Layout-Accurate Design and Implementation of a High-Throughput Interconnection Network for Single-Chip Parallel Processing*

Aydin O. Balkan† Michael N. Horak† Gang Qu† Uzi Vishkin†
{balkanay, mnhorak, gangqu, vishkin}@umd.edu

Abstract

A Mesh of Trees (MoT) on-chip interconnection network has been proposed recently to provide high throughput between memory units and processors for single-chip parallel processing [5]. In this paper, we report our findings in bringing this concept to silicon. Specifically, we conduct cycle-accurate verilog simulations to verify the analytical results claimed in [5]. We synthesize and obtain the layout of the MoT interconnection networks of various sizes. To further improve throughput, we investigate different arbitration primitives to handle load and store, the two most common memory operations. We also study the use of pipeline registers in large networks when there are long wires. Simulation based on full network layout demonstrates that significant throughput improvement can be achieved over the original proposed MoT interconnection network.

The importance of this work lies in its validation of performance features of the MoT interconnection network, as they were previously shown to be competitive with traditional network solutions. The MoT network is currently used in an eXplicit Multi-Threading (XMT) on-chip parallel processor, which is engineered to support parallel programming. In that context, a 32-terminal MoT network could support up to 512 on-chip XMT processors. Our 8-terminal network that could serve 8 processor clusters (or 128 total processors), was also accepted recently for fabrication.

1. Introduction

There is a recent surge of interest in single-chip parallel processors. As more and more components are integrated in such systems, it becomes a severe challenge to design and implement the on-chip interconnection network. *Throughput* and *latency* are among the most important performance characteristics of such network that carries memory requests from processors to memory modules and responses from memory modules to the processors.

Traditional interconnection networks such as hypercube and butterfly have been used in parallel computing systems (e.g. [9, 11, 12, 21]). However, their performance is limited in case of the described high traffic load of a parallel processor. Similarly, a more recent on-chip network [3] could provide high performance for local traffic, when processors have private caches and there are only a few globally shared memory units. However its 2D-mesh topology would throttle the throughput in a fully shared memory configuration with uniform traffic distribution. The main problem of these networks is that packets experience contention at high traffic rates, and this reduces the performance.

A mesh-of-tree (MoT) based network has been recently proposed to solve this problem by separating routing and arbitration operations. It guarantees interference-free communication and promises to deliver high throughput that is close to the theoretical maximum [5] (see Section 2). We discuss the impact of this promise on single-chip parallel processing later in Section 4.3.

Although the performance promise of MoT has been demonstrated, there are two major steps that remained to be done. First, the MoT network as a whole needed to be validated for accuracy (see Section 3.1). Second, it considers only short packets such as the most common *load* operation and not longer packets such as the *store* operation that is essential for completeness (see Section 3.2). Therefore, it was not yet established in [5] that the MoT network's promise can be accomplished in silicon.

This paper bridges these gaps and brings the concept of MoT network closer to reality. Cycle-accurate verilog simulations validate the claims in [5]; and physical design of various MoT networks evaluates layout-accurate performance. An extended arbitration mechanism supports *store* operation with high throughput. Finally, pipelined long wires improve throughput by increasing operating frequency. As part of the PRAM-on-Chip project, each terminal of MoT network could serve a processor cluster of up to 16 processors [20]. Our 8-terminal network that could serve 128 processors has been accepted for fabrication.

2. Background

In this section, we briefly review the network topology and the arbitration primitives in MoT network. Additional

*Partially supported by grant 0325393 from the National Science Foundation.

†University of Maryland Institute for Advanced Computer Studies (UMIACS), and Electrical and Computer Engineering Department

details on background, network features and operation have been discussed in [5].

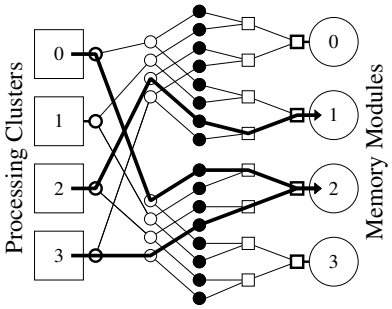


Figure 1. Mesh of Trees with 4 Clusters and 4 Memory Modules.

