Toolchain for Programming, Simulating and Studying the XMT Many-Core Architecture

Fuat Keceli, Alexandros Tzannes, George C. Caragea, Rajeev Barua and Uzi Vishkin
University of Maryland, College Park
{keceli,tzannes,gcaragea,barua,vishkin}@umd.edu

***Abstract*—The Explicit Multi-Threading (XMT) is a general-purpose many-core computing platform, with the vision of a 1000-core chip that is easy to program but does not compromise on performance. This paper presents a publicly available toolchain for XMT, complete with a highly configurable cycle-accurate simulator and an optimizing compiler.**

The XMT toolchain has matured and has been validated to a point where its description merits publication. In particular, research and experimentation enabled by the toolchain played a central role in supporting the ease-of-programming and performance aspects of the XMT architecture. The compiler and the simulator are also important milestones for an efficient programmer's workflow from PRAM algorithms to programs that run on the shared memory XMT hardware. This workflow is a key component in accomplishing the dual goal of ease-of-programming and performance.

The applicability of our toolchain extends beyond specific XMT choices. It can be used to explore the much greater design space of shared memory many-cores by system researchers or by programmers. As the toolchain can practically run on any computer, it provides a supportive environment for teaching parallel algorithmic thinking with a programming component. Unobstructed by techniques such as decomposition-first and programming for locality, this environment may be useful in deferring the teaching of these techniques, when desired, to more advanced or platform-specific courses.

***Index Terms*—Architecture Simulation, Compiler, Cycle-Accurate Simulator, Discrete-Event Simulator, Ease-of-Programming, Explicit Multi-Threading (XMT), Extending Serial Compiler for Parallelism, Fine-Grained Parallelism, Many-Core Architecture, Memory Model, Parallel Algorithms, Parallel Computing Education, Parallel Programming, PRAM, Toolchain, XMTC**

I. Introduction

Building a high-productivity general-purpose parallel system has been a grand challenge for parallel computing and architecture design communities. Especially, the goal of obtaining high-productivity, which is the combination of ease-of-programming and good performance, has been elusive. This brings us to today's landscape of general-purpose on-chip parallel computing with the hard-to-scale multi-cores consisting of up to a few tens of threads (*e.g.*, Intel Core i7), vs. highly parallel, scalable many-cores with hundreds of threads (*e.g.*, GPUs); though GPUs are difficult to program and can be slower for code that exhibit a low amount of parallelism.

It is important to break out of the shells of patching prior approaches and start from a clean slate for overcoming the productivity problem, as argued by Snir [1]. The eXplicit Multi-Threading (XMT) general-purpose parallel computer represents a promising attempt in this respect. XMT is a synergistic approach, combining a simple abstraction [2], a PRAM-based rich algorithmic theory that builds on that abstraction, a highly-scalable efficient hardware design and an optimizing compiler. The programmer's workflow of XMT aims to preempt the productivity "busters" of on-chip parallel programming, such as decomposition-first programming, reasoning about complex race conditions among threads, or requiring high amounts of parallelism for improving performance over serial code.

In this paper, we present the publicly-available XMT toolchain, consisting of a highly-configurable cycle-accurate simulator, XMTSim, and an optimizing compiler [3]. XMT envisions bringing efficient on-chip parallel programming to the mainstream and the toolchain is instrumental in obtaining results to validate these claims as well as making a simulated XMT platform accessible from any personal computer. We believe that the XMT toolchain and its documentation in this paper are important to a range of communities such as system architects, teachers and algorithm developers due to the following four reasons.

1. Performance advantages of XMT and PRAM algorithms. In Section II-B, we list publications that not only establish the performance advantages of XMT compared to exiting parallel architectures, but also document the interest of the academic community in such results. The XMT compiler was the essential component in all experiments while the simulator enabled the publications that investigate planned/future configurations. Moreover, despite past doubts in the practical relevance of PRAM algorithms, results facilitated by the toolchain showed not only that theory-based algorithms can provide good speedups in practice, but that sometimes they are the only ones to do so.

2. Teaching and experimenting with on-chip parallel programming. The toolchain enabled the experiments that established the ease-of-programming of XMT. These experiments were presented in publications [4]–[7] and conducted in courses taught to graduate, undergraduate, high-school and middle-school students including at Thomas Jefferson High School, Alexandria, VA. In addition, the XMT toolchain provides convenient means to teaching parallel algorithms and programming, because students can install and use it on any personal computer to work on their assignments.

3. Opportunity to evaluate alternative system components. The simulator allows users to change the parameters of the simulated architecture including the number of functional units, and organization of the parallel cores. It is also easy

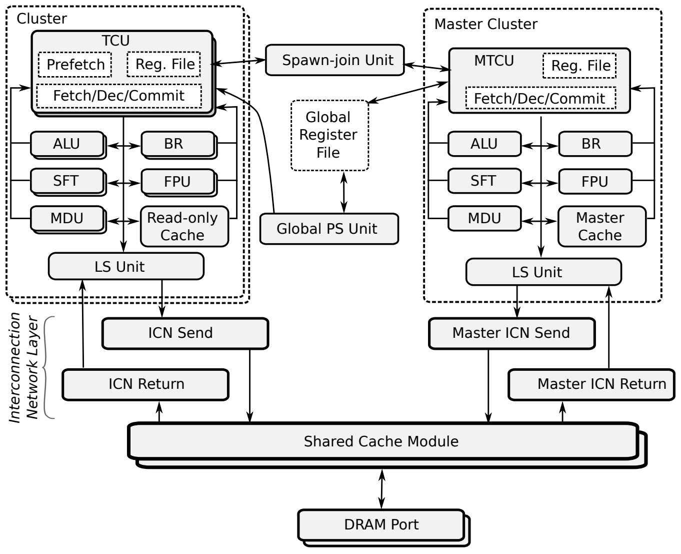
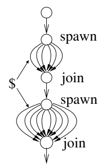


Fig. 1. Overview of the XMT architecture.

```
int A[N],B[N],base=0;
spawn(0,N-1) {
    int inc=1;
    if (A[$]!=0) {
        ps(inc,base);
        B[inc]=A[$];
} }
```

(a)



(b)

Fig. 2. (a) XMTC program example: Array Compaction. The non-zero elements of array A are copied into an array B. The order is not necessarily preserved. $ refers to the unique thread identifier. After the execution of the prefix-sum statement `ps(inc,base)`, the `base` variable is increased by `inc` and the `inc` variable gets the original value of `base`, as an **atomic** operation. (b) Execution of a sequence of `spawn` blocks.

to add new functionality to the simulator, making it the ideal platform for evaluating both architectural extensions and algorithmic improvements that depend on the availability of hardware resources. For example, [8] searches for the optimal size and replacement policy for prefetch buffers given limited transistor resources. Furthermore, to our knowledge, XMTSim is the only publicly available many-core simulator that allows evaluation of mechanisms, such as dynamic power and thermal management. Finally, the capabilities of our toolchain extend beyond specific XMT choices: system architects can use it to explore a much greater design-space of shared memory many-cores.

