

Multifacet's General Execution-driven Multiprocessor Simulator (GEMS) Toolset

Milo M. K. Martin¹, Daniel J. Sorin², Bradford M. Beckmann³, Michael R. Marty³, Min Xu⁴,
Alaa R. Alameldeen³, Kevin E. Moore³, Mark D. Hill^{3,4}, and David A. Wood^{3,4}

<http://www.cs.wisc.edu/gems/>

¹Computer and Information
Sciences Dept.
Univ. of Pennsylvania

²Electrical and Computer
Engineering Dept.
Duke Univ.

³Computer Sciences Dept.
Univ. of Wisconsin-Madison

⁴Electrical and Computer
Engineering Dept.
Univ. of Wisconsin-Madison

Abstract

The Wisconsin Multifacet Project has created a simulation toolset to characterize and evaluate the performance of multiprocessor hardware systems commonly used as database and web servers. We leverage an existing full-system functional simulation infrastructure (Simics [14]) as the basis around which to build a set of timing simulator modules for modeling the timing of the memory system and microprocessors. This simulator infrastructure enables us to run architectural experiments using a suite of scaled-down commercial workloads [3]. To enable other researchers to more easily perform such research, we have released these timing simulator modules as the Multifacet General Execution-driven Multiprocessor Simulator (GEMS) Toolset, release 1.0, under GNU GPL [9].

1 Introduction

Simulation is one of the most important techniques used by computer architects to evaluate their innovations. Not only does the target machine need to be simulated with sufficient detail, but it also must be driven with a realistic workload. For example, SimpleScalar [4] has been widely used in the architectural research community to evaluate new ideas. However it and other similar simulators run only user-mode, single-threaded workloads such as the SPEC CPU benchmarks [24]. Many designers are interested in multiprocessor systems that run more complicated, multithreaded workloads such as databases, web servers, and parallel scientific codes. These workloads depend upon

many operating system services (e.g., I/O, synchronization, thread scheduling and migration). Furthermore, as the single-chip microprocessor evolves to a chip-multiprocessor, the ability to simulate these machines running realistic multithreaded workloads is paramount to continue innovation in architecture research.

Simulation Challenges. Creating a timing simulator for evaluating multiprocessor systems with workloads that require operating system support is difficult. First, creating even a functional simulator, which provides no modeling of timing, is a substantial task. Providing sufficient functional fidelity to boot an unmodified operating system requires implementing supervisor instructions, and interfacing with functional models of many I/O devices. Such simulators are called full-system simulators [14, 20]. Second, creating a detailed timing simulator that executes only user-level code is a substantial undertaking, although the wide availability of such tools reduces redundant effort. Finally, creating a simulation toolset that supports both full-system and timing simulation is substantially more complicated than either endeavor alone.

Our Approach: Decoupled Functionality and Timing Simulation. To address these simulation challenges, we designed a modular simulation infrastructure (GEMS) that decouples simulation functionality and timing. To expedite simulator development, we used Simics [14], a full-system functional simulator, as a foundation on which various timing simulation modules can be dynamically loaded. By decoupling functionality and timing simulation in GEMS, we leverage both the efficiency and the robustness of a functional simu-

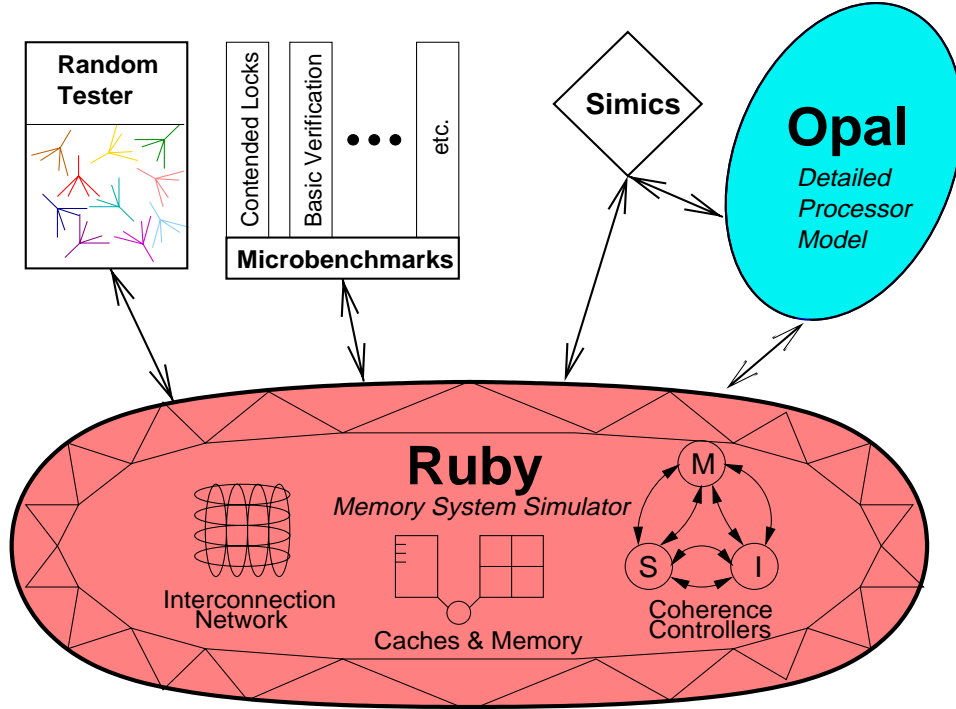


Figure 1. A view of the GEMS architecture: Ruby, our memory simulator can be driven by one of four memory system request generators.

lator. Using modular design provides the flexibility to simulate various system components in different levels of detail.