The MoT network consists of two main structures, a set of fan-out (routing) trees and a set of fan-in (arbitration) trees. Figure 1 shows the communication paths from processor clusters to memory modules for three memory requests. Paths of memory requests (0,2) (2,1) and (3,2) are highlighted. Empty circles and squares represent routing (Fig 2.a) and arbitration (Fig 2.b) primitives respectively. There is a unique path between Each memory request will travel from the source through a fan-out tree and then a fan-in tree before it reaches the destination. In fan-out trees, routing decision is trivial from the binary representation of the destination address. There is no routing decision in the fan-in trees, since each packet in a fan-in tree has the same destination.

Contention could occur, when two packets from different sources to different destinations compete for a shared resource. Fan-out trees eliminate competition between packets from different sources, and fan-in trees eliminate competition between packets to different destinations. This separation avoids contention and improves throughput.

Flow control is handled locally, through handshakes between successive switching primitives. A slightly modified version of a *relay station* [6] is used to prevent data loss when the successor primitive is full. Figure 2 illustrates the switching primitives in our MoT network. Each node in the fan-out and fan-in trees of the network will be implemented using the fan-out (Figure 2(a)) or fan-in (Figure 2(b)) primitives. The pipeline primitive (Figure 2(c)) is used to divide long wires into multiple short segments.

Main results of [5] show that MoT sustains high throughput and low latency at high traffic rates. The peak throughput, or the network capacity, for a MoT network is 1.0 packets-per-cycle (ppc) at each port; and the bisection bandwidth is equal to N ppc, where N is the number of ports. For example, in a 64-terminal MoT, the average throughput of all terminals reaches $0.98ppc$ under sustained uniform traffic that is injected at 100% network capacity Average packet latency is 14 cycles at a low traffic rate such as 10% of network capacity, and 23 cycles at a high traffic rate such as 90% of network capacity.

3. Design and Implementation Flow

In this section we first explain the importance of validating the previous results on MoT network with cycle-accurate Verilog simulator. We then modify the arbitration primitive to support the *store* operation. We describe the physical design of the MoT network as a further step towards evaluating its layout-accurate performance. Finally, pipelines are inserted to deal with the long wire delays.

3.1. Cycle-Accurate Validation

In [5], the performance model of the MoT network has been evaluated using a custom-made simulator, written in C++ using SystemC libraries. There was no earlier study of a cycle-accurate simulator for verifying the MoT network model in [5]. To demonstrate accuracy, some butterfly network simulations has been compared with the "booksim" simulator of [9]. However, the simulator in [5] is optimized for MoT network, and the simulator in [9] is optimized for traditional networks such as hypercube and butterfly. Therefore, the accuracy of the comparison was limited.

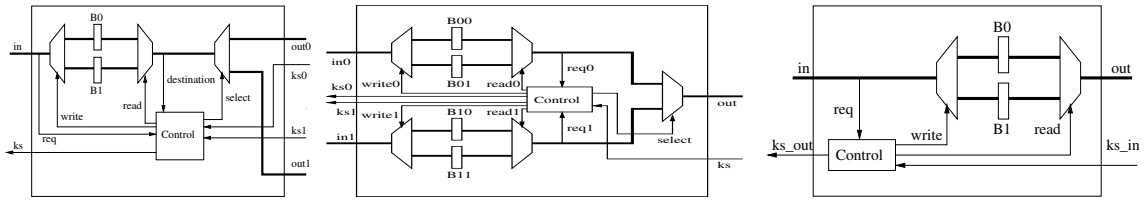
Prior to the current paper, switch primitives have been individually synthesized into generic technology, but the whole MoT network has not been synthesized and verified. Therefore, a realistic hardware model was not available for validation. In this paper we derive a synthesizable verilog model of the full MoT network using our own high level synthesis tool. We perform RTL and gate-level netlist simulations, and validate earlier results.

