XMTSim: A Simulator of the XMT Many-core Architecture

Fuat Keceli
Intel Corporation
Hillsboro, OR 97124
fuat.keceli@intel.com

Uzi Vishkin
Department of Electrical
and Computer Engineering
University of Maryland
College Park, MD 20740
vishkin@umd.edu

Abstract

This paper documents the features and the design of XMTSim, the cycle-accurate simulator of the Explicit Multi-Threading (XMT) computer architecture. 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. XMTSim is a primary component in its publicly available toolchain along with an optimizing compiler. 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 goal of ease-of-programming and performance.

The applicability of the XMT simulator 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.

XMTSim is the highly-configurable cycle-accurate simulator of the XMT computer architecture [38, 39, 48, 49]. It is tuned to approximate the behavior of major on-die components of XMT, such as the cores, interconnect and on-chip caches. Additionally XMTSim features a power model and a thermal model, and it provides means to simulate dynamic power and thermal management algorithms. We made XMTSim publicly available as a part of the XMT programming toolchain [7, 9], which also includes an optimizing compiler [45]. Detailed information on XMT architecture and the programming model can be found in [30].

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. XMTSim is useful to a range of communities such as system architects, teachers of parallel programming and algorithm developers due to the following four reasons:

1. Opportunity to evaluate alternative system components. XMTSim 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 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, Caragea, et. al [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 architectural mechanisms/features, 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.

2. Performance advantages of XMT and PRAM algorithms. A number of publications [6, 10, 15, 16, 17, 18, 19, 41] list the performance advantages of XMT compared to exiting parallel architectures, and also document the interest of the academic community in such results. XMTSim was the enabling factor for 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.

3. Teaching and experimenting with on-chip parallel programming. As a part of the XMT toolchain, XMTSim contributed to the experiments that established the ease-of-programming of XMT. These experiments were presented in publications [23, 40, 44, 46] and conducted in courses taught to graduate, undergraduate, high-school and middle-school students including at Thomas Jefferson High School, Alexandria, VA. The curriculum at Thomas Jefferson High School has featured XMT programming since 2008; more than two hundred of its students have already programmed XMT and in 2012 forty of these high-school students demonstrated Ph.D. level parallel programming [14]. In addition, the XMT toolchain provides convenient platform for teaching parallel algorithms and programming, because students can install and use it on any personal computer to work on their assignments.

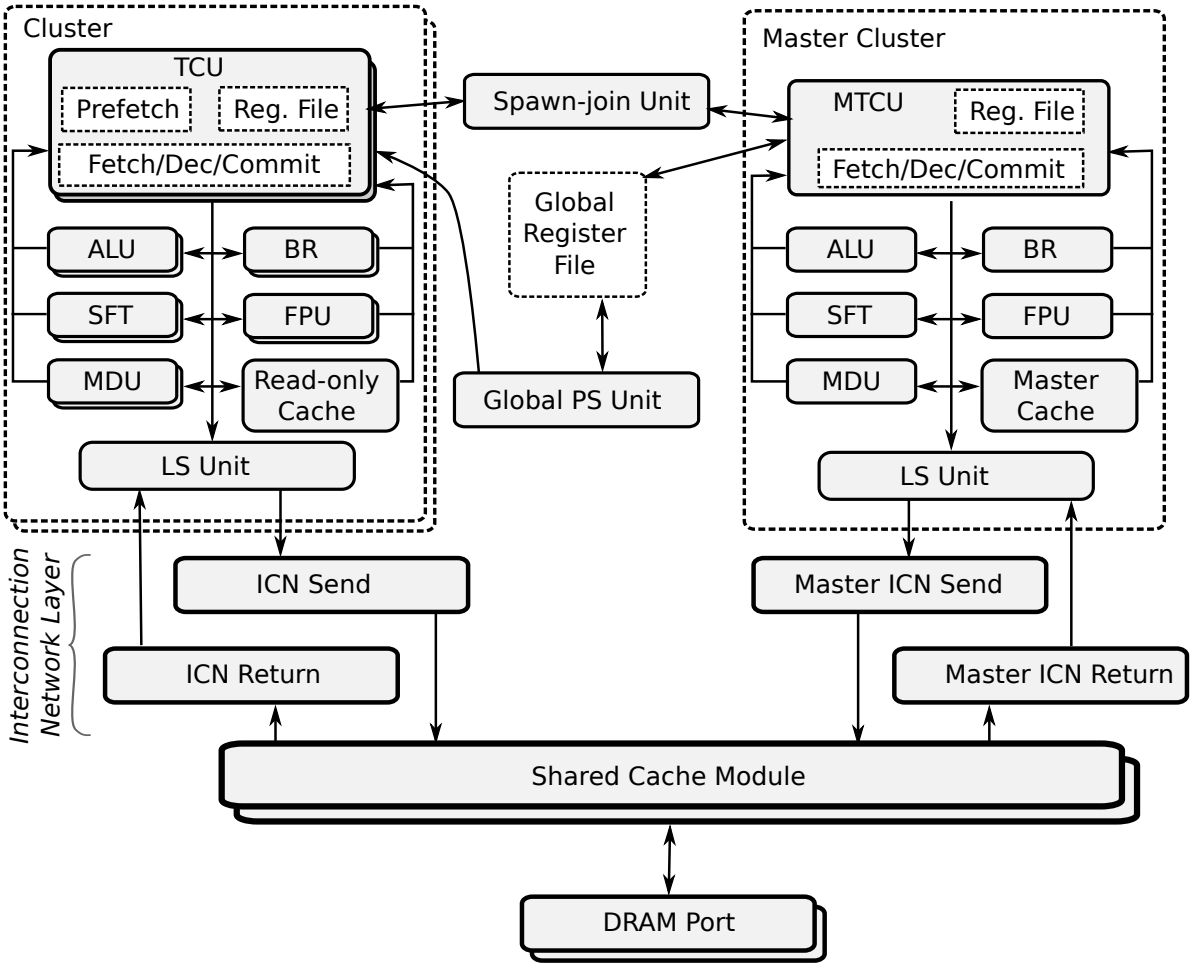

Figure 1: XMT overview from the perspective of XMTSim software structure.

4. Guiding researchers for developing similar tools. This paper also documents our experiences on constructing a simulator for a highly-parallel architecture, which, we believe, will guide other researchers who are in the process of developing similar tools.

The remainder of this paper is organized as follows. Section 1 gives an overview of the simulator. Section 2 elaborates on the mechanisms that enable users to customize the reported statistics and modify the execution of the simulator during runtime. Sections 3 and 4 describe the details of the cycle-accurate simulation and present the cycle verification against the FPGA prototype. Power and thermal models are explained in Section 5 and the dynamic management extensions are explained in Section 6. Sections 8 and 9 list the miscellaneous features that are not mentioned in other sections and possible improvements.

1 Overview of XMTSim

XMTSim accurately models the interactions between the high level micro-architectural components of XMT shown in Figure 1, *i.e.*, the Thread Control Units (TCUs, lightweight processing cores), 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 [49].

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 the 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

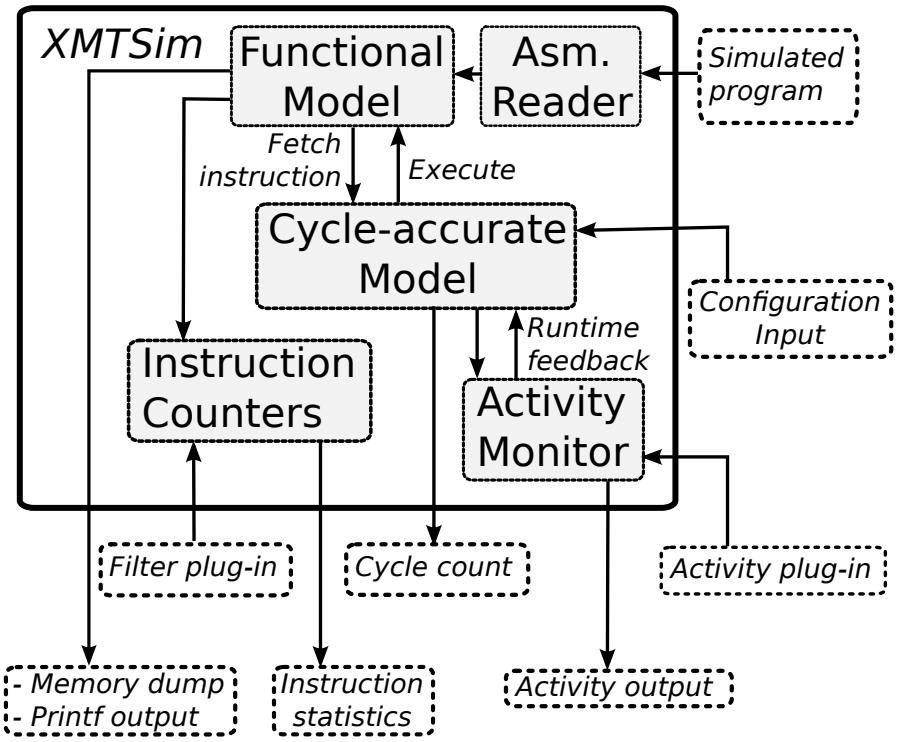

Figure 2: Overview of the simulation mechanism, inputs and outputs.

(ALU, memory, etc.).

Each solid box in Figure 1 corresponds to a Java object in XMTSim. Simulated assembly instruction instances are wrapped in objects of type *packet*. An instruction packet 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 packets that travel through it. In most cases, the specific amount of the delay depends on the previous packets that entered the component. In other words, these components are state machines, where the state input is the instruction/data packets 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 Figure 1 denote either the auxiliary classes that help store the state or the classes that enclose collections of other classes.

Figure 2 is the conceptual overview of the simulation mechanism. The inputs and outputs are outlined with dashed lines. A simulated program consists of assembly and memory map files that are typically provided by 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 [21]. 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 Figure 2.

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 faster by orders of magnitude 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 parallel program since it serializes the execution of the parallel code blocks, called spawn blocks in XMT terminology. 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.

2 Simulation Statistics and Runtime Control

As shown in Figure 2, 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. In Section 5, we will provide more information on the power/thermal model and management in XMTSim.

3 Details of Cycle-Accurate Simulation

In this section, we explain various aspects of how cycle-accurate simulation is implemented in XMTSim, namely the simulation strategy, which is discrete-event based, and the communication of data between simulated components. We then discuss the factors that affect the speed of simulation. Finally, we demonstrate discrete-event simulation on an example.

3.1 Discrete-Event Simulation

Discrete-event (DE) simulation is a technique that is often used for understanding the behavior of complex systems [4]. 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 9.

The building blocks of the DE simulation implementation in XMTSim are *actors*, which are objects that can schedule *events*. Events are scheduled at the *DE scheduler*, which maintains a chronological order of events in an *event list*. An actor is *notified* by the DE scheduler via a callback function when the time of an event it previously scheduled expires, and as a result the actor executes its action code. Some of the typical actions are to schedule another event, trigger a state change or move data between the cycle-accurate components. 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.

Figure 3 is an example of how actors schedule events and are then notified of events. 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.

It should be noted that XMTSim diverges from discrete-time (DT) architecture simulators such as SimpleScalar [2]. The difference is illustrated in Figure 4. The DT simulation runs in a loop that polls all the modeled components and increments the simulated time at the end of each iteration. Simulation ends when a certain criterion 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 in modeling small to medium sized systems. Only the former is a concern in our case and we elaborate further on simulation speed issues in Section 3.4.

