle, there are no local caches at the TCUs. Within each MM, order of operations to the same memory location is preserved; a store operation is acknowledged once the cache module accepts the request, regardless if it is a cache hit or miss. Some performance enhancements were already incorporated in the XMT computer seeking to optimize LSRTM and queuing delay: (i) broadcast: in case most threads in a spawn-join section need to read a variable, it is broadcasted through the instruction broadcasting bus to TCUs rather than reading the variable serially from the shared memory. (ii) Software prefetch mechanism with hardware support to alleviate the interconnection network round trip delay. A prefetch instruction brings the data to a prefetch buffer at the TCUs. (iii) Non-blocking stores where the program allows a TCU to advance once the interconnection network accepts a store request without waiting for an acknowledgement.

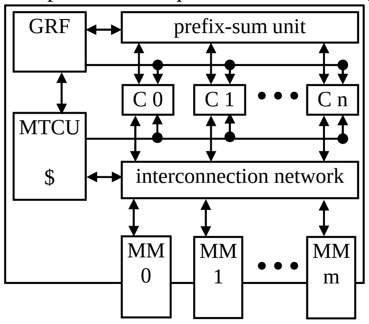

Fig.1 Block diagram of XMT

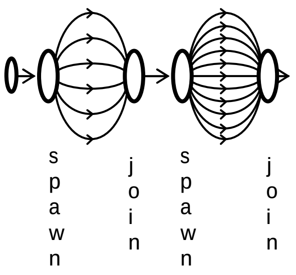

Fig. 2 Parallel and serial mode

The new XMT system consists of 3 FPGA chips graciously donated by Xilinx : 2 Virtex-4 LX200 and 1 Virtex-4 FX100. PCI is used as the interface. In order to prototype as much parallelism (e.g., number of TCUs) as possible: (i) there is no floating point (FP) support, and (ii) the MTCU is a weak in-order processor. Specifications of the system are presented in the table below. To fully utilize the bandwidth of the interconnection network, two cache modules are sharing one interconnection network port, resulting in a total of 8 cache modules.

Some specifications of the FPGA XMT system

Clock rate	75 MHz	Number of TCUs	64 (4 X 16)
Memory size	1GB DDR2	Shared cache size	64KB (8 X 8)
Mem. data rate	2.4GB/s	MTCU local cache	8KB

Execution time in seconds

Application	Basic	XMT	Opteron
M_Mult	182.8	80.44	113.83
Sorting	16.066	7.573	2.615

Two applications were tested: (i) dense integer matrix multiplication and (ii) randomized quicksort. Column XMT in the execution time table gives the XMT computer execution time. Column Basic gives XMT time without the 3 enhancements above. Column Opteron gives time of a 2.6 GHz AMD Opteron processor, with 64KB+64KB L1, 1MB L2, memory bandwidth 6.4GB/s (2.67X larger than XMT) using GCC in RedHat Linux Enterprise 3. Integer multiplication takes 5 cycles in the FPGA system. Since FP multiplication often also takes around 5 cycles, execution time on XMT system with FP support should also be similar. The size of the matrix was 2000x2000. The serial sorting is a standard randomized quicksort. A known parallel quicksort was used for XMT. Input size was 20 million integers. As can be seen, the run time of the 75MHz XMT parallel computer was a bit faster than the recent 2.6 GHz AMD uniprocessor on matrix multiplication and slower on quicksort. Also, initial performance study of the FPGA implementation for several micro-benchmarks confirms prior simulation-based results. Ongoing work includes architecture analysis using the FPGA system as a prototype.

Conclusion Using on-chip low overhead mechanisms including a high throughput interconnection network XMT executes PRAM-like programs efficiently. As XMT evolved from PRAM algorithm, it gives (i) an easy general-purpose parallel programming model, while still providing (ii) good performance with any amount of parallelism provided by the algorithm (up- and down-scalability), and (iii) backwards compatibility on serial code using its powerful MTCU with its local cache. Most other parallel programming approaches need more coarse-grained parallelism, requiring a (painful to program) decomposition step.

A single (though hard working) individual (X. Wen) with no prior design experience was able to complete synthesizable HDL description in Verilog, as well as the new FPGA-based XMT computer in slightly more than two years. This attests to the basic simplicity of the XMT architecture and ease of implementing it (in addition to the relative simplicity of its parallel programming).

References

[1] U. Vishkin, G. Caragea and B. Lee. Models for advancing PRAM and other algorithms into parallel programs for a PRAM-On-Chip platform. In Handbook of Parallel Computing: Models, Algorithms and Applications, Editors: R. Rajasekaran and J. Reif, CRC Press, Sept. 2007, to appear.
[2] U. Vishkin. Tutorial at ISCA'07 and ICS'07.