We assume uniform traffic pattern, which is expected for the memory architecture described in [16], due to the use of a hashing mechanism [2, 4, 10, 15].

3.2. Modified Arbitration

The smallest unit of information flowing in the network is called *flit* or *flow control digit* [9]. The performance model of [5] is based on exchanging single-flit packets between terminals. In case of a *load* operation, the processor sends the address to the memory module, and the memory module responds with the requested data. In this most common mode of operation, each packet consists of a single flit with sufficiently many bits, that contains either the address or the data.

In case of a *store* operation the processor sends the address and the data to the memory module. A flit could be sufficiently wide to hold both the address and the data, however this would waste bandwidth when *load* instructions are sent through the network. Alternatively, a *store* packet could consist of two flits that are injected consecutively to



(a) Routing (Fan-Out) primitive (b) Arbitration (Fan-In) primitive (c) Pipeline primitive

Figure 2. Switch primitives for MoT network. Data paths are marked with thick lines. Control paths are simplified. Signals: *req*: Request; *ks*: Kill-and-Switch; *write*/*read*: Write and Read pointers; *B*: Storage Buffer; *select*:Result of Arbitration; *destination*: Destination address.

the network. In this case additional effort is required to relate address and data pairs that belong together, and perform the correct operation. We consider the following two options for handling *store* operations.

- Both flits of address–data pair can be marked with an identifier tag, and sent as individual single-flit packets. The memory commits the operation when the second flit with the matching tag arrives. This method requires computation on the processor and the memory module. The network remains unchanged. This is called *fair bandwidth* arbitration [9], since the arbitration primitives perform fair arbitration regardless of the type of the packet.

- Second flit is chained to the first one, and they follow each other in the network. The memory receives the pair consecutively.This method requires computation in the network. Specifically, the arbitration primitive must ensure that second flit immediately follows the first one. This introduces a temporary bias to the arbitration operation. The processor and the memory modules remain unchanged. This method is called *winner-take-all* arbitration [9]. Extra logic in the arbitration primitive may increase clock period and, therefore, reduce throughput. On the other hand, this method reduces average packet latency for multi-flit packets in terms of clock cycles.

We modified the regular arbitration primitive to perform *winner-take-all* arbitration. We implement these two arbitration primitives and evaluate their performance with a cycle-accurate verilog simulator. The results show that they both provide similar throughput improvement over the single-flit arbitration used in [5]. The improvement is significant especially when load operation dominates. See Section 4.1 for details.

3.3. Physical Design

For the layout of the MoT network, we start with RTL-level verilog description of switch primitives. Our own high-level synthesizer generates higher level modules, such as balanced binary trees.

Terminals	4	8	16	32	64
Bits per flit	26	28	30	32	34
Cell Area	0.064	0.314	1.419	6.166	26.289
Wire Area	0.003	0.020	0.135	0.863	5.197

Table 1. Wire and cell area (in mm^2).

3.3.1 Network Layout

The wire area of the MoT network grows as $O(N^2 \log^2 N)$, and the number of tree nodes grow as $O(N^2)$, where N is the number of terminals [5]. This would imply that the wire area will dominate the cell area, and the floorplanning must consider wire area constraints. Synthesis results with this particular technology and standard cell library show that cell area is larger than the wire area for practical number of terminals. Table 1 shows these results for different network configurations that are considered in this paper. Wire area grows faster and it can exceed cell area for higher number of terminals and bits per flit. Therefore, our floorplan and placement strategy in this study is based on the cell area of the network.

In a network with N terminals, we create $N/2$ partitions in order to improve layout quality during placement and routing. Figure 3 shows a network with 8 terminals that has 4 partitions marked P_0 to P_3. An initially square floorplan is separated into partitions, and each partition is individually placed, routed, and optimized. Depending on other geometrical factors, such as height and width of terminal modules, two partitions could be separated by a gap.