A brief comparison of discrete-time versus discrete-event simulation is given in Table 1. As indicated in the table, DT simulation is preferable for simulation of up to mid-size synchronous systems, and the resulting code is often more compact compared to

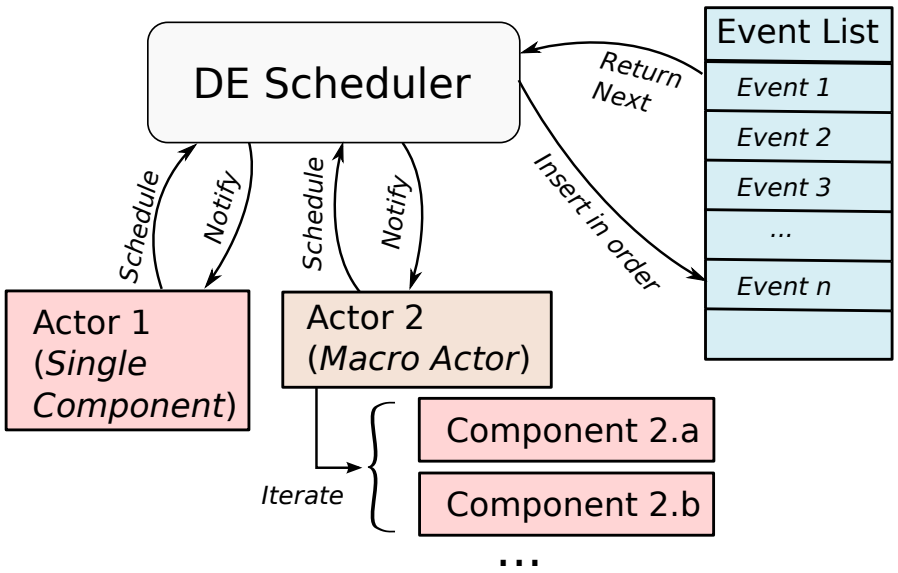

Figure 3: The overview of DE scheduling architecture of the simulator.

```
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)

Figure 4: Main loop of execution for (a) Discrete-time simulation, (b) Discrete-event simulation.

the DE simulation code. For larger systems DT simulation might require an extensive case study for ensuring correctness. Also, for the cases in which a lot of components are defined but only few of them are active every cycle, DT simulation typically wastes computation time on the conditional statements that do not fall through. Advantages of DE simulation were discussed earlier in this section. The primary concern about DE simulation is its performance, which may fall behind DT simulation as demonstrated in the next section.

3.2 Concurrent Communication of Data Between Components

In DE simulation, if the movement of data between various related components is triggered by concurrent events, special care should be paid to ensure correctness of simulation. As a result the DE simulation might require more work than DT simulation. We demonstrate this statement on an example for simulating a simple pipeline. We first show how the simulation is executed on a DT simulator, as it is the simpler case, and then move to the DE simulator.

Figure 5 illustrates how a 3 stage pipeline with a packet at each stage advances one clock cycle in case of no stalls. Figure 5(a) is the initial status. Figures 5(b), 5(c) and 5(d) show the steps that the simulation takes in order to emulate one clock cycle. In the first step, the packet at the last stage (packet 3) is removed as the output. Then packets 2 and 1 are moved to the next stage, in that order. By starting at the end, it is ensured that packets are not unintentionally overwritten.

Figure 6 shows the same 3 stage pipeline example of Figure 5 in DE simulation. We assume that each stage of the pipeline is defined as an actor. For advancing the pipeline, each actor will schedule an event at time T to pass its packet to the next stage. In DE simulation, however, there is no mechanism to enforce an order between the notify calls to the actors that schedule events for the same time (i.e., concurrent events). For example, the actors can be notified in the order of stages 1, 2 and 3. As the figure exhibits, this will cause accidental deletion of packets 2 and 3.

Figure 7 repeats the DE simulation but this time with intermediate storage for each pipeline stage, which is denoted by smaller white boxes in Figures 7(b) and 7(c). For this solution to work, we also have to incorporate the concept of *priorities* to the simulation. We define two priorities, *evaluate* and *update*. The event list is ordered such that evaluate events of a time instant

Table 1: Advantages and disadvantages of DE vs. DT simulation.

	Discrete Time Simulation	Discrete Event Simulation
Pros	·Efficient if a lot of work done for every simulated cycle ·More compact code for smaller simulations	·Naturally suitable for an object-oriented structure ·Can simulate asynchronous logic ·More flexible in quantization of simulated time
Cons	·Requires complex case analysis for a large simulator ·Slow if not all components do work every clock cycle	·Event list operations are expensive ·Might require more work for emulating one clock cycle

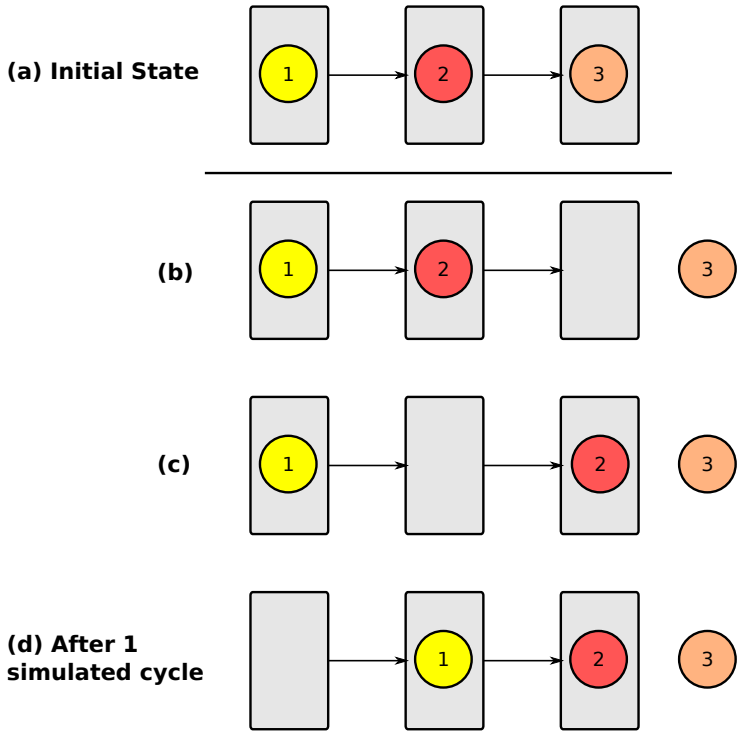

Figure 5: Example of discrete time pipeline simulation.

in simulation come before the update events of the same instant. In the example, at `T-1` (initial state, Figure 7(a)) all actors schedule events for `T.evaluate`. At the evaluate phase of T, `T.evaluate` (Figure 7(b)), they move packets to intermediate storage and schedule events for `T.update`. At the update phase of T, `T.update` (Figure 7(c)), they pass the packets to the next stage.

Next, we compare the work involved in simulating the 3-stage pipeline in DT and DE systems. In DT simulation, 3 move operations are performed to emulate one clock cycle. In DE simulation, 6 move operations and 6 events are required. Clearly, DE simulation would be slower in this example not only because of the number of move operations but also the creation of events is expensive, since they have to be sorted when they are inserted to the event list. This example supports the simulation speed argument in Table 1.

3.3 Optimizing the DE Simulation Performance

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 Figure 4), 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 Figure 3. 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 Figure 4a except the while loop is replaced by a callback from 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.

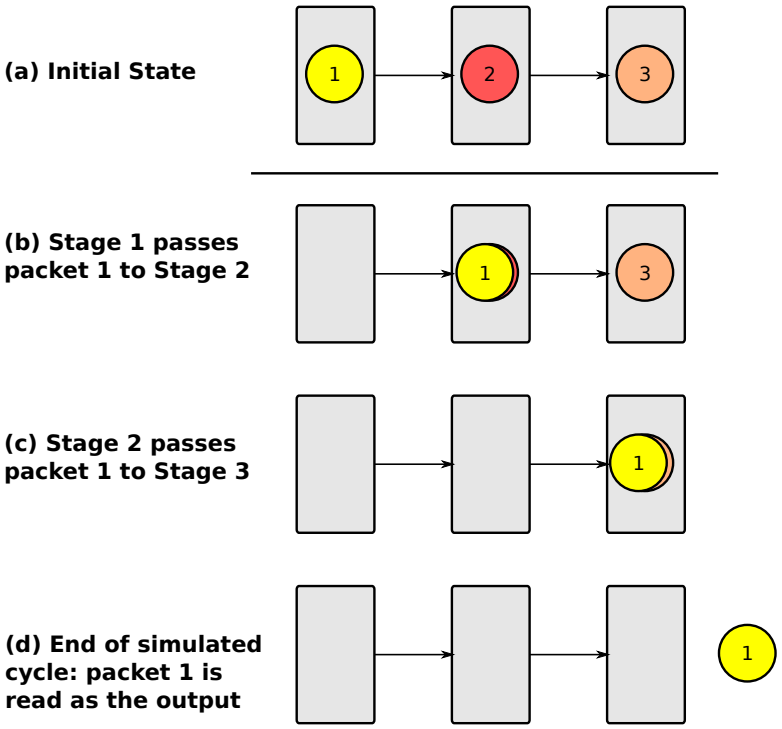

Figure 6: Example of discrete-event pipeline simulation. Simulation creates wrong output as the order of notify calls to actors cause packets to be overwritten.

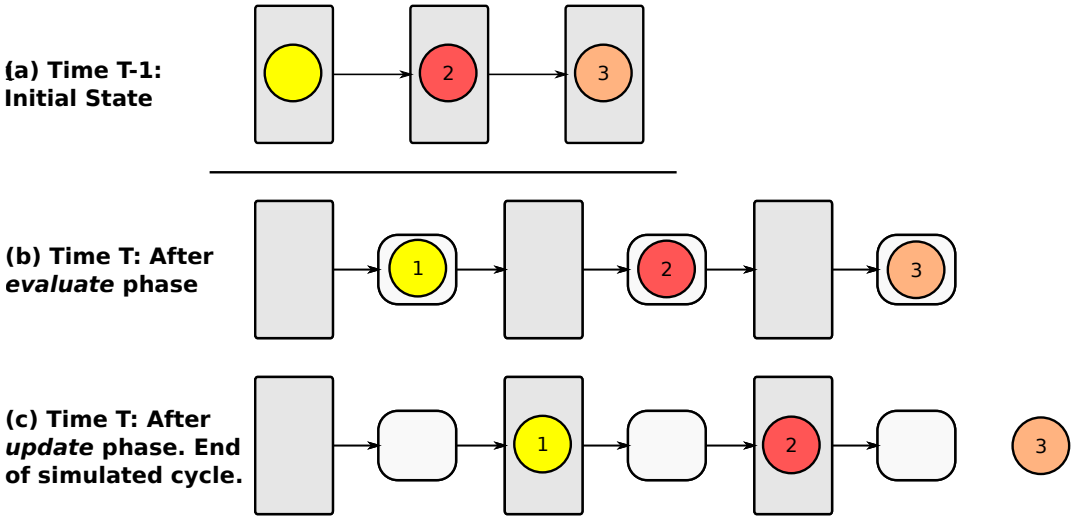

Figure 7: Example of discrete-event pipeline simulation with the addition of priorities. Intermediate storage is used to prevent accidental deletion of packets.

In XMTSim, clusters and shared caches are designed as *macro-actor*s, as well as each of the interconnection network (ICN) send and return paths. This organization not only improves performance, it also facilitates maintainability of the simulator code and provides the convenient means to replace any component by an alternative model, if needed. We define the following mechanism to formalize the coding of macro-actors.

Ports: A macro-actor accepts inputs via its `Port` objects. Port is a Java interface class which is defined by the two methods: (a) *available():* returns a boolean value which indicates that a port can be written to, (b) *write(obj):* accepts an object as its parameter, which should be processed by the actor; can only be called if `available` method returns true.

2-phase simulation: The phases refer to the priorities (*evaluate* and *update*) that were defined in the previous section. The evaluate phase of a simulation instant is the set of all events with evaluate priority at that instant. The update phase is defined similarly. Below are the rules for coding a macro-actor within the 2-phase framework.

Table 2: Simulated throughputs of XMTSim.

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

1. The available and write methods can only be called during the evaluate phase.
2. The output of the available method should be stable during the evaluate phase until it is guaranteed that there will be no calls to the write method of the port.
3. The write method should not be called if a prior call to available for the same simulation instant returns false.
4. If the write method of a port is called multiple times during the evaluate phase of a simulation instant, it is not guaranteed that any of the writes will succeed. Typically the write method should be called at most once for a simulation instant. However, specific implementations of the Port interface might relax this requirement, which should be noted in the API documentation for the class.
5. Typical actions of a macro-actor in the update phase are moving the inputs away from its ports, updating the outputs of the available methods of the ports, and scheduling the evaluate event for the next clock cycle (in XMTSim, evaluate phase comes before the update phase). However, these actions are not requirements.

An example implementation of a MacroActor is given in Figure 8.

3.4 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 2. 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 clock cycles than computation instructions, boosting the cycle throughput.

4 Cycle Verification Against the FPGA Prototype

We validated the cycle-accurate model of XMTSim against the 64-core FPGA XMT prototype, Paraleap. The configuration of XMTSim that matches Paraleap is given in Table 3. In addition to serving as a proof-of-concept implementation for XMT, Paraleap was also set up to emulate the operation of a 800MHz XMT computer with a DDR2 DRAM. The clock of the memory controller was purposefully slowed down so that its ratio to the core clock frequency matches that of the emulated system. The simulator configuration reflects this adjustment.

Even though XMTSim was based on the hardware description language description of Paraleap, discrepancies between the two exist:

- Due to its development status, certain specifications of Paraleap do not exactly match those of the envisioned XMT chip modeled by XMTSim. Given the same amount of effort and on-chip resources that are put towards an industrial grade ASIC product (as opposed to a limited FPGA prototype), these limitations would not exist. Some examples are:
  - Paraleap is spread over multiple FPGA chips and requires additional buffers at the chip boundaries which add to the ICN latency. These buffers are not necessary for modeling an ASIC XMT chip, and are not included in XMTSim.
  - Due to die size limitations, Paraleap utilizes a butterfly interconnection network instead of the MoT used in XMTSim.