4. Guiding researchers for developing similar tools. This paper also documents our experiences on adapting an existing serial compiler (GCC v4.0) to fit a fine-grained parallel programming environment and constructing a simulator for a highly-parallel architecture, which, we believe, will guide other researchers who are in the process of developing similar tools.

II. Overview of the XMT Platform

The primary goal of the eXplicit Multi-Threading (XMT) on-chip general-purpose computer architecture [9], [10] has been improving single-task performance through parallelism. XMT was designed from the ground up to capitalize on the huge on-chip resources becoming available in order to support the formidable body of knowledge, known as Parallel Random Access Model (PRAM) algorithmics [11], [12], and the latent, though not widespread, familiarity with it. A 64-core FPGA prototype was reported and evaluated in [13], [14].

The XMT architecture, depicted in Fig. 1, includes a multitude of lightweight cores, called Thread Control Units (TCUs), and a serial core with its own cache (Master TCU). TCUs are grouped into clusters, which are connected by a high-throughput interconnection network, for example using a mesh-of-trees topology [15], [16], to the first level of cache (L1). The L1 cache is shared and partitioned into mutually-exclusive cache modules, sharing several off-chip DRAM memory channels. The load-store (LS) unit applies hashing on each memory address to avoid hotspots. Cache modules handle concurrent requests, which are buffered and reordered to achieve better DRAM bandwidth utilization. Within a cluster, a read-only cache is used to store constant values across all threads. TCUs include lightweight ALUs, shift (SFT) and branch (BR) units, but the more expensive multiply/divide (MDU) and floating point units (FPU) are shared among TCUs in a cluster. TCUs also feature prefetch buffers which are utilized via a compiler optimization to hide memory latencies.

XMT allows concurrent instantiation of as many threads as the number of available processors. Tasks are efficiently started and distributed thanks to the use of prefix-sum for fast dynamic allocation of work and a dedicated instruction and data broadcast bus. The high-bandwidth interconnection network (ICN) and the low-overhead creation of many threads facilitate effective support of both fine-grained and small-scale parallelism.

A. From C to XMTC: Simple Extensions

Fig. 2a illustrates XMTC, the programming language of XMT. It is a modest single-program multiple-data (SPMD) parallel extension of C with serial and parallel execution modes. The *spawn statement* introduces parallelism in XMTC. It is a type of parallel "loop" whose "iterations" *can* be executed in parallel. It takes two arguments `low`, and `high`, and a block of code, the *spawn block*. The block is concurrently executed on *(high-low+1) virtual threads*. The ID of each virtual thread can be accessed using the dollar sign ($) and takes integer values within the range $low \leq \$ \leq high$. Variables declared in the spawn block are private to each virtual thread. All virtual threads must complete before execution resumes after the spawn block. In other words, a spawn statement introduces an implicit synchronization point. The number of virtual threads created by a spawn statement is independent from the number of TCUs in the XMT system. An algorithm designed following the XMT workflow [17] permits each virtual thread to progress at its own speed, without ever having to busy-wait for other virtual threads.

XMTC also provides access to the powerful hardware prefix-sum (`ps`) primitive, similar in function to the NYU

Ultracomputer atomic Fetch-and-Add [18]. It provides constant, low overhead coordination between virtual threads, a key requirement for implementing efficient fine-grained parallelism. Although the `ps` operation is efficient, it can only be performed over a limited number of global registers and only with values 0 and 1. For that reason XMTC also provides a prefix-sum to memory (`psm`) variant that doesn't have these limitations: the *base* can be any memory location and the value of the *amount* can be any signed (32 bit) integer. The `psm` operations are more expensive than `ps` as they require a round trip to memory and multiple operations that arrive at the same cache module will be queued.

More details on XMTC can be found in the XMT Toolchain Manual [19].

B. Performance Advantages

A cycle-accurate 64-core FPGA hardware prototype [13], [14] was shown to outperform an Intel Core 2 Duo processor [20], despite the fact the Intel processor uses more silicon resources. The XMT compiler was an essential component in this experiment. The simulator allowed comparing a 1024-core XMT chip to a silicon-area equivalent GPU, NVIDIA GTX280. Simulations revealed that, while easier to program than the GPU, XMT still has the potential of coming ahead in performance [21], under similar thermal constraints [22]. Another comparison with GPUs can be found in [23], in which the floating-point model of the simulator was what enabled the publication. A comparison of FFT (the Fast Fourier Transform) on XMT and on multi-cores showed that XMT can both get better speedups and achieve them with less application parallelism [24].

Many doubt the practical relevance of PRAM algorithms, and past work provided very limited evidence to alleviate these doubts; [25] reported speedups of up to 4x on biconnectivity using a 12-processor Sun machine and [26] up to 2.5x on maximum flow using a hybrid CPU-GPU implementation when compared to best serial implementations. New results, however, show that parallel graph algorithms derived from PRAM theory can provide significantly stronger speedups than alternative algorithms. These results include potential speedups of 5.4x to 73x on breadth-first search (BFS) and 2.2x to 4x on graph connectivity when compared with optimized GPU implementations. Also, with respect to best serial implementations on modern CPU architectures, we observed potential speedups of 9x to 33x on biconnectivity [27], and up to 108x on maximum flow [28]. The toolchain allows calibrating the number of XMT processors in order to match the silicon areas of the same-generation platforms, which facilitated a fair comparison.

C. Ease of Programming and Teaching

Ease-of-programing is one of the main objectives of XMT: considerable amount of evidence was developed on *ease of teaching* [4], [5] and improved *development time* with XMT relative to alternative parallel approaches including MPI [6], OpenMP [7] and CUDA (experiences in [21]). XMT provides a *programmer's workflow* for deriving efficient programs from PRAM algorithms, and reasoning about their execution time [17] and correctness.

In a joint teaching experiment between the University of Illinois and the University of Maryland comparing OpenMP and XMTC programming [7], none of the 42 students achieved speedups using OpenMP programming on the simple irregular problem of breadth-first search (BFS) using an 8-processor SMP, but reached speedups of 8x to 25x on XMT. Moreover, the PRAM/XMT part of the joint course was able to convey algorithms for more advanced problems than the other parts.

III. The XMT Simulator – *XMTSim*

XMTSim is the cycle-accurate simulator of the XMT architecture. It accurately models the interactions between the high level micro-architectural components of XMT shown in Fig. 1, *i.e.*, the TCUs, functional units, caches, interconnection network, etc. Currently, only on-chip components are simulated, and DRAM is modeled as simple latency. XMTSim is highly configurable and provides control over many parameters including number of TCUs, the cache size, DRAM bandwidth and relative clock frequencies of components. XMTSim is verified against the 64-TCU FPGA prototype of the XMT architecture. The results of the verification, as well as details about XMTSim further than presented in this paper can be found in [29].

