Automated Modeling and Emulation of Interconnect Designs for Many-Core Chip Multiprocessors

Colin J. Ihrig, Rami Melhem, and Alex K. Jones
University of Pittsburgh
cji3@pitt.edu, melhem@cs.pitt.edu, akjones@ece.pitt.edu

ABSTRACT
Simulation of new multi- and many-core systems is becoming an increasingly large bottleneck in the design process. This paper presents the ACME design automation tool flow that facilitates the hardware emulation of newly proposed large multi-core interconnection networks on FPGAs to mitigate the slowdowns of single threaded event driven simulation. The tool is aimed at computer and network architects who have knowledge of digital design but may not be comfortable with hardware description languages and synthesis flows. ACME uses a graphical entry that allows a mix of hardware components with software algorithms written in C, each with a user defined latency and throughput in terms of system cycles. ACME automatically generates a cycle accurate hardware emulator as a Xilinx Platform Studio project, which integrates synthesized hardware blocks with embedded soft-core processors that execute the C code. Our results demonstrate that for 16-core and 64-core cycle accurate packet switching networks, the FPGA-based emulation is faster than Simics-based software simulation by 2.5x and 14.6x, respectively.

Categories and Subject Descriptors: J.6 [COMPUTER-AIDED ENGINEERING]: Computer-aided design (CAD)

General Terms: Design, Experimentation, Verification

Keywords: emulation, simulation, multi-core, many-core, interconnection network

1. INTRODUCTION
Issues such as die yield, high power density, and heat dissipation are causing a migration from large, high-clock speed uniprocessors towards systems with multiple smaller, more modest performance and power efficient processor cores. Unfortunately, current simulators cannot keep up with this change in computer architecture because their performance does not scale well to a larger number of processors [1]. Current full system simulators such as Virtutech’s Simics [2] incur a slowdown of 3-4 orders of magnitude [3] where in some cases, one minute of simulated execution time can take days of real time. This results in over a month of simulation in order to run the SPEC2k benchmark suite [4, 5]. As more cores are added to a system, this slowdown increases exponentially [3].

∗This work is partially supported by NSF award CCF-0702452

Another disadvantage of these software simulation systems is that they are complicated and difficult to modify for simulation of new interconnect strategies or cache organizations. In many cases, commercial simulators, while more robust, provide limited application programming interfaces (APIs), leaving the user to write their design from scratch. In open source simulators, the source code is often very complex, cryptic, and potentially filled with bugs, making it hard to modify. In both cases, the learning curve and development time can be steep.

To replace simulation, we propose a hardware emulation system to test and evaluate new cache and interconnect design proposals. An overview of this system is shown in Figure 1. Processor cores are executed either natively on an actual processor or through traditional software simulation. In many cases, multiple cores can be simulated on multi-core processors, natively. Rather than simulating the processors’ on-board cache and interconnect (for multi-core processors), memory accesses are captured using interrupts and sent to a Field Programmable Gate Array (FPGA) for emulation. The FPGA conducts a cycle accurate emulation of the network and returns the system time when the transaction completes.

Figure 1: Emulation system overview.

An FPGA is a natural emulation engine for caches and interconnect structures because they contain many small on-board memories, a wide and flexible interconnect, and a sea of logic for implementing routers and other control logic. Hardware acceleration using FPGAs has been shown to provide speedups of more than 10x over software for application specific tasks [6]. Standard interconnect designs and cache organizations are provided as libraries for building systems. However, for new interconnect and cache strategies this system requires a tool to translate these concepts onto the FPGA-based emulation system.

This paper introduces the Architecture Compiler for Model Em-
2. RELATED WORK

A great deal of research effort has been placed into addressing the issue of simulating large many-core systems. Cho et al. introduced a many-core processor simulation system called Two-Phase Trace-driven Simulation (TPTS) [10]. TPTS breaks simulations up into two separate phases — a trace generation phase and a trace simulation phase. The idea behind TPTS is to only incur the overhead of many events once during the trace generation phase.

The SuperESCalear Simulator (SESC) is a multi-core MIPS-based simulator that attempts to provide fast simulations for large numbers of processors [11]. SESC decouples the functional simulation from the timing simulation by executing the code for each core using the MIPS Interpreted (MINT) [12]. The timing simulation is conducted from the functional simulation output to model everything from out of order execution to cache coherence traffic. SESC is fast but not cycle accurate, and makes many questionable assumptions (such as unlimited buffers for its mesh interconnect). When correcting some of these assumptions, the simulations slow down significantly.