```
class ExampleMacroActor extends Actor {
  // The only input port of the actor. It takes objects of type
  // InputJob as input.
  Port<InputJob> inputPort;

  // Temporary storage for the input jobs passed via inputPort.
  InputJob inputPortIn, inputPortOut;

  // Constructor -- Contains the initialization code for a new
  // object of type ExampleMacroActor.
  ExampleMacroActor() {
    inputPort = new Port<InputJob>() {
      public void write(InputJob job) {
        inputPortIn = job;
        // Upon receiving a new input, actor should make sure
        // that it will receive a callback at the next update
        // phase. That code goes here.
      }
      public boolean available() {
        return inputPortIn == null;
      }
    }
  }

  // Implementation of the callback function (called by the scheduler).
  // Event object that caused the callback is passed as a parameter.
  void notifyActor(Event e) {
    switch(e.priority()) {
      case EVALUATE:
        // Main action code of the actor, which processes
        // inputPortOut. The actor might write to the ports of
        // other actors. For example:
        // if(anotherActor.inputPort.available())
        //     anotherActor.inputPort.write(...)
        // Actor schedules next evaluate phase if there is more
        // work to be done.
        break;
      case UPDATE:
        if(inputPortOut == null & inputPortIn != null) {
          inputPortOut = inputPortIn;
          inputPortIn = null;
        }
        // Here actor schedules the next evaluate phase, if there
        // is more work to be done.
        // For example:
        //   scheduler.schedule(new Event(scheduler.time + 1),
        //                                              Event.EVALUATE)
        break;
    }
  }
}
```

Figure 8: Example implementation of a MacroActor.

Table 3: The configuration of XMTSim that is used in validation against Paraleap.

Principal Computational Resources	
Cores	8 clusters, each with 8 TCUs 32-bit RISC ISA 5 stage pipeline (4th stage may be shared and variable length)
Integer Units	64 ALUs (one per TCU), 8 MDUs and 8 FPUs (one each per cluster)
On-chip Memory	
Registers	8 KB integer and 8 KB FP (32 integer and 32 FP reg. per TCU)
Prefetch Buffers	1 KB (4 buffers per TCU)
Shared caches	256 KB total (8 modules, 32 KB each, 2-way associative, 8 word lines)
Read-only caches	64 KB (8 KB per cluster)
Global registers	8 registers
Other	
Interconnection Network (ICN)	8 x 8 Mesh-of-Trees
Memory controllers	1 controller, 32-b (1 word) bus width
Clock frequency	ICN, shared caches and the cores run at the same frequency. Memory controllers and DRAM run at 1/4 of the core clock to emulate the core-to-memory controller clock ratio of the FPGA.

  - In Paraleap, an idle TCU constantly executes a poll instruction in order to detect the beginning of a new parallel thread. A sleep-wake mechanism, where the TCU is woken by an external signal, would arguably be more power efficient. XMTSim makes this assumption in the modeling of TCUs. However, the difference between the two implementations would not cause more than a few clock cycles of difference per parallel spawn block.

- Our experiences show that some implementation differences do not cause a significant cycle-count benefit or penalty however their inclusion in the simulator would cause code complexity and slow down the simulation significantly (as well as requiring a considerable amount of development effort). Note that one of the major purposes of the XMT simulator is architectural exploration and therefore the simulation speed, code clarity, modularity, self documentation and extensibility are important factors. Going into too much detail for no clear benefit conflicts with these objectives. For example, some of the cycle-accurate features of the Master TCU are currently under development. The benchmarks that we use in our experiments usually have insignificant serial sections therefore the inefficiencies of the master TCU should not affect simulation results significantly.

- XMTSim models DRAM communication as constant latency. An accurate external DRAM model would be beneficial to include in XMTSim [47]. Beside improving accuracy, this would allow for testing new DRAM technologies, such as DDR3, not supported by Paraleap, as well as exploring different design choices for the DRAM controller.

- Currently XMT does not feature an OS, therefore I/O operations such as printf's and file operations cannot be simulated in a cycle-accurate way.

Another difficulty in validating XMTSim against Paraleap is related to the indeterminism in the execution of parallel programs. A parallel program can take many execution paths based on the order of concurrent reads or writes (via prefix-sum or memory operations in XMT). If the program is correct it is implied that all these paths will give correct results, however the cycle or assembly instruction count statistics will not necessarily match between different paths. The order of these concurrent events is arbitrary and there is no reliable way to determine if Paraleap and XMTSim will take the same paths if more than one path is possible. For this reason, a benchmark used for validation purposes should guarantee to yield near identical cycle and instruction counts for different execution paths.

Table 4 lists the first set of micro-benchmarks we used in verification. Each benchmark is hand-coded in assembly language for stressing a different component of the parallel TCUs as described in the table. The difference in cycle counts is calculated as

$$Difference = \frac{(CYC_{sim} - CYC_{fpga})}{CYC_{fpga}} \times 100 \quad (1)$$

where CYC_{sim} and CYC_{fpga} are the cycle counts obtained on the simulator and Paraleap, respectively. These benchmarks fulfill the determinism requirement noted earlier and the only significant deviations in cycle counts are observed for the $MicroPar4$ benchmark (33%) and the $MicroPar6$ benchmark (26%). The deviation in the former can be explained by the differences between the interconnect structures and the DRAM models of XMTSim and Paraleap. Latter benchmark contains a significant serial section and execution of serial code on XMTSim is not accurate as mentioned earlier.

Table 4: Microbenchmarks used in cycle verification

Name	Description	Cycles		
		Paraleap	XMTSim	Diff.
MicrPar0	Start 1024 threads and for each thread run a 50000 iteration loop with a single add instruction in it.	1600513	1600327	<1%
MicrPar1	Start 102400 threads and for each thread issue a sw instruction to address 0.	204943	204918	<1%
MicrPar2	Start 1024 threads and for each thread run a 18000 iteration loop with an add and a mult instruction in it.	3456482	3456318	<1%
MicrPar3	Start 1024 threads and for each thread run a 150 iteration loop with an add and a sw to address 0 instruction in it.	307349	307710	<1%
MicroPar4	Start 1024 threads and for each thread run a 1800 iteration loop with a sw instruction (and wrapper code) in it. Sw instructions from different TCUs will be spread across the memory modules.	935225	626908	-33%
MicroPar5	Start 1024 threads and for each thread run a 10K iteration loop with an add, a mult and a divide instruction in it.	8320486	8320318	<1%
MicroSer6	Measure the time to execute a starting and termination of 200K threads.	6226029	4587533	-26%

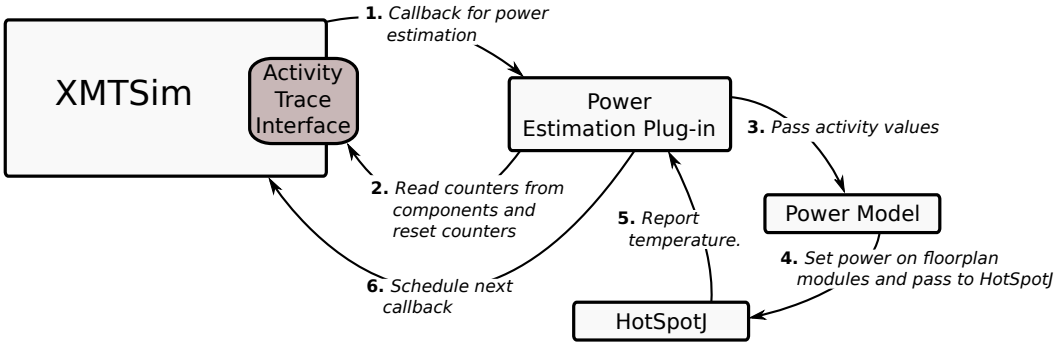

Figure 9: Operation of the power/thermal-estimation plug-in.

5 Power and Temperature Estimation in XMTSim

Power and temperature estimation in XMTSim is implemented using the activity plug-in mechanism. (The activity plug-in mechanism was first mentioned in Section 2). XMTSim contains a power-temperature (PTE) plug-in for a 1024 TCU configuration by default. Computation of the power model parameters for this configuration is beyond the scope of this paper but can be found in [30, 31, 32]. Other configurations can easily be added, however power model parameters should also be created for the new configurations.

As the thermal model, we incorporated HotSpot [25,27,42]. HotSpot is written in the C language and in order to make it available to XMTSim we created HotSpotJ, a Java Native Interface (JNI) [36] wrapper for HotSpot. HotSpotJ is available as a part of XMTSim but it is also a standalone tool and can be used with any Java based simulator. We extended HotSpotJ with a floorplan tool, FPJ, that we use as an interface between XMTSim and HotSpotJ. FPJ is essentially a hierarchical floorplan creator, in which the floorplan blocks are represented as Java objects. A floorplan is created using the FPJ interface and passed to the simulator at the beginning of the simulation. During the simulation it is used as a medium to pass power and temperature data of the floorplan blocks between XMTSim and HotSpotJ. More information on HotSpotJ (and FPJ) will be given in Appendix 7.

Figure 9 illustrates the working of XMTSim with the PTE plug-in. Steps of estimation for one sampling interval are indicated on the figure. XMTSim starts execution by scheduling an initial event for a callback to the PE plug-in. When the PE plug-in receives the callback, it interacts with the *activity trace interface* to collect the statistics that will be explained in the next section and resets the associated counters. Then, it converts these statistics, also called *activity traces*, to power consumption values according to the power model. It sets the power of each floorplan module on the floorplan object and passes it to HotSpotJ. HotSpotJ computes temperatures for the modules. Finally, the PTE plug-in schedules the next callback from the simulator. Users can create their own plug-ins with models other than the one that we will explain next, as long as the model can be described in terms of the statistics reported by the activity trace interface.

5.1 The Power Model

XMTSim uses the framework proposed by Martonosi and Isci [28] to estimate the total power of the system from its dynamic and static power components. According to the model, the simulation provides the access rate for each component (C_i), which

is a value between 0 and 1. The power of a component is a linear function of the access rate with a constant offset.

$$\begin{aligned} Power(C_i) = & AccessRate(C_i) \cdot \\ & MaxActPower(C_i) + \\ & Const(C_i) \end{aligned} \tag{2}$$

C is the set of microarchitectural components for which the power is estimated. We will give the exact definition of $AccessRate(C_i)$ shortly. $MaxActPower(C_i)$ is the upper bound on the power that is proportional to the activity and $Const(C_i)$ is the power of a component which is spent regardless of its activity.

The total power of a component, which is the sum of its dynamic power and leakage power, is expressed as:

$$P = P_{dyn,max} \cdot ACT \cdot CF + DUTY_{clk} \cdot P_{dyn,max} \cdot (1 - CF) + DUTY_V \cdot P_{leak,max} \tag{3}$$

The configuration parameters in the simulation are $P_{dyn,max}$ and $P_{leak,max}$, which are the maximum dynamic and leakage powers and CF, which is the activity correlation factor.

ACT is identical to $AccessRate(C_i)$ above, which is the average activity of a the component for the duration of the sampling period and obtained from simulation. XMTSim utilizes internal counters that monitor the activity of each architectural component. We will discuss the definition of activity on a per component basis in the remainder of this section. $DUTY_{clk}$ and $DUTY_V$ are the clock and voltage duty cycles (i.e., the fraction of the time that the clock and the power supply of a component are active – see [30] for details). Note that $DUTY_V$ is always greater than $DUTY_{clk}$ since voltage gating implies that clock is also stopped.

If we assume that no voltage gating or coarse grain clock gating is applied (i.e., both duty cycles are 1), Equation (3) can be simplified to:

$$P = P_{dyn,max} \cdot ACT \cdot CF + P_{dyn,max} \cdot (1 - CF) + P_{leak,max} \tag{4}$$

In this form, the $P_{dyn,max} \cdot ACT \cdot CF$ and the $P_{dyn,max} \cdot (1 - CF) + P_{leak,max}$ terms are the equivalents of $MaxActPower(C_i)$ and $Const(C_i)$ in Equation (2), respectively.

If CF is less than 1, $Const(C_i)$ not only contains the leakage power but, also contains a part of the dynamic power. A common value to set the activity correlation factor (CF) for aggressively fine-grained clock gated circuits is 0.9, which is the same assumption as Wattch power simulator [5] uses.

Next, we provide details on the activity models of the microarchitectural components in XMTSim. Parameters required to convert the activity to power values can be obtained using tools such as McPAT 0.9 [35] and Cacti 6.5 [37, 50].

Computing Clusters. The power dissipation of an XMT cluster is calculated as the sum of the individual elements in it, which are listed below: The access rate of a TCU pipeline is calculated according to the number of instructions that are fetched and executed, which is a simple but sufficiently accurate approximation. For the integer and floating point units (including arbitration), access rates are the ratio of their throughputs to the maximum throughput. The remainder of the units are all memory array structures and their access rates are computed according to the number of reads and writes they serve.

Memory Controllers, DRAM and Global Shared Caches. The access rate of these components are calculated as the ratio of the requests served to the maximum number of requests that can be served over the sampling period (i.e. one request per cycle).

Global Operations and Serial Processor. We omit the power spent on global operations, since the total gate counts of the circuits that perform these operations were found to be insignificant with respect to the other components, and these operations make up a negligible portion of execution time. In fact, prefix-sum operations and global register file accesses make up less than 1.5% of the total number of instructions among all the benchmarks. We also omit the power of the XMT serial processor, which is only active during serial sections of the XMTC code and when parallel TCUs are inactive. None of our benchmarks contain significant portions of serial code: the number of serial instructions, in all cases, is less than 0.005% of the total number of instructions executed.

Interconnection Network The power of the Mesh-of-Trees (MoT) ICN includes the total cost of communication from all TCUs to the shared caches and back. The access rate for the ICN is equal to its throughput ratio, which is the ratio of the packets transferred between TCUs and shared caches to the maximum number of packets that can be transferred over the sampling period.

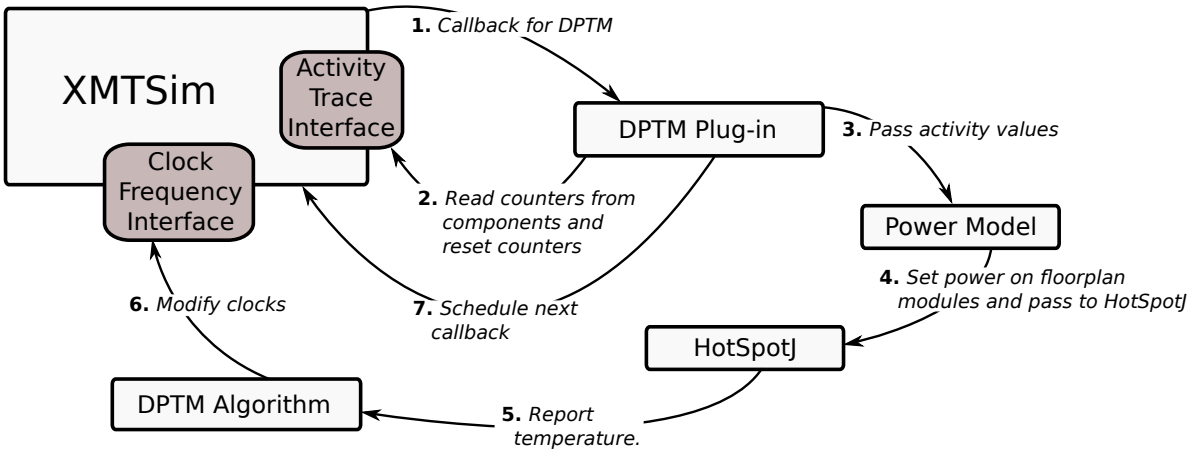

Figure 10: Operation of a DPTM plug-in.

The power cost of the ICN can be broken into various parts [29], which fit into the framework of Equation (2) in the following way. The power spent by the reading and writing of registers, and the charge/discharge of capacitive loads due to wires and repeater inputs can be modeled as proportional to the activity (i.e. number of transferred packets). All packets travel the same number of buffers in the MoT-ICN, and the wire distance they travel can be approximated as a constant which is the average of all possible paths. The power of the arbiters is modeled as a worst-case constant, as it is not feasible to model it accurately in a fast simulator.