While simulators of serial and multi-core computer architectures are common in the research community (for example, SimpleScalar [30], Simics [31] and their derivatives), simulation of many-cores (*i.e.*, hundreds or more) is a relatively new and growing area. Some examples of these simulators are GPGPUSim [32] and TPTS [33]. XMTSim is complementary to the existing many-core simulators, as it fulfills the need for an easy-to-program shared-memory platform.

A. Overview

The software structure of XMTSim is geared towards providing a suitable environment for easily evaluating additions and alternative designs. XMTSim is written in the Java programming language and the object-oriented coding style isolates the code of major components in individual units (Java classes). Consequently, system architects can override the model of a particular component, such as the interconnection network or the shared caches, by only focusing on the relevant parts of simulator. Similarly, a new assembly instruction can be added via a two step process: (a) modify the assembly language definition file of the front-end, and (b) create a new Java class for the added instruction. The new class should extend *Instruction*, one of the core Java classes of the simulator, and follow its application programming interface (API) in defining its functionality and type (ALU, memory, etc.).

Each solid box in Fig. 1 corresponds to a Java object in XMTSim. Simulated assembly instruction instances are wrapped in objects of type *Package*[1]. An instruction package originates at a TCU, travels through a specific set of cycle-accurate components according to its type (*e.g.*, memory, ALU) and expires upon returning to the commit stage of the originating TCU. A cycle-accurate component imposes a delay on packages that travel through it. In most cases, the specific amount of the delay depends on the previous packages that entered the component. In other words, these components are state machines, where the state input is the instruction/data packages and the output is the delay amount. The inputs and the states are processed at transaction-level rather than bit-level accuracy, a standard practice which significantly improves the simulation speed in high-level architecture simulators. The rest of the boxes in Fig. 1 denote either the auxiliary classes that help store the state or the classes that enclose collections of other classes.

Fig. 3 is the conceptual overview of the simulator architecture. The inputs and outputs are outlined with dashed lines. A simulated program consists of assembly and memory map files that are typically provided from the XMTC compiler. A memory map file contains the initial values of global variables. The current version of the XMT toolchain does not include an operating system, therefore global variables are the only way to provide input to XMTC programs, since OS dependent features such as file I/O are not yet supported. The front-end that reads the assembly file and instantiates the instruction objects is developed with SableCC, a Java-based parser-generator [34]. The simulated XMT configuration is determined by the user typically via configuration files and/or command line arguments. The built-in configurations include models of the 64-TCU FPGA prototype (also used in the verification of the simulator) and an envisioned 1024-TCU XMT chip.

XMTSim is execution-driven (versus trace-driven). This means that instruction traces are not known ahead of time but instructions are generated and executed by a functional model during simulation. The functional model contains the operational definition of the instructions, as well as the state of the registers and the memory. The core of the simulator is the cycle-accurate model, which consists of the cycle-accurate components and an event scheduler engine that controls the flow of simulation. The cycle-accurate model fetches the instructions from the functional model and returns the expired instructions to the functional model for execution, which is illustrated in Fig. 3.

The simulator can be set to run in a fast functional mode, in which the cycle-accurate model is replaced by a simplified mechanism that serializes the parallel sections of code. The functional simulation mode does not provide any cycle-accurate information hence it is orders of magnitude faster than the cycle-accurate mode and can be used as a fast, limited debugging tool for XMTC programs. However, the functional mode cannot reveal any concurrency bugs that might exist in a

[1]*Package* in this context is one of the core classes of the simulator. It should not be confused with the concept of Java packages.

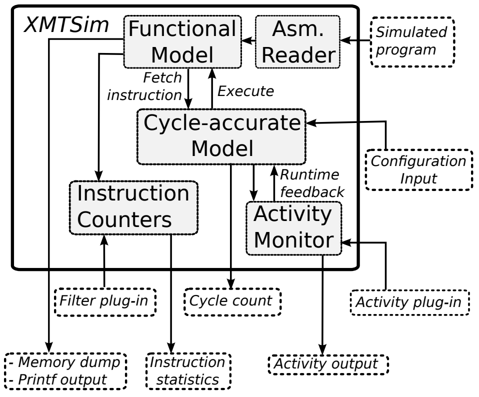


Fig. 3. Overview of the simulation mechanism, inputs and outputs.

parallel program since it serializes the execution of the spawn blocks. Another potential use for the functional simulation mode is fast-forwarding through time consuming steps (*e.g.*, OS boot, when made available in future releases) which would not be possible in the cycle-accurate mode due to simulation speed constraints.

B. Simulation Statistics and Runtime Control

As shown in Fig. 3, XMTSim features built-in counters that keep record of the executed instructions and the activity of the cycle-accurate components. Users can customize the instruction statistics reported at the end of the simulation via external *filter plug-in*s. For example, one of the default plug-ins in XMTSim creates a list of most frequently accessed locations in the XMT shared memory space. This plug-in can help a programmer find lines of assembly code in an input file that cause memory bottlenecks, which in turn can be referred back to the corresponding XMTC lines of code by the compiler. Furthermore, instruction and activity counters can be read at regular intervals during the simulation time via the *activity plug-in* interface. Activity counters monitor many state variables. Some examples are the number of instructions executed in functional units and the amount of time that TCUs wait for memory operations.

A feature unique to XMTSim is the capability to evaluate runtime systems for dynamic power and thermal management. The activity plug-in interface is a powerful mechanism that renders this feature possible. An activity plug-in can generate execution profiles of XMTC programs over simulated time, showing memory and computation intensive phases, power, etc. Moreover, it can change the frequencies of the clock domains assigned to clusters, interconnection network, shared caches and DRAM controllers or even enable and disable them. The simulator provides an API for modifying the operation of the cycle-accurate components during runtime in such a way.

C. Discrete-Event Simulation

Discrete-event (DE) simulation is a technique that is often used for understanding the behavior of complex systems [35]. In DE simulation, a system is represented as a collection of blocks that communicate and change their states via asynchronous events. XMTSim was designed as a DE simulator for two main reasons. First is its suitability for large object oriented designs. A DE simulator does not require the global picture of the system and the programming of the components can be handled independently. This is a desirable strategy for XMTSim as explained earlier. Second, DE simulation allows modeling not only synchronous (clocked) components but also asynchronous components that require a continuous time concept as opposed to discretized time steps. This property enabled the ongoing asynchronous interconnect modeling work mentioned in Section III-F.