3.3.2 Terminal Circuits

Ideally, our network would interconnect parallel processors and memory modules. We use a terminal node to replace a pair of cluster and memory module. In order to focus on the interconnection network, these nodes are dummy terminals that generate random requests based on programmable parameters, and record statistics upon receiving a packet.

The terminal modules do not affect critical delay path of the network modules. However, since they are generating packets and recording arrivals at each cycle, their critical delay path affects the operation frequency of our taped-out chip. Therefore, we report critical delays for the network module separately.

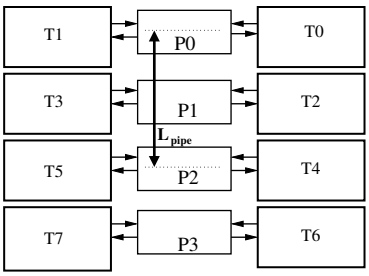


Figure 3. High level chip floorplan for 8-terminal network. Terminal modules: T_0 to T_7. Network partitions: P_0 to P_3.

3.4. Pipeline Insertion

Long wires of MoT network could increase the clock period and reduce the throughput. Inserting pipeline registers to long wires would improve performance [3,7,13,14]. Earlier work [5] proposed to use a pipeline primitive to cut long wires in shorter segments. However, the benefits could not be demonstrated without a physical layout.

Pipeline insertion can be automatized by several ways. State of the art synthesis tools are capable of inserting repeaters. However, they are usually unaware of final wire lengths. Place and route tools can insert any standard cell or module to an existing netlist and connect them to rest of the circuit. However, this requires use of low level commands of the specific tools, and may not be portable. Furthermore, state changes in the circuit cannot be traced back to RTL-level. This could complicate verification and performance evaluation. Our high level synthesis tool inserts pipeline registers at RTL level. Then, the network would have a portable and coherent state machine view through the entire physical design flow.

It is challenging to estimate the optimal wire length to fit in a single pipeline stage. It involves multiple physical design iterations. Furthermore, CAD tools perform several proprietary and heuristic optimizations. Therefore, it is virtually impossible to estimate the exact wire length between two consecutive registers before the layout is finalized.

In this prototyping study, we follow a high level heuristic approach to determine the amount of pipelining, guided by the wire length between the centers of partition P_i and the second partition P_{i+2}, denoted as L_{pipe} in Figure 3. Thus, we allow the signals to pass over one full partition P_{i+1} without being stored in a pipeline register. For lack of space, we only note that following this model, an 8-terminal network would not require pipelining. Furthermore, 16 and 32-terminal networks will ideally operate at the same frequency as the 8-terminal network.

4. Results and Discussion

In this section, we first present simulation results that validate the claims of [5] and provides average throughput per cycle. Then, we lay out networks with 4, 8, 16 and 32 terminals, and obtain their clock rate. The combination of both results will give layout-accurate average throughput for MoT. Finally, we taped-out the 8-terminal design for fabrication.

We used IBM CMOS9SF 90nm technology and regular ARM/Artisan SAGE-X standard cells. Typical operating conditions $(V_{DD}; T)$ for this library is given as $1.2V; 25°C$. In this paper we report delay estimations for a slow corner (worst case) operating conditions, such as $1.08V; 125°C$. We use NC-Verilog for simulations, Cadence RTL Compiler for synthesis, and Cadence SOC Encounter for layout generation. For tape-out, we use Synopsys Hercules for DRC, and Cadence Virtuoso for final details in layout.

4.1. Simulation Results

Latency and throughput characteristics for a 64-terminal network is compared with results of [5] in Figure 4. Table 2 compares the average throughput at highest traffic rate, and latency at three traffic levels. *Low*, *High*, and *Max* represent flit generation rates of 10%, 90%, 100% of network capacity. Throughput is averaged over all terminal ports, and latency is averaged over all recorded packets.