6 Dynamic Power and Thermal Management in XMTSim

Dynamic Power and Thermal Management (DPTM) in XMTSim works in a similar way to the PTE plug-in with the addition of a DPTM algorithm stage. The DPTM algorithm changes the clock frequencies of the microarchitectural blocks in reaction to the state of the simulated chip with respect to the constraints.

Figure 10 shows the changes to the PTE plug-in. The two mechanisms are identical up to step 5, at which point the PTE plug-in finishes the sampling period, while the DPTM plug-in modifies the clocks according to the chosen algorithm. The clocks are modified via the standardized API of the *clocked* components.

7 HotSpotJ

HotSpotJ is a java interface for HotSpot [25, 27, 42], an accurate and fast thermal model, which is typically used in conjunction with architecture simulators. Even though we use HotSPotJ with XMTSim, it can be used by any other Java based simulator.

HotSpot is written in C and so far has been available for use with C based simulators. We have originally developed HotSpotJ as an Application Programming Interface (API) to bridge between HotSpot and Java based architecture simulators and eventually it became a supporting tool that enhances the workflow with HotSpot by offering features such as alternative input forms, a floorplan GUI that can display color coded temperature and power values, etc.

The current version of HotSpotJ is based on HotSpot version 4.1. This documentation assumes that readers are familiar with the concepts of HotSpot.

7.1 Summary of Features

With HotSpotJ one can:

- Create floorplans in an object oriented way with Java or directly in a text file (using the FLP format of HotSpot),
- View a floorplan in the GUI and save the floorplan as an image or a FLP file,
- Run steady or transient temperature analysis experiments on a floorplan on the command line – this feature uses HotSpot as the analysis engine,

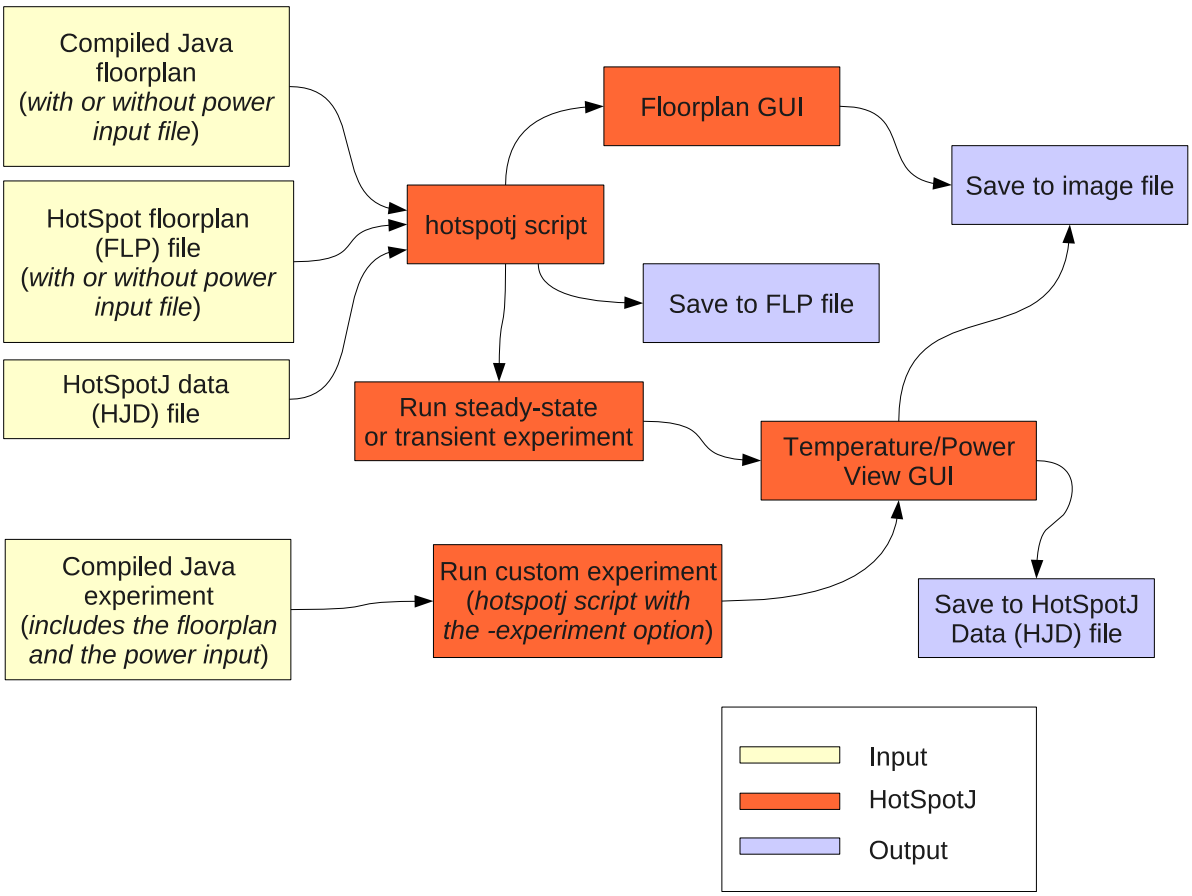

Figure 11: Workflow with the command line script of HotSpotJ.

- Interface with a Java based cycle-accurate simulator in order to feed the HotSpot engine with power values for an experiment,
- View the results of an experiment in the temperature/power viewer GUI, save the results as image files or data files that can later be opened in the GUI.

A noteworthy item in this list is the methodology to express a HotSpot floorplan with object oriented Java programming, which is particularly useful in constructing repetitive floorplans that contain a large number of blocks. HotSpotJ API defines methods to construct hierarchical blocks, which can be replicated and shifted to fill a 2-D grid. As a result, a floorplan that contains a few thousand blocks can easily be expressed under a hundred lines of Java code.

There are two ways that you can use the HotSpotJ software. First is to call the hotspotj script to process your input, which is in the form of a compiled Java floorplan/experiment or text files describing the floorplan and power consumption. The workflow with this option is shown in Figure 11. Second is to write your own Java executable that utilizes the HotSpotJ API, in which case all the functionality of the first option (and more) is provided in the form or function calls. Incorporating HotSpotJ into your cycle-accurate simulator falls into the second category.

7.2 HotSpotJ Terminology

In HotSpotJ, the building blocks of a floorplan are called *simple boxes*. A simple box is identified by its location, dimensions and power consumption value. While location and dimensions are immutable, power consumption value can change over time in the context of a transient experiment with consecutive runs.

In order to introduce the concept of hierarchy, HotSpotJ API defines *composite boxes* which can contain other simple or composite boxes. We refer the highest level composite box in the hierarchy as the *floorplan*. When a box is nested in a composite box, it is said to be *added* to a *parent*. The hierarchy graph of a floorplan should always be a connected tree, i.e. each box should have exactly one parent except the floorplan which has none. On the other hand, there is no limit to the number of children a composite box can have. Hierarchical boxes can be cloned at different locations, which allows for easy construction of repetitive floorplans with a large number of elements.

The hierarchy concept of HotSpotJ is just an abstraction for convenience. Internally, a floorplan is stripped off its hierarchical structure (in other words flattened) before it is passed to the HotSpot engine for temperature analysis. Therefore attributes such as location and color are not relevant for composite boxes.

A simple box has the following attributes:

- **Name** Box names do not need to be unique. Each simple box is assigned an index according to the order that it is added to its parent. In cases that uniqueness is required, this index will be appended to the name of the box.
- **Dimensions** Simple box dimensions are set through its constructor in micrometers. The resolution is $1\mu m$.
- **Location** The location of a box is defined as the coordinates of its corner with the smallest coordinates on a 2-D cartesian system. When a box is created it is initially located at the origin. It can later be *shifted* to any location on the coordinate system (negative coordinate values are allowed). A box can be shifted multiple times, in which case the effect will be accumulative. In the GUI, boxes are displayed in a upside-down cartesian coordinate system, i.e. *positive-x* direction is east and *positive-y* direction is south. Coordinates are set in micrometers at the resolution of $1\mu m$.
- **Power** The total power spent (static and dynamic) in watts per second.
- **Color** The color attribute is used by the GUI to display a simple box in color.

Above attributes (except for the location and the GUI color as explained before) are valid for a composite box as well. However dimensions and the power are automatically derived from its sub-boxes therefore they cannot be directly set. The dimensions of a composite box are defined as the dimensions of its bounding box, which is the smallest rectangle that covers all the boxes in it. Similarly, power of a composite box is defined as the sum of powers of all the simple box instances that it contains. The only user definable attribute of a composite box is its name which is passed in the constructor override (`super(...)` call).

A floorplan is valid if its bounding box is completely covered by the simple boxes in it and none of the simple boxes overlap. Composite box class API provides two geometrical check methods to ensure validity: `checkArea` reports an error if gaps or overflows in the floorplan exist by comparing the bounding box area of the floorplan with the total area of the simple boxes in the floorplan and `checkIntersections` checks for overlaps between boxes. These two methods form a comprehensive geometric check. However they might require a considerable amount of computation especially for floorplans that consist of many simple boxes. These overheads can be a problem if numerous experiments are to be performed on a floorplan, therefore one might choose to remove the checks after the initial run to optimize performance.

The relevant Java classes for creating floorplans are `SimpleBox`, `CompositeBox` and `Box`. A floorplan in a CompositeBox object can be displayed in a GUI via the `showFloorplan` method of the `FloorplanVisualizationPanel` class (which is equivalent to the -fp option of the hotspotj script). In the GUI, each SimpleBox object of the floorplan will be shown in the color that it is assigned (or gray if no color is assigned). Options for converting the image to grayscale and displaying box names are provided. The floorplan can be exported as an image file (jpeg, gif, etc.) from the GUI.

7.3 Creating/Running Experiments and Displaying Results

The command line of HotSpotJ allows running steady-state and transient experiments on a user provided floorplan without further setup. The input should either be in the form of a compiled Java floorplan class extending CompositeBox or a HotSpot floorplan (FLP) file. HotSpot configuration values and the initial temperatures/power consumption values can be set from text files or can be directly set in the constructor of the Java class. For details see the documentation of -steady and -transient options of the hotspotj script.

A typical experiment is built as follows. First the HotSpot engine and the data structures that will be used as the communication medium between the C and the Java code are initialized. Then the stages of the experiment, which can be any combination and number of steady-state and transient calculations, are executed. Finally the resources used by the HotSpot engine are deallocated and results are displayed and/or saved.

7.4 Limitations

A known limitation of HotSpotJ is, no floorplan layer configuration can be provided with the grid model. Users are limited to the default number of layers provided by HotSpot, which is a base layer (layer 0) and a thermal interface material layer[1]. Consequently, experiments where only the base layer dissipates power are supported. Other limitations of which we are not aware may exist and may be revealed over time. HotFloorplan, which is another tool that comes with HotSpot, is not supported through HotSpotJ.

[1]For the specifications of these layers, see the populate_default_layers function in temperature_grid.c file of HotSpot.

8 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 components through which the instruction and data packets travel. Traces can be limited to specific instructions in the assembly input and/or to specific TCUs.

Floorplan visualization. The FPJ package of the HotSpotJ tool can be used for purposes other than interfacing between XMTSim and HotSpotJ. The amount of simulation output can be overwhelming, especially for a configuration that contains many TCUs. FPJ 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. FPJ is explained in further detail in Appendix 7.

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.

9 Potential Improvements

XMTSim is an experimental tool that is under active development and as such, some features are in its future roadmap.

More accurate DRAM model. As mentioned earlier, an accurate external DRAM model, such as DRAMSim, would improve accuracy of XMTSim [47].

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 [22]. 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 [24], 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 [20]. 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.

10 Related Work

Cycle-accurate architecture simulators are particularly important for evaluating the performance of parallel computers due to the great variation in the systems that are being proposed. Many of the earlier projects simulating multi-core processors extended the popular uniprocessor simulator, SimpleScalar [2]. However, as parallel architectures started deviating from the model of simply duplicating serial cores, other multi/many-core simulators such as ManySim [51], FastSim [12] and TPTS [13] were built. XMTSim differs from these simulators, since it targets shared memory many-cores, a domain that is currently underrepresented. GPU simulators, Barra [43], Ocelot [33] and GPGPUSim [3] are closer to XMTSim in the architectures that they simulate but they are limited by the programming models of these architectures. Also, Barra and Ocelot are functional simulators, i.e., they do not report cycle-accurate measures. Kim, et al extended Ocelot with a power model, however it is not possible to simulate dynamic power and thermal management with this system.

Cycle-accurate architecture simulators can also be built on top of existing simulation frameworks such as SystemC. An example is the simulator presented by Lebreton, et al. [34]. Instead, we chose to build our own infrastructure since XMTSim is intended as a highly configurable simulator that serves multiple research communities. Our infrastructure gives us the flexibility to incorporate second party tools, for example SableCC [21], which is the front end for reading the input files. In this case, SableCC enabled easy addition of new assembly instructions as needed by users of XMTSim.

Simulation speed is an issue, especially in evaluating thermal management. Atienza, et al. [1] presented a hardware/software framework that featured an FPGA for fast simulation of a 4-core system. Nevertheless, it is not feasible to fit a 1024-TCU XMT processor on the current FPGAs.

Appendix A Extended XMTSim Documentation

This appendix contains detailed documentation of XMTSim, including installation instruction, a command line usage manual and a list of configurable parameters for XMT architecture research.

A.1 General Information and Installation

XMTSim is typically used with the XMTC compiler which is a separate download package. It can be used standalone in cases that the user directly writes XMT assembly code.

To use XMTSim, you must:

- Download and install the XMTC compiler (typically).
- Download and install XMT memory tools (optional).
- Build/install XMTSim.

The XMT toolchain can be found at

`http://www.umiacs.umd.edu/users/vishkin/XMT/index.shtml#sw-release`

and also on Sourceforge

`http://sourceforge.net/projects/xmtc/`

A.1.1 Dependencies and install

Pre-compiled binary distribution consists of a Java jar file and a bash script file. Due to platform independent nature of Java, this distribution is platform independent as well. The cygwin/linux dependent bash script is distributed for convenience and is not an absolute requirement. In future distributions, simulator may include platform dependent components.

System requirements:

- You must have Sun Java 6 (JRE - Java Runtime Environment) or higher on your system. Java executable must be on your PATH, i.e. when you type "java -version" on command line you should see the correct version of Java. XMTSim might work with other implementations of Java that are equivalents of the Sun Java 6 (or higher) however it is only tested with the Sun version of Java. Note that the XMTC compiler and memory tools come with their own set up system requirements that are independent of the simulator.
- In order to use the script provided in the package, you must have bash installed on your system. XMTSim can directly be run via the "java -jar" command. Read the xmtsim script if you would like to use the simulator in such a way.

Follow these steps to install XMTSim:

1. Create a new directory of your choice and place the contents of this package in the directory. Example: /xmtsim
2. Make sure java is on the PATH:

   ```
   > java -version
   ```

   The commands should display the correct java version.
3. Add the new directory to the PATH. Example (bash):

   ```
   > export PATH=~/xmtsim:$PATH
   ```
4. Test your installation:

   ```
   > xmtsim -version
   > xmtsim -check
   ```

Type "xmtsim -help" and "xmtsim -info" for information on how to use the simulator. For detailed examples see the XMTC Manual and Tutorial [11].

A.2 XMTSim Manual

This manual lists the usage of all command line controls of XMTSim and also includes a brief manual of the trace tool. The manual is also available on the command line and can be displayed via `xmtsim -info all`.