The building blocks of the DE simulation implementation in XMTSim are *actors*, which are objects that can schedule *events*. An actor is *notified* via a callback function when the time of an event it previously scheduled comes. A cycle-accurate component in XMTSim might extend the actor type, contain one or more actor objects or exist as a part of an actor, which is a decision that depends on factors such as simulation speed, code clarity and maintainability.

Any activity during simulation takes place due to one of the two reasons: (a) An actor is notified by the DE scheduler. The action code of the actor determines what it should do next. (b) An instruction or data package is passed from one cycle-accurate component to another, which implements the *Inputable* interface of the simulator. Inputable components are required to define the action that should be performed upon receiving an input package. In both cases some of the typical actions are to schedule another event, trigger a state change or move data between the cycle-accurate components.

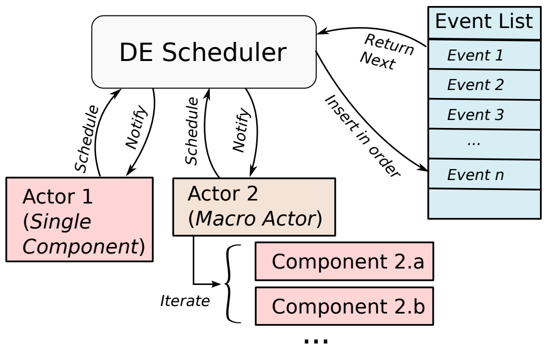


Fig. 4. The overview of DE scheduling architecture of the simulator.

Fig. 4 is an example of how actors schedule events and are notified. DE scheduler is the manager of the simulation that keeps the events in a list-like data structure, the *event list*, ordered according to their schedule times and priorities. In this example, *Actor 1* models a single cycle-accurate component whereas *Actor 2* is a *macro-actor*, which schedules events and contains the action code for multiple components.

Communication between components requires a mechanism that splits each clock cycle into two phases. In the first phase components negotiate the transfer of packages and in the second phase packages are passed between components. To support this mechanism, XMTSim introduces the concepts of

```
int time = 0;
while(true) {
  ...
  if(...) break;
  time++;
}
```

(a)

```
int time;
while(true) {
  Event e = eventList.next();
  time = e.time();
  e.actor().notify();
  if(...) break;
}
```

(b)

Fig. 5. Main loop of execution for (a) Discrete-time simulation, (b) Discrete-event simulation.

ports and *event priorities*. The event priority scheme ensures that the order of the phases is consistent among different clock cycles. Ports are used as points of transfer for packages.

It should be noted that XMTSim diverges from architecture simulators such as SimpleScalar [30] as it does not work in discrete-time (DT). The difference is illustrated in Fig. 5. The DT simulation runs in a loop that polls through all the modeled components and increments the simulated time at the end of each iteration. Simulation ends when a certain criteria is satisfied, for example when a *halt* assembly instruction is encountered. On the other hand, the main loop of the DE simulator handles one actor per iteration by calling its notify method. Unlike DT simulation, simulated time does not necessarily progresses at even intervals. Simulation is terminated when a specific type of event, namely the *stop* event is reached. The advantages of the DE simulation were mentioned at the beginning of this section. However, DT simulation may still be desirable in some cases due to its speed advantages and simplicity for simulated structures that are easier to handle. Only the former is a concern in our case and we elaborate further on simulation speed issues in the next section.

D. Simulation Speed

Simulation speed can be the bounding factor especially in evaluation of power and thermal control mechanisms, as these experiments usually require simulation of relatively large benchmarks. We evaluated the speed of simulation in throughput of simulated instructions and in clock cycles per second on an Intel Xeon 5160 Quad-Core Server clocked at 3GHz. The simulated configuration was a 1024-TCU XMT and for measuring the speed, we simulated various hand-written microbenchmarks. Each benchmark is serial or parallel, and computation or memory intensive. The results are averaged over similar types and given in Table I. It is observed that average instruction throughput of computation intensive benchmarks is much higher than that of memory intensive benchmarks. This is because the cost of simulating a memory instruction involves the expensive interconnection network model. Execution profiling of XMTSim reveals that for real-life XMTC programs, up to 60% of the time can be spent in simulating the interconnection network. When it comes to the simulated clock cycle throughput, the difference between the memory and computation intensive benchmarks is not as significant, since memory instructions incur significantly more

Benchmark Group	Instruction/sec	Cycle/sec
Parallel, memory intensive	98K	5.5K
Parallel, computation intensive	2.23M	10K
Serial, memory intensive	76K	519K
Serial, computation intensive	1.7M	4.2M

TABLE I
SIMULATED THROUGHPUTS OF XMTSIM.

clock cycles than computation instructions, boosting the cycle throughput.

As mentioned earlier, DT simulation may be considerably faster than DE simulation, most notably when a lot of actions fall in the same exact moment in simulated time. A DT simulator polls through all the actions in one sweep, whereas XMTSim would have to schedule and return a separate event for each one (see Fig. 5), which is a costly operation. A way around this problem is grouping closely related components in one large actor and letting the actor handle and combine events from these components. An example is the *macro-actor* in Fig. 4 (and the implementation of the interconnection network in XMTSim). A macro-actor contains the code for many components and iterates through them at every simulated clock cycle. The action code of the macro-actor resembles the DT simulation code in Fig. 5a except the while loop is replaced by a notify function called by the scheduler. This style is advantageous when the average number of events that would be scheduled per cycle without grouping the components (*i.e.*, each component is an actor) passes a threshold. For a simple experiment conducted with components that contain no action code this threshold was 800 events per cycle. In more realistic cases, the threshold would also depend on the amount of action code.

E. Other Features

In this section we will summarize some of the additional features of XMTSim.

Execution traces. XMTSim generates execution traces at various detail levels. At the functional level, only the results of executed assembly instructions are displayed. The more detailed cycle-accurate level reports the cycle-accurate components through which the instruction and data packages travel. Traces can be limited to specific instructions in the assembly input and/or to specific TCUs.

Floorplan visualization. The amount of simulation output can be overwhelming, especially for a configuration that contains many TCUs. In order to alleviate the complexity of interpreting data, XMTSim can be paired with the floorplan visualization package that is a part of the XMT software release. The visualization package allows displaying data for each cluster or cache module on an XMT floorplan, in colors or text. It can be used as a part of an activity plug-in to animate statistics obtained during a simulation run.

Checkpoints. XMTSim supports simulation checkpoints, *i.e.*, the state of the simulation can be saved at a point that is given by the user ahead of time or determined by a command line interrupt during execution. Simulation can be resumed at a later time. This is a feature which, among other practical uses, can facilitate dynamically load balancing a batch of long simulations running on multiple computers.

F. Features under Development

XMTSim is under active development and following is a list of features that are either currently being tested or are a part of our future road map.