In [5] and [4] the SMARTS framework attempts to reduce simulation times by only sampling various simulation points of execution. Once a simulation point is reached, a detailed simulation is performed for a specified time window. After the window has elapsed, the simulator will proceed to the next simulation point. In [13] this technique is improved upon by analyzing the program first in order to determine simulation points which are most representative of program behavior.

The Research Accelerator for Multiple Processors (RAMP) project is a multi-university project aimed at creating new computer architecture research tools by using FPGAs to emulate parallel computer systems [14]. The RAMP Blue is a prototype system for multi-core emulation using FPGAs [15]. The system is comprised of 768-1,008 MicroBlaze soft core processors running in 64-84 Xilinx Virtex-II Pro 70 FPGAs on 16-21 Bee2 boards. It also consists of a software architecture which utilizes gcc, uClinux, and Unified Parallel C (UPC) [16].

In contrast to these projects, the ACME tool provides a graphical design interface for multi-core network design. Unlike software simulators such as SESC, Simics, and SMARTS, the ACME concept accelerates cycle accurate simulation through hardware emulation. Unlike RAMP, ACME attempts to retain a hardware target independent, cycle accurate simulation through special hardware structures that guarantee a particular latency and throughput in the system. Finally, ACME provides a high-level design environment to build its emulation, which particularly compared to RAMP avoids the need to (1) develop using cumbersome hardware languages and (2) manually integrate hardware and software components. ACME also does not attempt to emulate the entire system including the processor cores as in RAMP. However, ACME could be configured to target RAMP hardware.

3. ACME TOOL FLOW

This section describes the ACME tool flow from graphical entry in Ptolemy to full FPGA design generation. The system works in a synchronous fashion, where each element of the design proceeds according to an emulated target clock. The target clock is different from the host clock of the FPGA. A hardware barrier ensures that all of the components have completed the appropriate amount of work as dictated by the target clock, at which point it allows execution to continue to the next target clock cycle. This method allows soft-core processors and hardware components, each with potentially different clock domains to operate in parallel but also remain in sync and generate cycle-accurate results. This is further discussed in Section 3.1.

A high level view of this flow from Ptolemy to FPGA synthesis is shown in Figure 3 and will be referenced throughout this section. In the chart, lightly shaded regions are provided by various third parties, while the darker segments of the flow were developed by the authors.

The graphic design and simulation environment shown in the flow as Ptolemy II is based on the Ptolemy II design environment [17].
Ptolemy focuses on modeling and simulation of concurrent, real-time, embedded systems [18]. ACME utilizes the discrete-event mode of Ptolemy, as discrete-event simulation is widely used for modeling digital circuits [7].

Ptolemy models are comprised of two types of components. Atomic components, or actors, represent the lowest level designs in the system, and are implemented in Java. Basic hardware components for building network switches such as multiplexers and buffers are provided in the Java Actor Library. These actor libraries are then used to create hierarchical, or composite, components which can be stored in an XML file.

The ACME tool starts with a library of VHDL implementations of the relevant Ptolemy actors shown in VHDL Actor Library. Based on this library, the ACME Front End can read the composite model XML file and generate a synthesizable VHDL description of the model which can be run on the FPGA. In the event that a desired actor is not available in the library, the user graphically designs a black box of the new actor by adding ports and port types. The ACME Actor Generator then generates a Java and VHDL actor skeleton from the black box for which the user describes the necessary Java and VHDL behavior.

In some cases it may not be practical to design a hardware implementation of a component of the network design. For example, it might be easier to design a switch arbiter in C rather than constructing one from hardware blocks. Figure 4 shows an extension to the actor generation from Figure 3. The VHDL Based Actor Generator follows the original flow. But to allow C code to be integrated into the system we add the Processor Based Actor Generator. This allows the user to generate a “processor actor” and write their behavioral code in C, while the tool automatically generates wrapper code using the Java Native Interface (JNI) to integrate it into a Java actor skeleton. The JNI is a framework which allows Java code to inter-operate with native (platform specific) code written in another language such as C/C++ or assembly language [19]. The same C code can be executed on a soft-core processor in the FPGA design.

ACME provides the ability to specify latency and throughput parameters independent of a component’s functionality. This allows multiple copies of a component to be tested simultaneously with different latency and throughput parameters. This is also beneficial because it requires no effort by the system designer or library writer to generate performance specific hardware directly.

In ACME’s latency model, \(\alpha\), defines how many \(\delta\) (target cycles) are required for the input of a component to propagate through to the output. By default \(\alpha = 0\) for all components.