```
Usage: xmtsim [<input assembly file> | -infile <input assembly file>]
    [-cycle ]
       [-conf <configName>]
       [-confprm <prmName> <value>]
       [-timer <?num | num~>]
       [-interrupt <num>]
       [-starttrace <num>]
       [-savestate <filename>]
       [-checkpoint <num>,<num>...]
       [-stop <num | actsw+num> | <+num>]
       [-activity | -activity=<options>]
       [-actout <filename>]
       [-actsw]
       [-preloadL1 | -preloadL1=<num>]
       [-debug | -debug=<num>]
       [-randomize <num>]
    [-count | -count=<options>]
    [-trace | -trace=<options>]
    [-mem <debug <?val> | simple | paged>]
    [-binload <filename>]
    [-textload <filename>]
    [-loadbase <address>]
    [-bindump <filename>]
    [-textdump <filename>]
    [-hexdump <filename>]
    [-dumprange <startAddr> <endAddr>]
    [-dumpvar <variableName>]
    [-printf <filename>]
    [-out <filename>]
    [-traceout <filename>]
    [-argfile <filename>]

For running a program, choose from the appropriate options
listed in square braces ([...]). A pipe character (|) means one
of the multiple variants should be chosen. Mandatory parameters to
an option are indicated with angle braces (<...>). Optional
parameters are indicated with angle braces with a question mark
(<?...>). Options indented under 'cycle' option can only be used
when 'cycle' is specified.

Below are different forms of the xmtsim command, that show options
that should be used standalone and not with each other or with the
ones above.

  xmtsim -resume <filename>
  xmtsim (-v|-version)
  xmtsim (-h|-help)
  xmtsim -info [<option>]
  xmtsim -check
  xmtsim -diagnose <?number> [-conf <configName>]
  xmtsim -diagnoseasm <?number>
  xmtsim -conftemplate <configName>
  xmtsim -checkconf <filename>
  xmtsim -listconf <filename>
```

```
OPTIONS

  -h -help
    Display short help message.
  -info
    Display this info message.
  -v -version
    Display the version number.
  -check
    Runs a simple self test.
  -diagnose <?number> [-conf <configName>]
    Runs a simple cycle-accurate diagnostics program to show how
    fast the user system is under full load (all TCUs working).
    Depending on the number (0 - default, 1, 2, etc.) a different
    test will be run. -conf can optionally be used to change the
    default configuration.
  -diagnoseasm <?number>
    Runs a simple diagnostics program in assembly simulation mode to
    show how fast the user system is under full load.
    Depending on the number (0 - default, 1, 2, etc.) a different
    test will be run.
  -conftemplate <configName>
    Creates a new template file that can be modified by the user
    and passed to the 'conf' option. The file created will have
    the name 'configName.xmtconf'.
  -checkconf <filename>
    As input, takes a file that is typically passed to the
    'conf' option. Checks if the field types and values are all
    correctly defined, if all fields exist and if all the
    configuration parameters are set in the input file.
  -listconf <filename>
    As input, takes a file that is typically passed to the
    'conf' option. Lists all the field names and their values sorted
    according to names. Can be used to compare two conf files.
    For this option to return without an error the conf file should
    pass checkconf with no errors.
  -argfile <filename>
    Reads arguments from a text file and inserts them on the command
    line at the location that this argument is defined. Lines
    starting with the '#' character are ignored. Multiple argument
    files can be defined using multiple occurrences of this
    parameter, ex: -argfile file1.prm -argfile file2.prm. The
    parameters from these file will be inserted in the order that
    they appear on command line.
  -cycle
    Runs the cycle accurate simulation instead of the assembly
    simulation. For obtaining timing results, this option should
    be used. It comes with a list of sub-options.
  -timer <?num | num~>
    Provides an updated cycle count every 5 seconds. The default
    value of 5 can be overridden by passing an integer number
    after the timer option.
    Timer option can be used in tandem with the count (or
    detailed count) option to display the detailed instruction
    counts as well as the cycle count.
    If passed integer is followed with a '~' character
    (ex. -timer 2500~), the information will be printed periodically
    in simulation clock cycles rather than real time.
  -interrupt <num>
    If this option is used, the cycle accurate simulation will be
    interrupted before completion and the simulator will exit with
    an error value. Interrupt will happen after N minutes after the
    simulation starts where N is the value of this parameter.
```

```
  If the simulation is completed before the set value, it will
  exit normally.
-conf <configName>
  Reads the simulator cycle accurate hardware configuration.
  This might be a built-in configuration or an externally
  provided configuration file. The search order is as follows:
    1. Search <configname> among the built-in configurations.
    2. Search for the file <configname>.xmtconf
    3. Search for the file <configname>
  The built-in configurations are '1024', '512', '256'
  (names signify the  tcu counts in the configuration) and 'fpga'
  (64 tcu configuration, similar to the Paraleap FPGA computer).
-confprm <prmName> <value>
  Sets the value of a configuration parameter on command line.
  It will overwrite the values set by the -conf option.
-starttrace <num>
  Starts the tracing (as specified by the -trace option)
  only after <num> cycles instead of from the beginning of the
  simulation.
-stop <num | actsw+num | +num>
  Schedules the simulation to stop at a used defined time. If
  actsw+num is passed the stop time will be relative to the
  actsw instruction. The latter requires actgate option. If +num
  is used in conjunction with the resume option, simulation will
  stop at current time plus num.
-activity
-activity=<options>
  This is an experimental option that logs activity of actors
  that implement ActivityCollectorInterface. For a list of
  available options, try -activity=help. Requires -cycle option.
-actout <filename>
  Redirects the output of the activity option to a file. If no
  activity option is defined, this option will not have an effect
  except for creating an empty file.
-actgate
  Gates the output of the activity collector unless its state is
  on. The state can be switched on and off via the actsw
  instruction in assembly. Initial state is off. The activity
  collection mechanism still works in the background but it
  doesn't print its output.
-savestate <filename>
  Dumps the state of the simulator in a file. This option is used
  in order to pause simulation and restart it at a later time.
  The paused simulation is resumed via the resume option.
  Simulation can be stopped and state can be saved in three
  different ways: simulation can terminate naturally at the
  end of the execution of the input (meaning a halt, hex/textdump,
  bindump, ... instruction is encountered), at a user defined
  time via the 'stop' argument or a via a SIGINT (CTRL-C)
  interrupt. If the simulation ends naturally the state dump
  is only useful for inspection with a debugging tool since
  there is nothing to resume.
  The output file should not already exist or the command will
  fail with an error.
  All command line arguments that are used during this
  call will be lost during the save operation. Exceptions are
  the arguments that directly affect the state of the simulation
  such as the input file, 'binload', 'conf', etc. Other
  exceptions are the 'count' and 'activity' arguments. If the
  activity option is using a user provided plug-in that cannot
  be saved (not implementing the serializable interface), it will
  be lost as well. Arguments that have been lost can be redefined
  while the simulation is resumed (see the 'resume' argument).
Also see: checkpoint
```

```
-checkpoint <num>,<num>...
  Used to dump states during execution without quitting the
  simulation. Should be used with the savestate option.
  The names of the state files are based on the filename passed to
  savestate. They will be appended with .[time] suffix. Checkpoint
  option takes a mandatory comma separated list of numbers which
  represent the clock cycles that the states will be saved.
  The state dump events have low priority, meaning the state will
  be saved after all events of a clock cycle are processed.
  The state will still be saved at the end as if savestate option
  was used stand-alone.
Also see: savestate
-resume <filename>
  Resumes a simulation that has been paused by the 'savestate'
  option. Implies the 'cycle' flag. This argument can be used
  with other command line arguments such as trace, count,
  bindump, dumpvar, out, printf. This is how a user can redefine
  the options that were lost during save state (see savestate
  argument). However command line
  arguments that directly affect the state of the simulation
  should not be used; the results are undefined. Such arguments
  are redefinition of input file, infile, conf, preloadL1,
  check/warnmemreads, bin/textload and loadbase. If count and
  activity arguments were defined in the original run,
  redefining them during resume will not have an effect.
  "-activity=disable" option can be used to remove an activity
  collection plug-in that was saved from the original run.
-trace
-trace=<options>
  By default dumps out the instruction results filtering out
  all instructions marked as skipped.
  When used with additional options -trace is a powerful tool
  to monitor the system for tracing instructions through the
  hardware and reading results of instructions.
  For more information see the "Trace Manual" section below.
-count
-count=<options>
  Displays the number of instructions executed. In case of
  parallel programs, this will be the total number of
  instructions executed by all TCUs. The instructions that are
  marked by the @skip directive are not counted.
  The simulator can have only one counter. To change the default
  counter specify the plugin:[class path]. The full path for the
  built-in counters (ones in the utility package of the simulator)
  is not required, only the class name is sufficient. For a list
  of available options, try -count=help.
-binload <binary memory file>
  Load the data memory image from a binary file. This option is
  compatible with the XMT Memory Map Creator tool.
-textload <text memory file>
  Load the data memory image from a text file. The format of the
  text file is, numerical values of consecutive words
  separated by white spaces. Each word is a 64-bit signed
  integer.This option is compatible with theXMT Memory Map
  Creator tool.
-loadbase <address>
  Loads the data file specified by a -binload or -textload
  option at the address <address>. Default address is 0.
-bindump <filename>
  Dump the contents of the data memory to the given file in
  little-endian binary format. This option is compatible with
  the XMT Memory Map Reader tool.
  Either a range of addresses using -dumprange, or a global
  variable using -dumpvar needs to be specified.
```