Improved power/temperature estimation and management. The temperature estimation feature of XMTSim made possible a recently submitted companion publication that focuses on establishing the thermal feasibility of a 1024-TCU XMT chip [22]. The power output is computed as a function of the activity counters and passed on to HotSpot [36], an accurate and fast thermal model, for temperature estimation. HotSpot is written in C programming language and incorporated to the simulation flow via the Java Native Interface (JNI) [37]. The future work on this track includes refining the power model of the simulator as well as incorporating efficient thermal and power management algorithms.

Phase sampling. Programs with very long execution times usually consist of multiple phases where each phase is a set of intervals that have similar behavior [38]. An extension to the XMT system can be tested by running the cycle-accurate simulation for a few intervals on each phase and fast-forwarding in-between. Fast-forwarding can be done by switching to a fast mode that will estimate the state of the simulator if it were run in the cycle-accurate mode. Incorporating features that will enable phase sampling will allow simulation of large programs and improve the capabilities of the simulator as a design space exploration tool.

Asynchronous interconnect. Use of asynchronous logic in the interconnection network design might be preferable for its advantages in power consumption. Following up on [39], work in progress with our Columbia University partner compares the synchronous versus asynchronous implementations of the interconnection network modeled in XMTSim.

Increasing simulation speed via parallelism. The simulation speed of XMTSim can be improved by parallelizing the scheduling and processing of discrete-events [40]. It would also be intriguing to run XMTSim as well as the computation hungry simulation of the interconnection network component on XMT itself. We are exploring both.

IV. THE XMTC COMPILER.

Our compiler translates XMTC code to an optimized XMT executable. The compiler consists of three consecutive passes: the *pre-pass* performs source-to-source (XMTC-to-XMTC) transformations and is based on CIL [41], the *core-pass* performs the bulk of the compilation and is based on GCC v4.0, and the *post-pass*, built using SableCC [34] takes the assembly produced by the core-pass, verifies that it complies with XMT semantics and links it with external data inputs. More details than presented hereafter can be found in [42].

A. The XMTC Memory Model

The memory consistency model for a parallel computing environment is a contract between the programmer and the platform, specifying how memory actions (reads and writes) in a program appear to execute to the programmer, and specifically which value each a read of a memory location may return [43].

Consider the example in Fig. 6. If memory store operations are non-blocking, meaning they do not wait for a confirmation that the operation completed, it is possible for Thread B to read {x=0 and y=1}. At first this relaxed consistency is counterintuitive. However, because it allows for much better performance, in this example by allowing multiple pending write operations, it is usually favored over more intuitive but also more restrictive models [43], [44].

```
Initially:   x=0, y=0

Thread A:                          Thread B:

x := 1                             Read y
y := 1                             Read x
```

Possible results read by Thread B: $(x, y) \in \{(0,0),(1,0),(1,1),(1,0)\}$

Fig. 6. Two threads with no order-enforcing operations or guarantees.

The XMT memory model is a relaxed model that allows the same results for Thread B as in the previous example. It relaxes the order of memory operations and only preserves relative ordering with respect to prefix-sum operations (`ps` and `psm`), and to the beginning and end of `spawn` statements. This makes prefix-sum operations important for synchronizing between virtual threads, as will be shown in Fig.7. The XMT memory model gives the programmer two rules about the ordering of (read and write) memory operations.

First, within one virtual thread it guarantees serial execution, which means that a read to a location will return the last value written to that location by the current virtual thread, provided it was not overwritten by a different virtual thread. Intuitively, read or write operations from the same source (TCU) to the same destination (memory address) cannot be reordered, neither by the hardware, nor by the compiler.

Second, for each pair of virtual threads, it guarantees a partial ordering of memory operations relative to prefix-sum operations over the same base. This rule is a bit more involved so we explain it through the example in Fig. 7. This example shows how to code the example of Fig. 6 if we want the invariant *if y=1 then x=1* to hold at the end of Thread B (*i.e.*, disallow $(x, y) = (0, 1)$). Both threads synchronize (in a loose sense) over variable `y` using a `psm` operation; thread A writes (increments) `y`, while Thread B reads it. At run-time one of the two threads executes its `psm` instruction first; the second rule of the XMT memory model guarantees that all memory operations issued before the `psm` of the thread to execute its `psm` first will complete before any memory operation after the `psm` of the second thread is issued. In our example, assume that Thread A completes it's prefix-sum first. That means that the operation `x=1` completed before Thread B reads X, which enforces the desired invariant *if y=1 then x=1* for Thread B.

```
Initially:   x=0, y=0

Thread A:                          Thread B:

x = 1;                             tmpB = 0;
tmpA = 1;                          psm(tmpB, y); // Read y
psm(tmpA, y);  //y++               .. = x        // Read x
```

Possible results read by Thread B: $(x, y) \in \{(0,0),(1,0),(1,1)\}$

Fig. 7. Enforcing partial order in the XMT memory model.

The implementation of these two rules by the hardware and compiler is straightforward. For the first rule, the static hardware routing of messages from TCUs to memory guarantees that the order of operations issued from the same source to the same destination will be preserved. The compiler enforces the second rule by (a) issuing a *memory fence* operation before each prefix-sum operation to wait until all pending writes complete, and by (b) not moving memory operations across prefix-sum instructions. The current implementation does not take into account the base of prefix-sum operations and may be overly conservative in some cases. Using static analysis to reduce the number of memory fences and to selectively allow motion of memory operations across prefix-sums is the subject of future research.

Note that in Fig. 7 both `psm` operations are needed. If, for example, Thread B used a simple read operation for `y` instead of a prefix-sum, prefetching could cause variable `x` to be read before `y` and the invariant *if y=1 then x=1* would not hold.

An implication of the XMT memory model is that register allocation for parallel code is performed as if the code were serial. The programmer, however, must still declare the variables that may be modified by other virtual threads as `volatile`. These variables will not be register allocated, as in the case of with multi-threaded C code. This is rarely needed in user code, but it is useful in library and system code.

B. Compiling Explicitly Parallel Code with a Serial Compiler

Although we have extended the core-pass (GCC) to parse the additional XMTC parallel constructs, it inherently remains a serial C compiler. Changing the internal GCC data structures to express parallelism would have required great effort and all the optimization passes would have had to be updated to account for these new constructs. Such a task was beyond our limited resources and is unnecessary. Instead, a spawn statement is parsed as if there was a `spawn` assembly instruction at the beginning of the code block of the statement (spawn-block) and a `join` at the end of it, as seen in Fig. 8b. Therefore, to GCC, a spawn statement appears as a sequential block of code. This opens the door for illegal dataflow because (a) it hides the fact that the spawn block might be executed multiple times, (b) it hides the concurrency of these multiple executions, and (c) it hides the transfer of control from the Master TCU to the parallel TCUs where the spawn-block is executed. A case of an invalid code transformation caused by illegal dataflow is code-motion across spawn-block boundaries. For example, in Fig. 8a, the compiler might move the conditional increment statement `if(found) counter+=1` inside the spawn-block. The counter could then be incremented multiple times instead of only once, which is illegal.