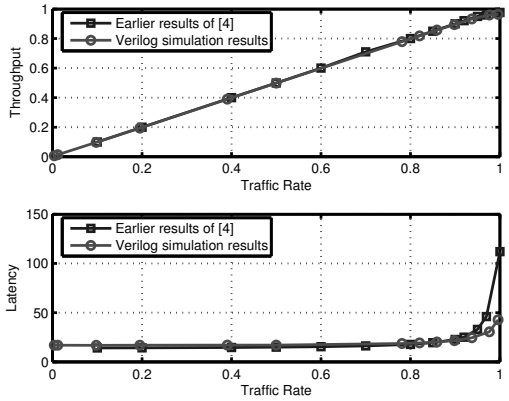


Figure 4. Throughput, and latency of 64-terminal MoT at various traffic rates. Verilog simulations compared to earlier results. Throughput is measured in terms of flits per cycle and averaged over all ports.

Compared to results of [5], throughput differs between 1% to 2%. Latency results for 64 terminal MoT network are 17% higher for low-traffic case and 6.5% lower for high-traffic case. Such deviations are expected due to different implementation of *source queue* component as described in Section 3.1.

We simulate the network with different ratios of 1-flit and 2-flit packets, to model a mixture of *load* and *store* operations. Traffic rate is adjusted for each run so that average

Terminals	4	8	16	32	64
Tput from [5]	N/A	N/A	0.95	0.96	0.98
Average Tput	0.88	0.91	0.93	0.95	0.96
Latency (low)	8.64	10.8	12.8	14.8	16.9
Latency (high)	18.0	16.9	17.9	19.3	21.6
Latency (max)	26.6	29.8	33.6	38.0	42.7

Table 2. Simulation results for different network configurations. Throughput is measured in flits per cycle per port, at the maximum traffic generation rate of 1 flit per cycle per port. Latency is measured in cycles.

flit injection rate remains constant at the maximum capacity of the network, namely 1 flit per cycle per port. Higher traffic rates would saturate the source queue in the terminal. In that case several packets would be dropped, and the mixture rate could change. For example, a mixture ratio of 30% means that each cycle there is a 77% probability of generating a packet. Additionally, the generated packet has two flits with a probability of 30%, and one flit with a probability of 70%. As a result, the average rate of flit generation is 1.0 per cycle. We simulated *fair arbitration* and *winner-take-all* arbitration methods as described in Section 3.2. The variation in latency and throughput for 64-terminal network is shown in Figure 5. The *wide flit* case assumes that the flit width is doubled so that any one of *load* or *store* operations fits in a single flit.

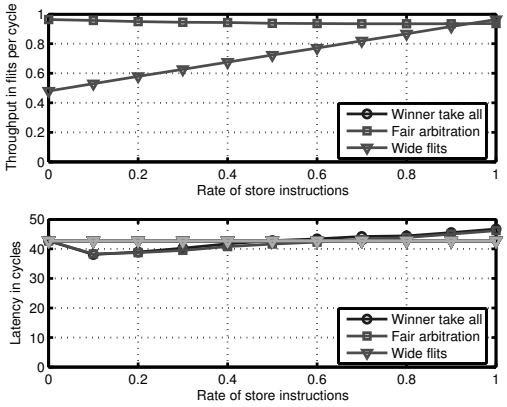


Figure 5. 64-terminal MoT simulation results for different methods of handling store operations.

Simulations show that using multiple flits for *store* instructions improves throughput for almost all mixture ratios. There is no significant difference between two methods of arbitration. Layout of both arbitration primitives shows that the increase in clock period due to additional logic is negligible. Latency is improved for low amounts

Config.	**4**	**8**	**16**	**32**	**16 p**	**32 p**
Clock Rate	970	890	680	578	748	764
Bits per flit	26	28	30	32	30	32
Peak Tput	101	199	326	592	359	782
Avg Tput	88.6	180	302	563	334	747
Low trf lat	8.64	10.8	12.8	14.8	13.5	17.8
High trf lat	18.0	16.9	17.9	19.3	18.7	22.6
Cell area	0.08	0.41	1.89	6.5	1.88	7.3
BBbox area	0.16	0.74	3.21	13.4	3.21	13.4
Power	72	268	794	N/A*	967	N/A*