```
-textdump <filename>
  Dump the contents of the data memory to the given file in
  text format. The output file is in the same format described
  for 'textload' option.
  Either a range of addresses using -dumprange, or a global
  variable using -dumpvar needs to be specified.
-hexdump <filename>
  Dump the contents of the data memory to the given file in
  hex text format.
  Either a range of addresses using -dumprange, or a global
  variable using -dumpvar needs to be specified.
-dumprange <startAddr> <endAddr>
  Defines the start and end addresses of the memory section that
  will be dumped via 'hex/textdump' or 'bindump' options.
  Without the 'hex/textdump' or 'bindump' options, this parameter
  has no effect.
-dumpvar <variableName>
  Marks a global variable to be dumped after the execution via
  'hex/textdump' or 'bindump' options. This option can be repeated
  for all the variables that need to be dumped.
  Without the 'hex/textdump' or 'bindump' options, this parameter
  has no effect.
-out <filename>
  Write stderr and stdout to an output file. The display order of
  stderr and stdout will be preserved.
  If the printf option is defined, output of printf instructions
  will not be included.
  If the traceout option is defined, output of traces will not be
  included.
-printf <filename>
  Write the output of printf instructions to an output file.
-traceout <filename>
  Write the output of traces to an output file.
-mem <paged | debug <?val> | simple>
  Sets the memory implementation used internally. Default is
  paged.
  Paged memory allocates memory locations in pages as they are
  needed. This allows non-contiguous accesses over a wide range
  (i.e. up to 4GB) without having to allocate the whole memory.
  For example, if the top of stack (tos) is set to 4GB-1,
  simulator will allocate one page that contains the tos and one
  page that contains address 0 at the beginning instead of
  allocating 4GB of memory. In this memory implementation
  all addresses are automatically initialized to 0, however it
  is considered bad coding style to rely on this fact. This is
  the default memory type.
  Debug parameter is used to keep track of initialized
  memory addresses for code debugging purposes. Paged and simple
  memory implementations do not report if an uninitialized memory
  location is being read (remember that they automatically
  initialize all addresses to 0), whereas the debug
  implementation reports a warning or an error. If no additional
  parameter is passed to debug, simulator will print out a
  warning whenever an uninitialized address is read.
  If 'err' is passed as a parameter (-mem debug err), simulation
  will quit with an error for such accesses. If a decimal integer
  address is passed as a parameter a warning will be displayed
  every time this address is accessed (read or write) regardless
  of its initialization status. Users should be aware that
  underlying memory implementation for debug is a hashtable which
  is quite inefficient in terms of storage/speed, therefore it
  should not be used for programs with large data sets.
  Simple memory allocates a one dimensional memory with no
  paging. It remains as an option for internal development and
```

```
    should not be chosen by regular users.
  -infile <filename>
    If the name of the input file starts with a '-' character it
    can be passed through this argument. Otherwise this argument is
    not required.
  -preloadL1
  -preloadL1=<num>
    Preloads the L1 cache with data in order to start the cache
    warm. If used with no number it should be used with -textload
    or -binload, in which case the passed binary data will be
    preloaded into L1 caches. If a number is provided and no
    -textload or -binload is passed, given number of words will
    be assumed preloaded with valid garbage (!) startng from the
    data memory start address. Latter case is intended for
    debugging assembly etc.
    If the data to be preloaded is larger than the total L1 size,
    smaller addresses will be overwritten.
  -debug
  -debug=<num>
    Used to interrupt the execution of a simulation with the
    debug mode. If a cycle time is specified, the debug mode will be
    started at the given time. If not, the user can interrupt
    execution to start the debug mode by typing 'stop' or just
    simply 's' and pressing enter. In the debug mode, a prompt will
    be displayed, in which debugging commands can be entered. Debug
    mode allows stepping through simulation and printing the states
    of objects in the simulation. For a list of commands, type
    'help'. Commands in debugging mode can be abbreviated.
  -randomize <num>
    Used with cycle-accurate simulation to introduce some
    deterministic variations. The flag expects one argument
    that will be used as the seed for the pseudo-random
    generators.

TRACE TOOL MANUAL

Trace option can take additional options in the form below

     -trace=<option 1>,<option 2>,<option 3>,...,<option n>

Each option is separated with commas and no white spaces exist.
Following are the list of trace options:

  track
    Displays instruction package paths through all the hardware
    actors. This option cannot be used unless the -cycle option
    is specified for the simulator.
  result
    Displays the dynamic instruction traces.
  tcu=<num>
    Limits the instructions traced to the ones that are generated
    from the TCU with the given hardcoded ID.
  directives
    Displays only the instructions that are marked in the
    assembly source (see below for a list of directives).
    This option can only be used with tcu=<num>.
    It will not display an instructions if it is marked as 'skip'.

'-trace' with no additional options is equivalent to '-trace=result'.

What are the assembly trace directives?
    Assembly programmers can manually add prefixes to lines of
    assembly to track specific instructions. This feature is
    activated from command line via the -trace=directives option.
```

```
Otherwise all such directives are ignored.

A trace directive should be prepended to an assembly line.
Example:

@track addi $1, $0, 0

Following is the list of directives:

@skip: Excludes the instruction from execution and job traces.
Check for the specific trace command line option you are using
for exceptions.

@track <?num>: Turns on the track suboption just for this
instruction. If a number is specified, only the instructions
that are generated from the TCU with the given hardcoded ID
will be tracked.

@track <?num>: Turns on the track suboption just for this
instruction. If a number is specified, only the instructions
that are generated from the TCU with the hardcoded ID that is
given with this directive and/or on command line via tcu=<num>
will be tracked.

@result <?num>: Turns on the result suboption just for this
instruction. If a number is specified, only the instructions
that are generated from the TCU with the hardcoded ID that is
given with this directive and/or on command line via tcu=<num>
will be tracked.
```

A.3 XMTSim Configuration Options

The configuration options of XMTSim are passed in a text file via the `conf` command line option or one by one via the `confprm` option. The default configuration file, which is listed in this section, can also be generated using the `conftemplate` option of XMTSim.

The initial configuration given here models a 1024 core XMT in 64 clusters. DRAM clock frequency is 1/4 of the to emulate a system with 800MHz core clock frequency and 200MHz DRAM controller. If features 8 DRAM ports with 20 DRAM clock cycle latency.

```
# Number of clusters in the XMT chip excluding master TCU cluster.
int NUM_OF_CLUSTERS 64

# Number of TCUs per cluster.
int NUM_TCUS_IN_CLUSTER 16

# Number of pipeline stages in a tcu before the execute stage.
# (Initially the stages are IF/ID and ID/EX).
int NUM_TCU_PIPELINE_STAGES 3


#*****************************
# CLOCK RATE PARAMETERS
#*****************************

# Period of the Cluster clock. This number is an integer that
# is relative to the other constants ending with _T.
int CLUSTER_CLOCK_T 1

# Period of the interconnection network clock. This number is an
# integer that is relative to the other constants ending with _T.
int ICN_CLOCK_T 1
```

```
# Period of the L1 cache clock. This number is an integer that
# is relative to the other constants ending with _T.
int SC_CLOCK_T 1

# Period of the DRAM clock for the simplified DRAM model. This
# number is an integer that is relative to the other constants
# ending with _T.
int DRAM_CLOCK_T 4

#*****************************
# FU PARAMETERS
#*****************************

# Number of ALUs per cluster. Has to be in the interval
# (0, NUM_TCUS_IN_CLUSTER].
int NUM_OF_ALU 16

# Number of shift units per cluster. Has to be in the interval
# (0, NUM_TCUS_IN_CLUSTER].
int NUM_OF_SFT 16

# Number of branch units per cluster. Has to be in the interval
# (0, NUM_TCUS_IN_CLUSTER].
int NUM_OF_BR 16

# Number of multiply/divide units per cluster. Has to be in the
# interval (0, NUM_TCUS_IN_CLUSTER].
int NUM_OF_MD 1

# Latency in terms of Cluster clock cycles.
int DECODE_LATENCY 1

# Latency in terms of Cluster clock cycles. Does not
# include the arbitration latency.
int ALU_LATENCY 1

# Latency in terms of Cluster clock cycles. Does not
# include the arbitration latency.
int SFT_LATENCY 1

# Latency in terms of Cluster clock cycles. Does not
# include the arbitration latency.
int BR_LATENCY 1

# If true, branch prediction in TCUs will be turned on.
boolean BR_PREDICTION true

# The size of the branch prediction buffer (i.e. maximum number
# of branch PCs for which prediction can be made).
int BRANCH_PREDICTOR_SIZE 4

# The number of bits for the branch predictor counter.
int BRANCH_PREDICTOR_BITS 2

# Latency in terms of Cluster clock cycles. Does not
# include the arbitration latency. Does not include
# the MD register file latency.
int MUL_LATENCY 6

# Latency in terms of Cluster clock cycles. Does not
# include the arbitration latency. Does not include
# the MD register file latency.
int DIV_LATENCY 36
```

```
# Latency of mflo/mtlo/mfhi/mflo operations in terms of
# Cluster clock cycles. Does not include the arbitration
# latency. Does not include the MD register file latency.
int MDMOVE_LATENCY 1

# Latency of the MD unit internal register file in terms
# of Cluster clock cycles.
int MDREG_LATENCY 1

# Latency in terms of base cluster clock cycles.
int PS_LATENCY 12

# This is the half-penalty for a TCU that is requesting
# a PS to a global register that is not the one that is
# currently being handled. See the simulator technical
# report for details. This is in terms of base cluster
# clock cycles.
int PS_REG_MATCH_PENALTY 4

# Latency at the cluster input in terms of interconnection
# network clock cycles.
int LS_RETURN_LATENCY 1

# Latency in terms of base cluster clock cycles.
# Found empirically from the FPGA via microbenchmarks.
# This is an average but might not exactly match all cases due
# to mechanism differences between the simulator and FPGA.
int SPAWN_START_LATENCY 23

# Latency in terms of Cluster clock cycles. This is the latency
#  of the SJ unit to return to serial mode after all TCUs go idle.
#  Cannot be 0.
int SPAWN_END_LATENCY 1

#*****************************************
# FLOATING POINT FUNCTIONAL UNITS PARAMETERS
#*****************************************

# Number of Floating Point ALUs per cluster.
int NUM_OF_FPU_F 1

# All folowing latencies are in terms of Cluster clock cycles.
# Do not include the arbitration latency.
int MOV_F_LATENCY 1

int ADD_SUB_F_LATENCY 11

int MUL_F_LATENCY 6

int DIV_F_LATENCY 28

int CMP_F_LATENCY 2

int CVT_F_LATENCY 6

int ABS_NEG_F_LATENCY 1

#****************************
# MEMORY PARAMETERS
#****************************

# Total number of memory ports. Has to be a 2's power multiple of
# NUM_OF_CLUSTERS.
int NUM_CACHE_MODULES 128
```

```
# Total number of DRAM ports. Has to be of the form
#NUM_CACHE_MODULES / 2^k
# with k >= 0. Default is is one DRAM port per CACHE_MODULE
# (no contention). This parameter is ignored unless the memory
# model does not imply that DRAM is simulated.
# NUM_CACHE_MODULES should be a multiple of this parameter. If
# they are not equal the type of the DRAM port that will be
# instantiated is SharedSimpleDRAMActor. If not the type is
# SimpleDRAMActor.
int NUM_DRAM_PORTS 8

# If the ICN_MODEL parameter is set to "const", this value
# will be used as the constant ICN latency. The delay time
# will be equal to ICN_CLOCK_T x CONST_SC_LATENCY.
int CONST_ICN_LATENCY 50

#****************************************
# ADDRESS AND CACHE PARAMETERS
#****************************************

# The number of words (not bytes) that a cache line contains.
# The number of bytes in a word is defined by MEM_BYTE_WIDTH in
# xmtsim.core.Constants.
int CACHE_LINE_WIDTH 8

# Master Cache parameters

# Size of the MasterCache in bytes.
int MCACHE_SIZE 16384

# Master cache can serve only this many different cache line misses. For
# example, if master cache receives 5 store word instructions all of
# which are misses to different cache lines, 5th instruction will stall.
int MCACHE_NUM_PENDING_CACHE_LINES 4