To prevent illegal dataflow, we implemented *outlining* (also known as *method extraction*) in the CIL pre-pass, an operation akin to the reverse of function inlining. Fig. 8c shows the outlined version of the code in Fig. 8a. Outlining places each spawn statement in a new function and replaces it by a call to this new function. According to XMTC semantics, the spawn statement should have access to the variables in the scope of the enclosing serial section. We detect which of these variables are accessed in the parallel code and whether they might be written to. Then, we pass them as arguments to the outlined function by value or by reference. In Fig. 8c, because the variable `found` is updated in the spawn block, it is passed by reference to the outlined function.

Outlining prevents illegal dataflow without requiring all optimizations to be turned off. This solution works since GCC, like many compilers, does not perform inter-procedural optimizations. Compilers that perform inter-procedural optimizations often provide a *no-inline* flag that has the same effect of preventing inter-procedural code motion.

(a) Original Code

```
int A[N];
bool found=false;
spawn(0,N-1) {
    if (A[$]!=0)
        found = true;
}
if (found) counter+=1;
```

(b) What the compiler sees

```
int A[N];
bool found=false;
asm(spawn 0, N-1);
if (A[$]!=0)
    found = true;
asm(join);
if (found) counter+=1;
```

(c) After Outlining

```
int A[N];
bool found=false;
outl_sp_1 (A, &found);
if (found) counter+=1;
....................

void
outl_sp_1(int (*A),
          bool *found) {
    spawn(0,N-1) {
        if (A[$]!=0)
            (*found) = true;
    }
}
```

Fig. 8. Simple example of outlining. All the elements of array A are read in parallel and if an element is non-zero `found` is set to `true`. After the parallel section, `counter` is incremented if a non-zero element was found.

We now present an other example of illegal dataflow, but this time without code motion. It happens because GCC is unaware of the transfer of control from the serial processor (Master TCU) to the parallel TCUs that a spawn statement entails. GCC optimizations incorrectly assume that a value can be loaded to a register before the spawn statement (within the outlined function) and accessed within the spawn-block. This is false because the value is loaded to a register of the Master TCU but the spawn-block code can only access the TCU registers. There are two ways to fix this problem: (a) move the load instruction back into the spawn-block, but load the value for each virtual thread, or (b) broadcast (an XMT specific operation) all live Master TCU registers to the parallel TCUs at the onset of a virtual thread. We chose the second approach because it conserves memory bandwidth.

(a) Wrong layout by GCC

```
outlined_spawn_1:
    spawn
BB1:
    ...
    bneq $r, $0, BB2
    join
    jr $31   # return
BB2:
    ...
    j BB1
```

(b) Corrected layout

```
outlined_spawn_1:
    spawn
BB1:
    ...
    bneq $r, $0, BB2
    j BBjoin
BB2:
    ...
    j BB1
BBjoin:
    join
    jr $31   # return
```

Fig. 9. Example of assembly basic-block layout issue. (a) Basic Block BB2 belongs logically to the spawn-join section but is placed after the return instruction (jr $31). (b) BB2 is pulled back into the spawn-join by inserting it before the `join` instruction and adding a jump to the join at the end of the preceding basic block.

Finally, XMT places a restriction on the layout of the assembly code of spawn blocks, because it needs to broadcast it to the TCUs: all spawn-block code must be placed between the `spawn` and `join` assembly instructions. Interestingly, in its effort to optimize the assembly, GCC might decide to place a basic-block (a short sequence of assembly instructions) that logically belongs to a spawn-block, after it. In the example in Fig. 9 basic-block 2 (BB2) is placed after the return statement of the outlined_spawn function to save one jump instruction (Fig. 9a). The code is semantically correct but it will lead to incorrect execution because BB2 will not be broadcast by the XMT hardware and TCUs currently don't have access to instructions that were not broadcast. We wrote a pass in SableCC to check for this situation and fix it by relocating such misplaced basic-blocks between the spawn and join instructions, as shown in Fig. 9b. Note that the inclusion of instruction caches at the cluster or TCU level that would future versions of XMT will allow TCUs to fetch instructions that are not in their instruction buffer by implementing instruction caches at the cluster or TCU level.

Why illegal dataflow is not an issue for thread libraries. There are several libraries that are used to introduce parallelism to serial languages (*e.g.*, *Pthreads*). Code written using such library calls are compiled using a serial compiler so one might wonder why illegal dataflow is not an issue in that scenario. In Pthreads the programmer creates an additional thread using the *pthread_create* call which takes a function to execute in the new thread as an argument. In other words, the programmer is forced to do the outlining manually. Moreover, thread libraries do not introduce new control structures in the base language (such as XMTC's spawn statement), so the compiler does not need to be updated. That said, serial compilers can still perform illegal optimizations on Pthreads code [45], but this is rare enough that Pthreads can still be used in practice. However, the main disadvantage to using a thread library is the added complexity for the programmer: creating parallelism through a library API is arguably harder and less intuitive than directly creating it through parallel constructs incorporated in the programming language.

C. XMT Specific Optimizations

Some of the design decisions of XMT create new opportunities for compiler optimizations that were not possible or not needed for other architectures. Novel parallel architectures may adopt similar designs, making these optimizations relevant to them. This section presents some of them.

Latency tolerating mechanisms. The XMT memory hierarchy is designed to allow for scalability and performance. To avoid costly cache coherence mechanisms (in terms of chip resources as well as performance overheads), the first level of cache is shared by all the TCUs, with an access latency in the order of 30 clock cycles for a 1024 TCU XMT configuration. Several mechanisms are included in the XMT architecture to overlap shared memory requests with computation or avoid them: non-blocking stores, TCU-level prefetch buffers and cluster-level read-only caches.

Currently the XMT compiler includes support for automatically replacing eligible writes with non-blocking stores and inserting prefetching instructions to fetch data in the TCU prefetch buffers. Support for automatically taking advantage of the read-only caches is planned for future revisions of the compiler. In the meantime, programmers can explicitly load data into the read-only caches if needed.

The XMT compiler prefetching mechanism was designed to match the characteristics of a lightweight, highly parallel many-core architecture. It has been shown to out-perform state-of-the-art prefetching algorithms such as the one included in the GCC compiler suite, as well as hardware prefetching schemes [8]. The prefetching algorithm is potentially applicable to other many-core platforms.