Table 3. Comparison of MoT configurations after layout. The letter 'p' indicates pipelined configuration. "BBox" stands for bounding box. Clock rates are in MHz; throughput values are in $Gbps$; latency values are in cycles; area values are in mm^2; power is in mW. *Due to constraints on computing resources, these results are not available.

of *store* instructions, but this could also be caused by the source queue implementation. For 64-terminal network, *fair arbitration* has slightly lower latency. Additional simulations show that for a 4-terminal network, *winner-take-all* has lower latency. We conclude that the number of flits in a *store* instruction is not sufficiently high to make a difference in latency. Further studies with more flits per packet would be beneficial to evaluate MoT performance for cases where multiple data words are moved through the network, such as loading or storing long vectors or streams.

4.2. Layout Results

Following the standard flow of the Cadence tools, we synthesized, placed and routed networks with different configurations. Table 3 shows the area and performance results.

We extended the 8-terminal configuration with power routing and I/O pads for fabrication. The final layout is shown in Figure 6.

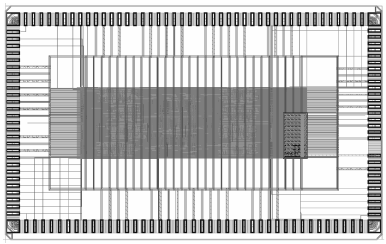


Figure 6. Final layout of 8-terminal chip.

Table 3 shows that the clock frequency reduces as the number of terminals increases. This is mainly caused by longer wires on the critical path. Results of pipelined configurations 16p and 32p show the benefit of pipelining on

frequency and throughput. Average latency increases in pipelined configurations due to increased number of stages between some sources and destinations.

Partitioning constraints prevented optimal pipeline placement on long wires. Therefore, the improved frequency did not reach the the expected level of an 8-terminal network. Reducing the critical length for pipelining could improve performance. Pipeline circuits would be placed within the partitions, instead of between them. Such improvements could incur additional area and latency cost. Evaluation of these trade-offs requires further studies.

Table 3 shows that the cell area of laid-out networks exceeds estimations (Table 1), since the layout tool optimizes for performance by inserting repeaters and using larger cells.

Cell area of 32p is larger than 32, as expected, due to additional pipeline stages. In 16p, the area of added pipeline stages turn out to be comparable to large repeaters on long wires of 16. Therefore, the area of 16p is approximately equal to the area of 16.

The area of the bounding box is approximately twice as much as the cell area, because of the gaps between partitions, and overestimated design margins. We introduced gaps between partitions in order to level the partitions with the terminals (Figure 3). The amount of gaps depend on the area and aspect ratio of terminal circuits. In an ongoing study, we are investigating the relationship between processor geometry, and MoT area and performance. In this prototyping study we did not optimize for the area. However, based on Table 1, we expect the actual area to be close to the cell area.

Power consumption has been estimated based on the layout, and simulated switching activity with highest traffic rate. As expected, the power consumption grows quadratically with the number of terminals, that is, at the same rate as the number of cells. Pipelining increases power consumption by both adding more cells, and increasing operating frequency. In this study, we did not optimize for power consumption. However, typical approaches such as clock-gating could reduce power consumption.

4.3. Impact on Single-Chip Parallel Processing

A clear lesson of several decades of parallel computing research is that the issue of parallel programming must be properly resolved. The Parallel Random Access Model (PRAM) is an easy model for parallel algorithmic thinking and for programming, as recognized by Culler and Singh [8], and at least 3 major standard texts on serial algorithms and data-structures. Earlier attempts to support PRAM by a multi-chip multiprocessor (e.g. TERA-MTA [2], SB-PRAM [17], NYU Ultracomputer and the IBM RP3, [1,10]) have been constrained on memory access performance and had limited success.