# Master cache can serve only this many different misses for a
# given cache line. For example, if master cache receives 9 store
# word instructions all of which are misses to the same cache line,
# 9th instruction will stall.
int MCACHE_NUM_PENDING_REQ_FOR_CACHE_LINE 8

# L1 parameters

# Size of the L1 Cache in bytes (per module).
int L1_SIZE 32768

# Associativity of the L1 cache. 1 for direct mapped and
# Integer.MAX_VALUE for fully associative.
# Note that setting this value to Integer.MAX_VALUE has the same
# effect as MCACHE_SIZE / (MCACHE_LINE_WIDTH * MEM_BYTE_WIDTH)
int L1_ASSOCIATIVITY 2

# L1 cache can serve at most this many different pending cache line
# misses. For example, if master cache sends 9 store word instructions
# all of which are misses to different cache lines, 9th instruction will
# stall.
int L1_NUM_PENDING_CACHE_LINES 8

# L1 cache can serve at most this many different pending misses for a
# given cache line. For example, if L1 cache receives 9 store word
# instructions all of which are misses to the same cache line,
# 9th instruction will stall.
int L1_NUM_PENDING_REQ_FOR_CACHE_LINE 8
```

```
# The size of the DRAM request buffer, which is the module that the
# requests from L1 to DRAM wait until they are picked up by DRAM.
# NOTE: Setting this to a size that is too small (8) causes deadlocks.
# Deadlocks can be encountered even with bigger sizes if the
# DRAM_LATENCY is not large enough.
# NOTE2: In Xingzhi's thesis, this buffer is called L2_REQ_BUFFER for
# historical reasons.
int DRAM_REQ_BUFFER_SIZE 16

# The size of the DRAM response buffer, which is the module that the
# responses from DRAM to L1 wait until they are picked up by L1.
# NOTE: In Xingzhi's thesis, this buffer is called L2_RSPS_BUFFER for
# historical reasons.
int DRAM_RSPS_BUFFER_SIZE 2

#****************************************
# ADDRESS HASHING PARAMETERS
#****************************************

# If this is set to true, hashing will be applied on physical memory
# addresses before they get sent over the ICN. This variable does not
# change the cycle-accurate delay that is incurred by the hashing unit
# but it turns on/off the actual hashing of the address.
boolean MEMORY_HASHING true

# Constant set by the operating system (?) for hashing
# See ASIC document for algorithm.
int HASHING_S_CONSTANT 63

#****************************************
# PREFETCHING PARAMETERS
#****************************************

# The number of words that fit in the TCU Prefetch buffer.
int PREFETCH_BUFFER_SIZE 16

# The replacement policy for the prefetch buffer unit:
#  RR       - RoundRobin
#  LRU      - Least Recently Used
#  MRU      - Most Recently Used
String PREFETCH_BUFFER_REPLACEMENT_POLICY RR

# The number of words that fit in the read only buffer.
int ROB_SIZE 2048

# The maximum number of pending requests to ROB per TCU.
int ROB_MAX_REQ_PER_TCU 16

#****************************************
# MISC PARAMETERS
#****************************************

# The interconnection network model used in the simulation.
# const   - Constant delay model. If this model is chosen, the cache and
#           DRAM models will be ignored. The amount of delay is taken
#           from CONST_ICN_LATENCY.
# mot     - Separate send and receive Mesh-of-Trees networks between
#           clusters and caches.
String ICN_MODEL mot

# The shared cache model used in the simulation. This has no effect if
# const model is chosen for the ICN model.
# const   - Constant delay model. If this model is chosen, the DRAM
```

```
#           model will be ignored. The amount of delay is taken from
#           CONST_SC_LATENCY.
# L1      - One layer shared cache as it is implemented in the Paraleap
#           FPGA computer.
# L1_old  - One layer shared cache as in L1 model. This is an old
#           implementation of the L1 cache and should not be used by the
#           typical user due to possible bugs.
String SHARED_CACHE_MODEL L1

# The DRAM model used in the simulation.  This has no effect if either
# const model is chosen for the ICN model or const model is chosen for
# the shared cache model.
# const  - Constant delay model. The amount of delay is taken from
#          CONST_DRAM_LATENCY.
String DRAM_MODEL const

# The memory model for the master tcu:
# hit    - All memory requests will be hits in the cache. The amount of
#          the delay can be set through the MCACHE_HIT_LATENCY.
# miss   - All memory requests will go through the ICN model defined in
#          MEMORY_MODEL. The master cache will add two clock
#          cycles to the ICN latency (one on the way out one on the way
#          back).
# full   - full MCACHE implementation.
String MCLUSTER_MEMORY_MODEL miss

# The number of CLUSTER clock cycles that Master cache serves a cache
# hit in case of the 'hit' value of MCLUSTER_MEMORY_MODEL.
int MCACHE_HIT_LATENCY 1

# The number of clock cycles that a shared cache module serves a request
# for the constant delay implementation of shared cache. The delay time
# will be equal to SC_CLOCK_T x CONST_SC_LATENCY.
# See SHARED_CACHE_MODEL.
int CONST_SC_LATENCY 1

# The number of DRAM cycles that DRAM serves a memory request in case
# of the simplified implementation of DRAM. See DRAM_MODEL parameter.
# Also see the note at DRAM_RSPS_BUFFER_SIZE.
# The typical latency of DRAM for DDR2 at 200MHz is about 20 cycles. If
# scaling this number for a higher clock frequency the latency should
# also be increased proportionally to give the same delay in absolute
# time.
int CONST_DRAM_LATENCY 20

# If 'grouped', cluster 0 will get TCUs 0, 1, 2, ..., (T-1) and
# cluster 1 will get T, T+1, T+2, T+3, etc. T is the number of TCUs in
# a cluster.
# If 'distributed' cluster 0 will get TCUs 0, N, 2N and cluster 1 will
# get TCUs 1, N+1, 2N+1, etc. N is the number of clusters.
# This parameter makes a difference in programs with low parallelism.
# 'Grouped' option might be used if power is a concern, otherwise
# 'distributed' option should result in better performance.
String TCU_ID_ASSIGNMENT distributed
```

Appendix B HotSpotJ Extended Documentation

This appendix contains detailed documentation of HotSpotJ, a command line including installation instructions, a command line usage manual and a tutorial for constructing a new floorplan.

B.1 Installation

B.1.1 Software Dependencies

HotSpotJ relies on Java Native Interface (JNI) [36] for interfacing with C-language which is platform dependent unlike pure Java code. Development and testing is done under Linux OS. In earlier development stages, it has been tested on Windows OS/Cygwin and the build system still supports compilation under Cygwin. It is very probable that the Cygwin build still works without problems however, we are not actively supporting it. Table 5 lists the specifications of the system under which HotSpotJ is tested.

Linux OS	kernel: 2.6.27-13 distribution: ubuntu 8.10
Bash	3.2.48
GNU Make	3.81
Sun Java Development Kit (JDK)	1.6.0
GNU C Compiler (GCC)	4.3.3

Table 5: Specifications of the HotSpotJ test system.

HotSpotJ package contains a copy of the HotSpot source code therefore a separate HotSpot installation is not required.

B.1.2 Building the Binaries

HotSpotJ installation is built from source code. Prior to running the build script, you should make sure that the required tools are installed and their binaries are on the PATH environment variable (see Table 5 for the list).

Following are the steps to build HotSpotJ. Each step includes example commands for the `bash` shell.

- Download the source package at `http://www.ece.umd.edu/~keceli/web/software/HotSpotJ/`.
- Uncompress the package in a directory of your choice (/opt in our example, `xxx` is the version). A hotspotj directory will be created.

```
> tar xzvf hotspotj_xxx.tgz /opt/
```

- Make sure that the `javac`, `java` and `javah` executables are on the path (you can check this using the linux `which` command). If not, set the PATH environment variable as in the example below.

```
> export PATH=$PATH:/usr/lib/jvm/java-6-sun/bin
```

- Include the bin directory under the HotSpotJ installation in the PATH environment variable.

```
> export PATH=$PATH:/opt/hotspotj/bin
```

- Run the `make` command in the installation directory.

```
> cd /opt/hotspotj
> make
```

For proper operation, the PATH variable should be set everytime before the tool is used (which can be done in the `.bashrc` file for the `bash` shell). You can turn on a math acceleration engine for HotSpot by editing the `hotspotj/hotspotcsrc/Makefile` file. For more information on the math acceleration engines that can be used with HotSpot, see the HotSpot documentation.

Compiling the floorplans written in Java requires the CLASSPATH environment variable to include the HotSpotJ java package. For example,

```
> export CLASSPATH=$CLASSPATH:/opt/hotspotj
```

If you are using Sun Java for MS Windows under Cygwin, the colon character should be exchanged with backslash and semi-colon characters (\;).

In order to use HotSpotJ as an API in another Java based software (e.g. a cycle-accurate simulator or a custom experiment), you should set the LD_LIBRARY_PATH environment variable to include the HotSpotJ java package. For example,

```
> export LD_LIBRARY_PATH=$LD_LIBRARY_PATH:/opt/hotspotj/bin
```

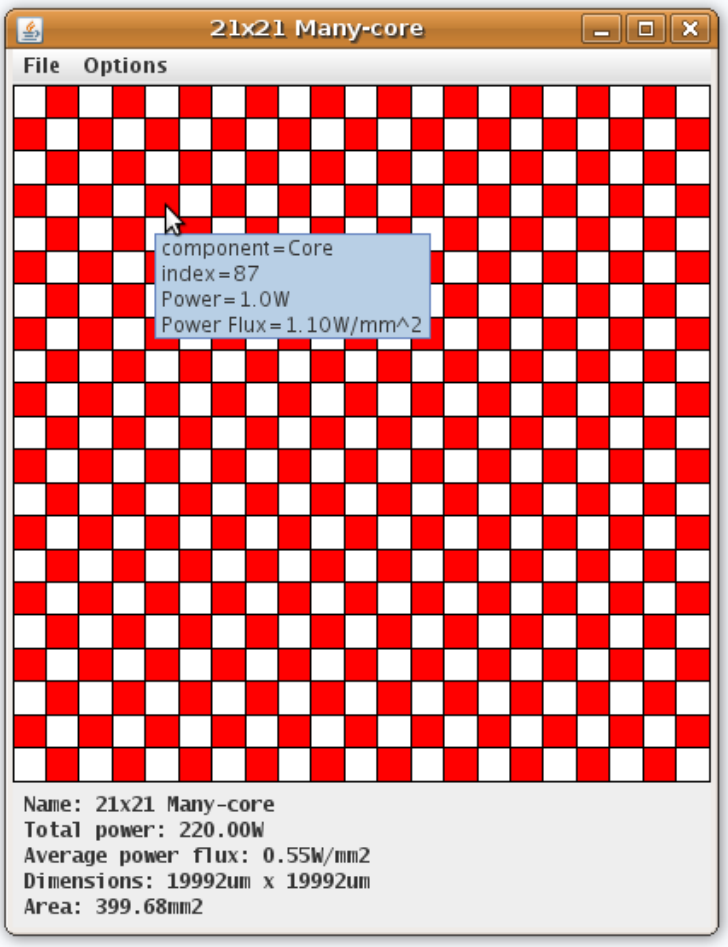

Figure 12: 21x21 many-core floorplan viewed in the floorplan viewer of HotSpotJ. Red boxes denote the cores and white boxes are the cache modules. The GUI displays information about a box as a tooltip when the mouse pointer is held steady over it.

B.2 Tutorial – Floorplan of a 21x21 many-core processor

In this tutorial, we will show how to construct a floorplan using the HotSpotJ Java API, compile it, check it for geometric errors and view it using the GUI.

The examples that we will demonstrate are taken from a paper by Huang et al. [26], in which they investigate the thermal efficiency of a processor with 220 simple cores and 221 cache modules. Cores and caches are assumed to be shaped as squares and they are placed in a 20mm by 20mm die using a checkerboard layout. 1W power is applied to each core and caches do not dissipate any power. Figure 12 shows the floorplan.

Below is the self documented Java code for this floorplan[2]. It should be noted that the code contains less than 20 statements if the inline comments are ignored.

In the code, first, one non-hierarchical box (also called *simple box*) per cache module and core is created and its attributes are set. Each simple box is then added to its parent level hierarchical box (or *composite box*). As the last step of the code, the floorplan is checked for geometric errors.

B.2.1 The Java code for the 21x21 Floorplan

```
package tutorial;

import java.awt.Color;

import hotspotjsrc.CompositeBox;
import hotspotjsrc.SimpleBox;
```

[2]The associated Java file can be found at `tutorial/ManyCore21x21.java`.