Virtual Thread clustering. The XMT programming methodology encourages the programmer to express any parallelism, regardless of how fine-grained it may be. The low overheads involved in scheduling virtual threads to cores allow this fine-grained parallelism to be competitive. However, despite the efficient implementation, extremely fine-grained programs can benefit from coarsening (*i.e.*, grouping virtual threads into longer virtual threads), consequently reducing the overall scheduling overhead. Furthermore, thread coarsening may allow to exploit spatial locality through prefetching and reduce duplicate work: combining multiple short virtual threads in a loop usually enables the application of loop prefetching as well as the re-use of computed values between iterations of the loop. The clustering mechanism is described and evaluated in [10] and is currently implemented as an optional optimization pass in the XMT compiler.

D. The XMTC Runtime

In the current release of the XMT toolchain, the XMTC runtime (allocation of work and stack to TCUs and dynamic memory allocation) relies mainly on hardware support. In future versions (see next section), the compiler will play the central role in the runtime due to the planned inclusion of parallel stack allocation and support for scheduling of nested spawn statements.

Because parallel stack allocation is not yet publicly supported, virtual threads can only use registers or global memory for intermediate results. For that reason, the compiler checks if the available registers suffice and produces a *register spill error* otherwise.

The other responsibility of the compiler is to insert code to orchestrate the execution of virtual threads. It inserts a `ps` instruction to allow TCUs to concurrently get the ID of the next available virtual thread and a `chkid` instruction to validate it. If the ID is valid ($\in [low, high]$), the TCU starts executing the virtual thread, but if not, the `chkid` blocks the TCU. If all TCUs are blocked at a `chkid`, then all virtual threads have completed and the hardware returns control to the Master TCU which resumes the serial execution. The combination of code broadcasting, virtual thread allocation with `ps` operations and the barrier-like function of `chkid` allow fine-grained load-balancing and lightweight initialization and termination of parallel sections. These enable XMT to benefit from very small amounts of parallelism [24], while allowing easy programming.

Dynamic memory allocation is currently supported only in serial code as a library call, but the allocated memory can be accessed and modified in parallel code as well. Parallel dynamic memory allocation is an interesting research topic which will be the focus of future work.

E. Features under Development

Some advanced features such as support for a parallel cactus-stack, which allows function calls in parallel code, and support for nested spawn statements (other than serializing inner spawns), are still being debugged and will be included in a future release, but they have already been used in [27], [28]. Implementing support for these features culminated in a scheduling algorithm, *Lazy Binary Splitting* (LBS) [46], that improves on existing work-stealing load-balancing techniques. LBS helps improve the ease-of-programming because it greatly reduces the need to manually coarsen parallelism by adapting parallelism granularity automatically at runtime. At the same time LBS outperforms existing approaches thanks to its runtime adaptability.

V. CONCLUSION

In this paper, we outlined the concept and the design of the current optimizing compiler and the cycle-accurate simulator of the XMT general-purpose many-core computing platform. These publicly available tools have been vital in exploring the design space for the XMT computer architecture, as well as establishing the ease-of-programming and competitive performance claims of the XMT project. This presentation is timely as the stability and usability of the tools have been adequately demonstrated numerous times via publications and teaching. Furthermore, the capabilities of our toolchain extend beyond the scope of the XMT vision. It can be used to explore a much greater design space of shared memory many-cores by a range of researchers such as algorithm developers and system architects.

Acknowledgment

The authors would like to thank Tali Moreshet (Swarthmore), Genette Gill (Columbia U.) and the anonymous reviewers for their constructive comments. Tali Moreshet also provided valuable feedback that helped advancing the toolchain. Partial support by NSF grants 0325393, 0811504, and 0834373 is gratefully acknowledged.

References

[1] M. Snir, "Multi-core and parallel programming: Is the sky falling?" 2008. [Online]. Available: http://www.cccblog.org/2008/11/17/multi-core-and-parallel-programming-is-the-sky-falling/

[2] U. Vishkin, "Using simple abstraction to guide the reinvention of computing for parallelism," *Communications of the ACM*, vol. 54, no. 1, pp. 76–86, January 2011.

[3] "Software release of the XMT programming environment," http://www.umiacs.umd.edu/users/vishkin/XMT/sw-release.html, 2008 – 2010.

[4] S. Torbert, U. Vishkin, R. Tzur, and D. J. Ellison, "Is teaching parallel algorithmic thinking to high-school student possible? one teachers experience," in *Proc. ACM Technical Symposium on Computer Science Education*, 2010.

[5] U. Vishkin, R. Tzur, D. Ellison, and G. C. Caragea, "Programming for high schools," July 2009, keynote,The CS4HS Workshop. Download from http://www.umiacs.umd.edu/~vishkin/XMT/CS4HS_PATfinal.ppt.

[6] L. Hochstein, V. R. Basili, U. Vishkin, and J. Gilbert, "A pilot study to compare programming effort for two parallel programming models," *Journal of Systems and Software*, vol. 81, no. 11, pp. 1920 – 1930, 2008.

[7] D. Padua and U. Vishkin, "Joint UIUC/UMD parallel algorithms/programming course," in *Proc. NSF/TCPP Workshop on Parallel and Distributed Computing Education, in conjunction with IPDPS*, 2011.

[8] G. Caragea, A. Tzannes, F. Keceli, R. Barua, and U. Vishkin, "Resource-aware compiler prefetching for many-cores." in *Proc. Intl. Symposium on Parallel and Distributed Computing*, 2010.

[9] U. Vishkin, S. Dascal, E. Berkovich, and J. Nuzman, "Explicit multi-threading (XMT) bridging models for instruction parallelism (extended abstract)," in *Proc. ACM Symposium on Parallelism in Algorithms and Architectures*, 1998.

[10] D. Naishlos, J. Nuzman, C.-W. Tseng, and U. Vishkin, "Towards a first vertical prototyping of an extremely fine-grained parallel programming approach," in *Proc. ACM Symposium on Parallelism in Algorithms and Architectures*, 2001.

[11] J. JáJá, *An introduction to parallel algorithms*. Redwood City, CA, USA: Addison Wesley Longman Publishing Co., Inc., 1992.

[12] J. Keller, C. Kessler, and J. L. Traeff, *Practical PRAM Programming*. New York, NY, USA: John Wiley & Sons, Inc., 2000.

[13] X. Wen and U. Vishkin, "PRAM-on-chip: first commitment to silicon," in *Proc. ACM Symposium on Parallelism in Algorithms and Architectures*, 2007.

[14] X. Wen and U. Vishkin, "FPGA-based prototype of a PRAM on-chip processor," in *Proc. ACM Computing Frontiers*, 2008.

[15] A. O. Balkan, G. Qu, and U. Vishkin, "A mesh-of-trees interconnection network for single-chip parallel processing," in *Proc. IEEE Intl. Conf. on Application-specific Systems, Architectures and Processors*, 2006.

[16] A. O. Balkan, M. N. Horak, G. Qu, and U. Vishkin, "Layout-accurate design and implementation of a high-throughput interconnection network for single-chip parallel processing," in *Proc. of Hot Interconnects*, 2007.