The throughput metric, \(\beta\), specifies how many \(\delta\) must elapse between changes at the inputs of a component. In the ACME model, if a new input value arrives in fewer than \(\beta\) cycles, then the old input value will be overwritten and the number of elapsed cycles will be reset to zero. \(\beta\) has a default value of zero, and \(\beta < \alpha\).

In ACME generated VHDL, latency and throughput are modeled using variations of the circuit shown in Figure 5. The circuit in Figure 5 has \(\beta = 3\) and \(\alpha = 4\). Latency is easily implemented as shift registers of length \(\alpha - \beta\). In Figure 5 the latency is represented by register C. This is equivalent to a shift register of length one.

The throughput is represented by the entire circuit, with the exception of register C, which is used to model latency. If \(\beta < 2\) then this additional logic is removed, leaving only the latency register(s).
of decoupling which can take many clock cycles to execute, necessitates a method inclusion of processors, and more specifically software programs, least significant bit becomes a ‘1’ then This bit string is right shifted by one place each cycle. When the constant value of “10” is used to reset the input value is guaranteed to appear at the output. In Figure 5, the old and new values are equivalent then the bit string is right shifted by one position. Eventually, the least significant bit of the bit string will become a ‘1’ and the new value is passed to register B.

3.1.2 Hardware Barrier Synchronization

In designs comprised solely of logic, it is perfectly acceptable to use the host clock to serve as the target clock, δ. However, the inclusion of processors, and more specifically software programs, can take many clock cycles to execute, necessitates a method of decoupling δ from the FPGA clock.

In a typical SoC, processors communicate with custom logic via SARs. In an example system, processors will write data to the SARs and signal the logic to perform some set of computations. The logic will then write the results back to the SARs and notify the processors of completion, at which point the processors can continue execution with the new data. Synchronizing the processors with the hardware is difficult due to factors such as nondeterminism in software and the use of potentially different clock frequencies. Barriers are a common synchronization construct used in many parallel programming paradigms. Barrier synchronization is also fairly simple to implement in hardware, making it a natural fit for a FPGA based system.

In [22], a simple hardware barrier circuit is described. A hardware barrier for P processors is composed of P single bit registers connected to zero detect logic. Once the barrier point is reached, each processor asserts one of the P registers to zero. The zero detect logic will assert a barrier signal once all of the processors set their corresponding bits to zero. Once the barrier is cleared, all of the processors can reset their bits and proceed with execution.

The barrier circuit used in ACME designs extends the circuit of [22]. In the original design, it was sufficient for each processor to know only that the barrier had been reached. However, in the emulation system the processors must reach the barrier point, set the barrier, notify the custom logic to execute for exactly one δ, and then reset the barrier.

The adapted circuit shown in Figure 6 utilizes both zero detect logic and one detect logic, while using the barrier output as the clock signal for the custom logic. Once the barrier is set by the processors, the zero detect logic asserts the barrier output, which acts as the rising clock edge to the custom hardware. Simultaneously, the processors are notified that the barrier has been set and can begin resetting their signals. Once the barrier has been reset, the one detect logic will unassert the barrier signal, creating a falling edge on the clock signal to the custom logic. The barrier ensures not only that all of the processors stay in sync, but also that the hardware executes for exactly one δ, where δ is dictated by the software execution and is independent of the FPGA clock frequency. Similar to the hardware blocks, processors can mimic the latency and throughput by adding a circuit like Figure 5 to the output connections of the processor that connect to the custom hardware (not shown in Figure 6).

4. CASE STUDY

In this section the power of the ACME tool is demonstrated by implementing a 4x4 packet switching mesh network using round robin arbitration, and comparing the performance of the FPGA emulator to a commercial, cycle-accurate software simulator. The top level design figure is too dense to be included in the paper, however, Figure 7 shows the schematic of a single switch point in the mesh. On the top left, there are five buffers corresponding to each cardinal direction and a local core connection. These buffers are implemented as atomic actors in Ptolemy, and are realized in the FPGA using custom logic. Just below the buffers is a processor-based round robin controller. The controller receives packets from the buffers and configures the crossbar appropriately. The crossbar is the actor on the right and is a composite actor containing a network of multiplexers.

The crossbar composite actor is expanded in Figure 8. The crossbar contains five data input ports, five control signal input ports, five data output ports, and five 4:1 multiplexers. The data input and output ports represent the North, South, East, West, and local ports of an interconnect switch. The local connection is routed to the external processor for the local core via the core interface hardware. Each input port is routable, via the multiplexers, to any other output port. During ACME synthesis, the circuit from Figure 5 is appended to the output of the VHDL representation of each actor to provide the appropriate latency and throughput values as specified in the Ptolemy design.