The "PRAM-on-Chip" project at the University of Maryland seeks to advance implementation of PRAM in a single-chip parallel processor, using an eXplicit Multi-Threading (XMT) architecture (see Appendix). The XMT architecture eliminates local private caches in order to avoid cache coherence issues and uses hashing mechanism to avoid hot spots [16]. This dramatically increases the load on the interconnection network and makes the network traffic reasonably uniform, rendering the current interconnection networks ineffective. MoT network, as we have described in Section 2, promises high throughput and low latency. This current work brings the concept of MoT network closer to silicon and thus has significant impact.

Based on a recent design [20], each terminal port of the network could serve up to 16 processors and up to two globally shared cache modules. Our results show that, a pipelined 16-terminal network, supporting up to 256 processors, operates at $748MHz$. With 30-bit wide channels, it provides a peak throughput of $359Gbps$, and an average throughput of $334Gbps$ under uniform traffic.

5. Conclusion

We perform cycle-accurate Verilog simulation to validate the earlier results on MoT network, which has clear advantages on throughput and latency over traditional interconnection networks. For example, for a 64-terminal network, earlier results overestimated throughput by only 2%, and latency by 6.5% at high traffic load. We propose two extensions: support store operations and avoid long wire delay to further improve throughput of MoT. We conduct the physical design and obtain layout for MoT network of various sizes. The layout of 8-terminal network has recently been accepted by the foundry for fabrication.

While our initial layouts of switch primitives indicate an average throughput of 4.6Tbps in the ideal case for a 64-terminal MoT network, practical constraints led us to defer seeking such rate to future work, perhaps not in a university environment.

References

[1] G. S. Almasi and A. Gottlieb. *Highly Parallel Computing*. Benjamin/Cummings, second edition, 1994.

[2] R. Alverson, et al. The Tera Computer System. In *Proc. Int. Conf. On Supercomputing*, pages 1–6, 1990.

[3] F. Angiolini, et al. A Layout-Aware Analysis of Networks-on-Chip and Traditional Interconnects for MP-SoCs. *Computer-Aided Design of Integrated Circuits and Systems, IEEE Trans. on*, 26(3):421–434, March 2007.

[4] P. Bach, M. Braun, A. Formella, et al. Building the 4 processor SB-PRAM prototype. In *Proceedings of the Thirtieth Hawaii International Conference on System Sciences*, volume 5, pages 14–23, Jan. 1997.

[5] A. Balkan, G. Qu, and U. Vishkin. A Mesh-of-Trees Interconnection Network for Single-Chip Parallel Processing. In *Proceedings of the Application-Specific Systems, Architectures and Processors (ASAP)*, 2006.

[6] L. P. Carloni, et al. A Methodology for Correct-by-Construction Latency Insensitive Design. In *IEEE/ACM International Conference on Computer Aided Design (ICCAD)*, pages 301 – 315, 1999.

[7] P. Cocchini. Concurrent Flip-Flop and Repeater Insertion for High Performance Integrated Circuits. In *ICCAD '02: Proceedings of the 2002 IEEE/ACM International Conference on Computer-Aided Design*, pages 268–273, 2002.

[8] D. E. Culler and J. P. Singh. *Parallel Computer Architecture*. Morgan Kaufmann, 1999.

[9] W. J. Dally and B. Towles. *Principles and Practices of Interconnection Networks*. Morgan Kaufmann, San Fransisco, CA, 2004.

[10] A. Gottlieb, et al. The NYU Ultracomputer–Designing an MIMD Shared Memory Parallel Computer. *IEEE Trans. Comput.*, pages 175–189, Feb. 1983.

[11] C. Grecu, P. Pande, A. Ivanov, and R. Saleh. A scalable communication-centric SoC interconnect architecture. In *Quality Electronic Design, 2004. Proceedings. 5th International Symposium on*, pages 343–348, 2004.