[17] U. Vishkin, G. C. Caragea, and B. C. Lee, *Handbook of Parallel Computing: Models, Algorithms and Applications*. CRC Press, 2007, ch. Models for Advancing PRAM and Other Algorithms into Parallel Programs for a PRAM-On-Chip Platform.

[18] A. Gottlieb, R. Grishman, C. P. Kruskal, K. P. McAuliffe, L. Rudolph, and M. Snir, "The NYU ultracomputer: designing a MIMD, shared-memory parallel machine (extended abstract)," in *Proc. Intl. Symposium on Computer Architecture*, 1982.

[19] G. C. Caragea, A. Tzannes, A. O. Balkan, and U. Vishkin, "XMT Toolchain Manual for XMTC Language, XMTC Compiler, XMT Simulator and Paraleap XMT FPGA Computer," 2010. [Online]. Available: http://sourceforge.net/projects/xmtc/files/xmtc-documentation/

[20] G. C. Caragea, B. Saybasili, X. Wen, and U. Vishkin, "Performance potential of an easy-to-program pram-on-chip prototype versus state-of-the-art processor," in *Proc. ACM Symposium on Parallelism in Algorithms and Architectures*, 2009.

[21] G. Caragea, F. Keceli, A. Tzannes, and U. Vishkin, "General-purpose vs. GPU: Comparison of many-cores on irregular workloads," in *Proc. USENIX Workshop on Hot Topics in Parallelsim*, 2010.

[22] F. Keceli, T. Moreshet, and U. Vishkin, "Power-performance comparison of single-task driven many-cores," submitted for publication.

[23] T. M. DuBois, B. Lee, Y. Wang, M. Olano, and U. Vishkin, "XMT-GPU: A PRAM architecture for graphics computation," in *Proc. Intl. Conf. on Parallel Processing*, 2008.

[24] A. B. Saybasili, A. Tzannes, B. R. Brooks, and U. Vishkin, "Highly parallel multi-dimentional fast fourier transform on fine- and coarse-grained many-core approaches," in *Proc. IASTED Intl. Conf. on Parallel and Distributed Computing and Systems*, 2009.

[25] G. Cong and D. Bader, "An Experimental Study of Parallel Biconnected Components Algorithms on Symmetric Multiprocessors (SMPs)," in *Proc. IEEE Intl. Parallel and Distr. Processing Symposium*, 2005.

[26] Z. He and B. Hong, "Dynamically Tuned Push-Relabel Algorithm for the Maximum Flow Problem on CPU-GPU-Hybrid Platforms," in *Proc. IEEE Intl. Parallel and Distr. Processing Symposium*, 2010.

[27] J. Edwards and U. Vishkin, "An Evaluation of Biconnectivity Algorithms on Many-Core Processors," 2011, submitted for publication.

[28] G. C. Caragea and U. Vishkin, "Better Speedups for Parallel Max-Flow," 2011, submitted for publication.

[29] F. Keceli and U. Vishkin, "XMTSim: Cycle-accurate Simulator of the XMT Many-Core Architecture," University of Maryland Institute for Advanced Computer Studies, Tech. Rep. UMIACS-TR-2011-02, 2011.

[30] T. Austin, E. Larson, and D. Erns, "Simplescalar: An infrastructure for computer system modeling," *Computer*, vol. 35, no. 2, pp. 59–67, 2002.

[31] P. S. Magnusson, M. Christensson, J. Eskilson, D. Forsgren, G. Hallberg, J. Hogberg, F. Larsson, A. Moestedt, and B. Werner, "Simics: A full system simulation platform," *Computer*, vol. 35, pp. 50–58, 2002.

[32] A. Bakhoda, G. L. Yuan, W. W. L. Fung, H. Wong, and T. M. Aamodt, "Analyzing CUDA workloads using a detailed GPU simulator." in *Proc. IEEE Intl. Symposium on Performance Analysis of Systems and Software*, 2009.

[33] S. Cho, S. Demetriades, S. Evans, L. Jin, H. Lee, K. Lee, and M. Moeng, "TPTS: A novel framework for very fast manycore processor architecture simulation," in *Proc. Intl. Conf. on Parallel Processing*, 2008.

[34] E. M. Gagnon and L. J. Hendren, "Sablecc, an object-oriented compiler framework," in *Proc. Technology of Object-Oriented Languages*, 1998.

[35] J. Banks, J. S. Carson, B. L. Nelson, and D. M. Nicol, *Discrete-Event System Simulation*, 4th ed. Upper Saddle River, NJ, USA: Prentice Hall, 2004.

[36] K. Skadron, M. R. Stan, W. Huang, S. Velusamy, K. Sankaranarayanan, and D. Tarjan, "Temperature-aware microarchitecture," in *Proc. Intl. Symposium on Computer Architecture*, 2003.

[37] S. Liang, *Java Native Interface: Programmer's Guide and Reference*. Addison-Wesley Longman Publishing, 1999.

[38] G. Hamerly, E. Perelman, J. Lau, and B. Calder, "Simpoint 3.0: Faster and more flexible program analysis," *Journal of Instruction Level Parallelism*, 2005.

[39] M. N. Horak, S. M. Nowick, M. Carlberg, and U. Vishkin, "A low-overhead asynchronous interconnection network for GALS chip multiprocessors," in *Proc. ACM/IEEE Intl. Symposium on Networks-on-Chip*, 2010.

[40] R. M. Fujimoto, "Parallel discrete event simulation," *Communications of the ACM*, vol. 33, pp. 30–53, 1990.

[41] G. C. Necula, S. McPeak, S. P. Rahul, and W. Weimer, "CIL: Intermediate Language and Tools for Analysis and Transformation of C Programs," in *Proc. Intl. Conf. on Compiler Construction*, 2002.

[42] A. Tzannes, G. C. Caragea, U. Vishkin, and R. Barua, "The compiler for the XMTC parallel language: Lessons for compiler developers and in-depth description," University of Maryland Institute for Advanced Computer Studies, Tech. Rep. UMIACS-TR-2011-01, 2011.

[43] J. Manson, W. Pugh, and S. V. Adve, "The Java memory model," in *Proc. ACM Symposium on Principles of Programming Languages*, 2005.

[44] S. V. Adve and K. Gharachorloo, "Shared memory consistency models: A tutorial," *Computer*, vol. 29, no. 12, pp. 66–76, 1996.

[45] H.-J. Boehm, "Threads cannot be implemented as a library," in *Proc. Conf. on Programming Language Design and Implementation*, 2005.

[46] A. Tzannes, G. C. Caragea, R. Barua, and U. Vishkin, "Lazy binary-splitting: a run-time adaptive work-stealing scheduler," in *Proc. Symposium on Principles and Practice of Parallel Programming*, 2010.