```
public class ManyCore21x21 extends CompositeBox {

  private static final long serialVersionUID = 1L;

  public ManyCore21x21() {
    // Pass the name of the floorplan to the constructor of
    // CompositeBox.
    super("21x21 Many-core");

    // A building block is a cache or a core.
    // This is the dimension of the rectangular unit in um.
    int BOX_DIM = (int)((20.0/21.0)*1000);

    // Total power of one core in watts.
    double CORE_POWER = 1;

    // This is the two level loop where the floorplan is created.
    for(int x = 0; x < 21; x++) {
      for(int y = 0; y < 21; y++) {
        SimpleBox bb;
        if((y+x*21)%2 == 1) {
          // The odd numbered elements are the cores.
          // Build a non-hierarchical square box for a core.
          bb = new SimpleBox("Core", BOX_DIM, BOX_DIM);
          // Cores dissipate power.
          bb.setPower(CORE_POWER);
          // Set the color of the cores to red in the GUI.
          bb.setFloorplanColor(Color.red);
        } else {
          // The even numbered elements are the caches.
          // Build a non-hierarchical square box for a cache module.
          bb = new SimpleBox("Cache", BOX_DIM, BOX_DIM);
          // The even numbered elements are the caches and they
          // do not dissipate power.
          bb.setPower(0.0);
          // Set the color of the caches to white in the GUI.
          bb.setFloorplanColor(Color.white);
        }
        // Shift the box to the correct location in the checkerbox
        // grid.
        bb.shift(x * BOX_DIM, y * BOX_DIM);
        // Add the new simple box to the ManyCore21x21 object.
        addBox(bb);
      }
    }

    // Check the floorplan for geometric errors. These checks
    // might take a long time for a large floorplan.
    checkArea();
    checkIntersections();
  }
}
```

B.3 HotSpotJ Command Line Options

```
-fp <class path>
  Loads a CompositeBox class to view its floorplan on the
  HotSpotJ GUI. Class path should be expressed in Java
  notation, the associated class file should have been
  previously compiled and the CLASSPATH environment variable
  should be set appropriately in order to load the class. See
  the HotSpotJ tutorials for detailed examples.
```

```
-steady <class path>
  Loads a CompositeBox class to run a steady state
  experiment on it. It is assumed that the power values are
  already set in the floorplan. Class path should be expressed
  in Java notation, the associated class file should have been
  previously compiled and the CLASSPATH environment variable
  should be set appropriately in order to load the class. See
  the HotSpotJ tutorials for detailed examples.

  This option internally uses the steadySolve method of
  CompositeBox class.

-transient <class path> <num>
  Loads a CompositeBox class to run a transient experiment
  with constant power values on it. It is assumed that the
  power values are already set in the floorplan. The number of
  iterations is set by the num parameter. The iteration period
  is controlled by the const_sampling_intvl field in the
  HotSpotConfiguration class, which can be set in the
  floorplan file in compilation time.

  Class path should be expressed in Java notation, the
  associated class file should have been previously compiled
  and the CLASSPATH environment variable should be set
  appropriately in order to load the class. See the HotSpotJ
  tutorials for detailed examples.

  This option internally uses the transientSolve method of
  CompositeBox class.

-loadhjd <?filename>
  Loads a previously saved data file. If no data file is
  specified, a GUI window will be brought up to select a file
  from the file system.

-experiment <classpath> [-save [-base <name>]] [-showinfo] <run_options>
  This option is used to call the run method of a class that
  extends Experiment. The classpath argument should point to
  the full java name of the Experiment class (for example
  tutorial.TutorialExperiment), which should be on the
  CLASSPATH. If save option is defined, returned panels will
  be saved in HJD files. File names will be derived from the
  name of the class. If the base option is defined the file
  names will use it as the base. If no save option is defined
  panels will be shown in the GUI. The showinfo option prints
  the info text (see hjd2info option) for each panel to
  standard out.

  See HotSpotJ documentation for more information on setting
  up experiments.

-showflp <FLP file>
  Reads a HotSpot floorplan (FLP) file and loads it into the
  floorplan viewer GUI.

-fp2flp <class path>
  Converts a CompositeBox class to a HotSpot floorplan (FLP)
  file representation and prints it on standard output. This
  option can be used to write a complex floorplan in HotSpotJ
  and then work on it in HotSpot.

-hjd2image [-savedir <dirname>]
           [-mintemp <value>] [-maxtemp <value>]
```

```
          [-minpower <value>] [-maxpower <value>] <hjd filenames..>
  This is a batch processing option that saves the power and
  temperature map images in the HJD files to image files.
  Multiple image types are supported (platform dependent) and
  a list of supported types can be obtained via the "hotspotj
  -hjd2image -type list" command. The image type is set with
  the "-type" option. If a type is not provided, default is
  jpeg.

  By an output file is written to the same directory that
  the associated input file is read from. This can be changed
  via the savedir option. The relative paths of the input
  files will be conserved in the save directory.

  Arguments that are not options are considered as input
  files.

  A purpose of batch converting HJD files to images is to
  assemble movies that show the change in power and
  temperature maps over the course of an experiment. For this,
  the experiment should periodically save the data. The the
  user can use the hjd2image option to convert the data to
  image files and these files can be converted to an mpeg
  movie (a script that does this conversion is provided in the
  HotSpotJ package).

  Note that, the color scheme in a temperature or power map
  is derived relative to the minimum and maximum values in the
  map (i.e. minimum and maximum will be at the opposite ends
  of the color spectrum). However in order to make a
  meaningful movie, the color schemes should be consistent
  between all maps. Therefore below options are provided for
  the user to set minimum and maximum values for the
  temperature and power map color schemes:

  [-mintemp <value>]  [-maxtemp <value>]
  [-minpower <value>] [-maxpower <value>]

  Values are in Kelvins. If a value is not provided, it will
  be set from the minimum/maximum found in the associated map.
  If a value is outside the user provided range, it will be
  truncated to the minimum/maximum.

-hjd2info [-intbase <float>] <filenames...>
  This is a batch processing command that works in the same
  way as the hjd2image option but instead of saving map images
  it prints the information of each input hjd file on standard
  output. The information is the text displayed in the power
  and temperature tabs below the maps.

  If instbase is specified, the base value for the
  temperature integral will be set. See HotSpotJ documentation
  for information on temperature integral.
```

References

[1] D. Atienza, P. G. D. Valle, G. Paci *et al.*, "HW-SW Emulation Framework for Temperature-Aware Design in MPSoCs," in *Proceedings of the Design Automation Conference*, 2006.

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

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

[4] 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.

[5] D. Brooks, V. Tiwari, and M. Martonosi, "Wattch: a framework for architectural-level power analysis and optimizations," in *Proceedings of the International Symposium on Computer Architecture*, 2000.

[6] G. Caragea and U. Vishkin, "Better speedups for parallel max-flow." in *Proceedings of the ACM Symposium on Parallelism in Algorithms and Architectures*, 2011.

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

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

[9] G. C. Caragea, F. Keceli, and A. Tzannes, "Software release of the XMT programming environment," www.umiacs.umd.edu/users/vishkin/XMT/sw-release.html, 2008 – 2010.

[10] 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 *Proceedings of the ACM Symposium on Parallelism in Algorithms and Architectures*, 2009.

[11] 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," sourceforge.net/projects/xmtc/files/xmtc-documentation/, 2010.

[12] O. Certner, Z. Li, A. Raman, and O. Temam, "A very fast simulator for exploring the many-core future," in *Proceedings of the IEEE International Parallel and Distributed Processing Symposium*, 2011.

[13] 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 *Proceedings of the International Conference on Parallel Processing*, 2008.

[14] "Vishkin's supercomputer enables Ph.D. level parallel programming in high school, press release," Department of Electrical and Computer Engineering, University of Maryland, College Park, Maryland 20742, May 2012. [Online]. Available: http://www.ece.umd.edu/News/news_story.php?id=6473

[15] J. Edwards, "Can pram graph algorithms provide practical speedups on many-core machines?" Dimacs Workshop on Parallelism: A 2020 Vision, March 2011.

[16] J. A. Edwards and U. Vishkin, "Better Speedups Using Simpler Parallel Programming for Graph Connectivity and Biconnectivity," in *Proceedings of the International Workshop on Programming Models and Applications for Multicores and Manycores*, 2012, in conj. with PPoPP.

[17] J. A. Edwards and U. Vishkin, "Speedups for Parallel Graph Triconnectivity," in *Proceedings of the ACM Symposium on Parallelism in Algorithms and Architectures*, 2012, brief announcement.

[18] J. A. Edwards and U. Vishkin, "Empirical speedup study of truly parallel data compression," University of Maryland, College Park, Tech. Rep., 2013.

[19] J. A. Edwards and U. Vishkin, "Truly Parallel Burrows-Wheeler Compression and Decompression," in *Proceedings of the ACM Symposium on Parallelism in Algorithms and Architectures*, 2013, brief announcement.

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

[21] E. M. Gagnon and L. J. Hendren, "SableCC, an Object-Oriented Compiler Framework," in *Proceedings of the Technology of Object-Oriented Languages*, 1998.

[22] G. Hamerly, E. Perelman, J. Lau, and B. Calder, "Simpoint 3.0: Faster and more flexible program analysis," *Journal of Instruction Level Parallelism*, vol. 7, Sept. 2005.

[23] 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.

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

[25] W. Huang, K. Sankaranarayanan, R. J. Ribando, M. R. Stan, and K. Skadron, "An Improved Block-Based Thermal Model in HotSpot 4.0 with Granularity Considerations," in *Proceedings of the Workshop on Duplicating, Deconstructing, and Debunking*, 2007.

[26] W. Huang, M. R. Stan, K. Sankaranarayanan, R. J. Ribando, and K. Skadron, "Many-core design from a thermal perspective." in *Proceedings of the Design Automation Conference*, 2008.

[27] W. Huang, M. Stan, K. Skadron, K. Sankaranarayanan, S. Ghosh, and S. Velusamy, "Compact thermal modeling for temperature-aware design," in *Proceedings of the Design Automation Conference*, 2004.

[28] C. Isci and M. Martonosi, "Runtime power monitoring in high-end processors: Methodology and empirical data," in *Proceedings of the International Symposium on Microarchitecture*, 2003.

[29] A. B. Kahng, B. Li, L.-S. Peh, and K. Samadi, "Orion 2.0: A power-area simulator for interconnection networks," *IEEE Transactions on VLSI Systems*, 2011, to appear.

[30] F. Keceli, "Power and performance studies of the Explicit Multi-Threading (XMT) architecture," Ph.D. dissertation, University of Maryland, College Park, 2011.

[31] F. Keceli, T. Moreshet, and U. Vishkin, "Power-performance comparison of single-task driven many-cores," 2011.

[32] F. Keceli, T. Moreshet, and U. Vishkin, "Thermal Management of a Many-Core Processor under Fine-Grained Parallelism," in *Proceedings of the Workshop on Highly Parallel processing on Chip*, 2011, in conj. with Euro-Par.

[33] A. Kerr, G. Diamos, and S. Yalamanchili, "A characterization and analysis of PTX kernels," in *Proceedings of the IEEE International Symposium on Workload Characterization*, 2009.

[34] H. Lebreton and P. Vivet, "Power Modeling in SystemC at Transaction Level, Application to a DVFS Architecture," in *Proceedings of the IEEE Computer Society Annual Symposium on VLSI*, 2008.

[35] S. Li, J. H. Ahn, R. D. Strong, J. B. Brockman, D. M. Tullsen, and N. P. Jouppi, "McPAT: an integrated power, area, and timing modeling framework for multicore and manycore architectures," in *Proceedings of the International Symposium on Microarchitecture*, 2009.

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

[37] N. Muralimanohar, R. Balasubramonian, and N. P. Jouppi, "CACTI 6.0: A Tool to Model Large Caches," HP Laboratories, Tech. Rep., 2005.

[38] D. Naishlos, J. Nuzman, C.-W. Tseng, and U. Vishkin, "Evaluating the xmt parallel programming model," in *Proceedings of the International Workshop on High-Level Parallel Programming Models and Supportive Environments*, 2001.

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

[40] D. Padua and U. Vishkin, "Joint UIUC/UMD parallel algorithms/ programming course," in *Proceedings of the NSF/TCPP Workshop on Parallel and Distributed Computing Education*, 2011, in conj. with IPDPS.

[41] 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 *IASTED International Conference on Parallel and Distributed Computing and Systems*, 2009.

[42] K. Skadron, M. R. Stan, W. Huang, S. Velusamy, K. Sankaranarayanan, and D. Tarjan, "Temperature-aware microarchitecture," in *Proceedings of the International Symposium on Computer Architecture*, 2003.

[43] D. D. Sylvain Collange, Marc Daumas and D. Parello, "Barra, a modular functional GPU simulator for GPGPU," University de Perpignan, Tech. Rep., 2009.

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

[45] 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.

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

[47] D. Wang, B. Ganesh, N. Tuaycharoen, K. Baynes, A. Jaleel, and B. Jacob, "DRAMsim: a memory system simulator," *SIGARCH Computer Architecture News*, vol. 33, pp. 100–107, 2005.

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

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

[50] S. Wilton and N. Jouppi, "CACTI: an enhanced cache access and cycle time model," *IEEE Journal of Solid-State Circuits*, vol. 31, no. 5, pp. 677–688, May 1996.

[51] L. Zhao, R. Iyer, J. Moses, R. lllikkal, S. Makineni, and D. Newell, "Exploring large-scale cmp architectures using manysim," *IEEE Micro*, vol. 27, pp. 21 – 33, 2007.