The control signals to the multiplexers are controlled by an actor that implements a simple round robin arbitration protocol in the following manner: In the example crossbar it is possible for every input port’s data to be passed through to an output port. However, if two or more input ports wish to send to the same output port then contention occurs. The round robin policy gives each input port a priority for sending data. In the event of contention, the input port with the higher priority is allowed to send its data while the other port(s) must wait until the next cycle. After each cycle, each input port receives a higher priority and the input port that previously had the highest priority is demoted to the lowest priority.
Figure 7: Switch point composite design.

Ptolemy actors contain the following three most important methods, `init()`, `prefire()`, and `fire()`. `init()` initializes the actor to a particular start state, `prefire()` looks at messages received by the actor and decides if the actor needs to take any action (e.g., call `fire()`), and `fire()` executes the behavior of the system. As processor actors require the behavior to be written in C, the JNI is used to create C versions of the functions `init()`, `prefire()`, and `fire()`.

Figure 9 shows the `jniinit()` method which initializes the round robin policy to assign the highest priority to the North input port. The South, East, West, and local ports are each assigned a progressively lower priority. In this example, the `jniprefire()` method does not perform any special checks and always returns true. The `jnifire()` method, shown in Figure 10, determines the ports that will be allowed to send data based on priority and any conflicts that exist. First, the highest priority queue is found by checking the priority array in the `for` loop. Each queue is accessed in priority order, and the destination of each queue’s message is checked using the `get_buff_dest(queue)` method. If that destination has not been used by another message (checked in array `q`), then a connection from that input port to the requested output port is scheduled using the `schedule(src, dest)` method. As the queues are checked, the priority is updated by increasing the priority by one with a modulo five to wrap around back to zero when necessary.

The full system was implemented using the ACME design flow and in Simics [2] using a cycle accurate network simulator. Only the actual interconnection network was emulated in hardware. The remainder of the system (cores and caches) was simulated in Simics. The systems contained 16 and 64 cores operating at a frequency of 2 GHz, and a network clocked at 1 GHz. The cores have a private L1 cache and a non-uniform cache access distributed shared L2 cache. The simulation assumes a switch latency of two cycles with a throughput of one message per cycle. The arbitration using round robin scheduling was set to one cycle. To test the system, the Raytrace benchmark from the SPLASH-2 multi-threaded benchmark suite [23] was run.

To allow for reasonable simulation testing times, between 2,500 and 5,000 target cycles from the parallel section of the application were run. Messages were provided to the ACME design through traces generated by Simics. For the 16-core (4x4) mesh, a 1,000 message segment of the Raytrace benchmark was run, resulting in approximately 2,600 target cycles of executing. For the 64-core (8x8) mesh, a 5,000 message segment was run, requiring approximately 4,000 target cycles. The execution times of the network simulator are shown in Table 1. Ptolemy refers to running the design within the Ptolemy II framework. Simics refers to the time of executing the network simulator connected to a Simics simulation. Finally, FPGA is the interconnect emulation time. The Ptolemy and Simics simulations were run on a Intel Xeon 8 core machine (dual-quad core “Clovertown” processors) operating at 2.3 GHz with 8 GB of RAM. The emulation was executed on a Xilinx Virtex 5 110T operating at 100 MHz. Speedups of emulation over the other two implementations are shown in Table 2.
5. CONCLUSIONS AND FUTURE WORK

This paper has presented a tool flow for the creation and emulation of interconnection networks for multi-core systems. The Ptolemy modeling infrastructure is used to design these systems. From there, the ACME tool flow is able to generate a description of the system, which is then synthesized using commercial FPGA tools. Additionally, the components used in the system can be parameterized in terms of latency and throughput. Sophisticated component interfaces are easily created graphically, while their actual implementations are written in C code and integrated into the Ptolemy simulation engine via the JNI.

Benchmarking tests demonstrate that the ACME-based approach can generate a viable network-on-chip emulator with a substantial speedup (more than 10x) over a software system, particularly as the number of processor cores scales beyond 16. Future directions of this work include the creation of large ACME component libraries, integration of shared cache hierarchies into the FPGA emulation, targeting of multi-FPGA systems, and automated integration with software-based simulators such as Simics. Further studies are also needed to determine the ideal partitioning of the systems between software simulation and hardware emulation in order to minimize the potential bottleneck of host to FPGA communication.

6. REFERENCES