[12] R. I. Greenberg and L. Guan. On the Area of Hypercube Layouts. *Information Processing Letters*, 84:41–46, 2002.

[13] S. Hassoun, et al. Optimal Buffered Routing Path Constructions for Single and Multiple Clock Domain Systems. In *IEEE/ACM International Conference on Computer Aided Design (ICCAD)*, pages 247 – 253, 2002.

[14] R. Lu, G. Zhong, C. Koh, and K. Chao. Flip-Flop and Repeater Insertion for Early Interconnect Planning. In *DATE '02: Proceedings of the Conference on Design, Automation and Test in Europe*, page 690, 2002.

[15] K. Mehlhorn and U. Vishkin. Randomized and Deterministic Simulations of PRAMs by Parallel Machines with Restricted Granularity of Parallel Memories. *Acta Informatica*, 21:339–374, 1984.

[16] 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 Computer Systems*, 2003. Special Issue of SPAA 2001.

[17] W. J. Paul, et al. Real PRAM Programming. In B. Monien and R. Feldman, editors, *Proceedings of 8th International Euro-Par Conference*, page 522. Springer-Verlag Heidelberg, 2002.

[18] U. Vishkin. Tutorial on how to develop PRAM-like programs and run them on the FPGA prototype. ICS 2007, June 2007.

[19] U. Vishkin, G. Caragea, and B. Lee. Models for Advancing PRAM and Other Algorithms into Parallel Programs for a PRAM-On-Chip Platform. *Handbook on Parallel Computing: Models, Algorithms, and Applications*. Ed: S. Rajasekaran and J. Reif; CRC Press, 2007, to appear.

[20] X. Wen and U. Vishkin. PRAM-on-Chip: 1st committment to Silicon. In *ACM-SPAA 07*, June 2007, to appear.

[21] C.-H. Yeh. Optimal Layout for Butterfly Networks in Multilayer VLSI. In *International Conference on Parallel Processing (ICPP)*, pages 379 – 388, 2003.

A. Explicit Multi-Threading Architecture

The eXplicit Multi-Threading (XMT) on-chip general-purpose computer architecture [16] is aimed at the classic goal of reducing single task completion time. It is a parallel algorithmic architecture in the sense that it seeks to provide good performance for parallel programs derived from Parallel Random Access Machine/Model (PRAM) algorithms. 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. First commitment to silicon of XMT is reported in [20]. A 64-processor, 75MHz computer based on field-programmable gate array (FPGA) technology was built at the University of Maryland (UMD).

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 [19]. 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 Smiths 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.

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-Singhs 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 and the interconnection network is being realized in ASIC. The current paper is part of an overall effort to advance the perception of PRAM implementability from impossible to available. The effort provides freedom and opportunity to pursue PRAM-related research, development and education without waiting for vendors to make the first move. For example, consider [18].

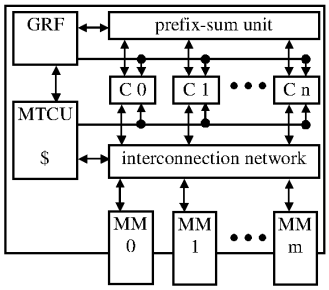


Fig. A.1 Block diagram of the XMT

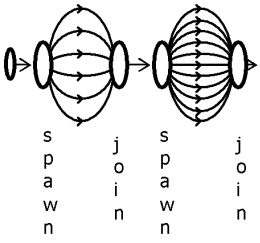


Fig. A.2 Parallel and serial mode

Overview of the XMT Architecture The XMT processor (see Fig. A.1) includes a master thread control unit (MTCU), processing clusters (C0...Cn in Fig. A.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 prefix-sum unit. Fig. A.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. [16] 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. A.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. 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. (iv) Read-only-buffer: Within a TCU cluster, read requests to the same memory location from multiple TCUs operating concurrently can be replaced by a single request into the interconnection network. This optimization is applied only to addresses that cannot be written into by any concurrent thread.

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.