While some researchers used the approach of adding full-system simulation capabilities to existing user-level-only timing simulators [6, 21], we adopted the approach of leveraging an existing full-system functional simulation environment (also used by other simulators [7, 22]). This strategy enabled us to begin workload development and characterization in parallel with development of the timing modules. This approach also allowed us to perform initial evaluation of our research ideas using a more approximate processor model while the more detailed processor model was still in development.

We use the *timing-first simulation* approach [18], in which we decoupled the functional and timing aspects of the simulator. Since Simics is robust enough to boot an unmodified OS, we used its functional simulation to help us avoid implementing rare but effort-consuming instructions in the timing simulator. Our timing modules interact with Simics to determine *when* Simics should execute an instruction. However, *what* the result of the

execution of the instruction is ultimately dependent on Simics. Such a decoupling allows the timing models to focus on the most common 99.9% of all dynamic instructions. This task is much easier than requiring a monolithic simulator to model the timing *and* correctness of every function in all aspects of the full-system simulation. For example, a small mistake in handling an I/O request or in modeling a special case in floating point arithmetic is unlikely to cause a significant change in timing fidelity. However, such a mistake will likely affect functional fidelity, which may prevent the simulation from continuing to execute. By allowing Simics to always determine the result of execution, the program will always continue to execute correctly.

Our approach is different from *trace-driven* simulation. Although our approach decouples functional simulation and timing simulation, the functional simulator is still affected by the timing simulator, allowing the system to capture timing-dependent effects. For example, the timing model will determine the winner of two processors that are trying to access the same software lock in memory. Since the timing simulator determines when the functional simulator advances, such tim-

ing-dependent effects are captured. In contrast, trace-driven simulation fails to capture these important effects. In the limit, if our timing simulator was 100% functionally correct, it would always agree with the functional simulator, making the functional simulation redundant. Such an approach allows for “correctness tuning” during simulator development.

Design Goals. As the GEMS simulation system has primarily been used to study cache-coherent shared memory systems (both on-chip and off-chip) and related issues, those aspects of GEMS release 1.0 are the most detailed and the most flexible. For example, we model the transient states of cache coherence protocols in great detail. However, the tools have a more approximate timing model for the interconnection network, a simplified DRAM sub-system, and a simple I/O timing model. Although we do include a detailed model of a modern dynamically-scheduled processor, our goal was to provide a more realistic driver for evaluating the memory system. Therefore, our processor model may lack some details, and it may lack flexibility that is more appropriate for certain detailed micro-architectural experiments.

Availability. The first release of GEMS is available at <http://www.cs.wisc.edu/gems/>. GEMS is open-source software and is licensed under GNU GPL [9]. However, GEMS relies on Virtutech’s Simics, a commercial product, for full-system functional simulation. At this time, Virtutech provides evaluation licenses for academic users at no charge. More information about Simics can be found at <http://www.virtutech.com/>.

The remainder of this paper provides an overview of GEMS (Section 2) and then describes the two main pieces of GEMS: the multiprocessor memory system timing simulator *Ruby* (Section 3), which includes the *SLICC* domain-specific language for specifying cache-coherence protocols and systems, and the detailed microarchitectural processor timing model *Opal* (Section 4). Section 5 discusses some constraints and caveats of GEMS, and we conclude in Section 6.

2 GEMS Overview

The heart of GEMS is the Ruby memory system simulator. As illustrated in Figure 1, GEMS pro-

vides multiple drivers that can serve as a source of memory operation requests to Ruby:

1) **Random tester module:** The simplest driver of Ruby is a random testing module used to stress test the corner cases of the memory system. It uses false sharing and action/check pairs to detect many possible memory system and coherence errors and race conditions [25]. Several features are available in Ruby to help debug the modeled system including deadlock detection and protocol tracing.

2) **Micro-benchmark module:** This driver supports various micro-benchmarks through a common interface. The module can be used for basic timing verification, as well as detailed performance analysis of specific conditions (e.g., lock contention or widely-shared data).

3) **Simics:** This driver uses Simics’ functional simulator to approximate a simple in-order processor with no pipeline stalls. Simics passes all load, store, and instruction fetch requests to Ruby, which performs the first level cache access to determine if the operation hits or misses in the primary cache. On a hit, Simics continues executing instructions, switching between processors in a multiple processor setting. On a miss, Ruby stalls Simics’ request from the issuing processor, and then simulates the cache miss. Each processor can have only a single miss outstanding, but contention and other timing affects among the processors will determine when the request completes. By controlling the timing of when Simics advances, Ruby determines the timing-dependent functional simulation in Simics (e.g., to determine which processor next acquires a memory block).

4) **Opal:** This driver models a dynamically-scheduled SPARC v9 processor and uses Simics to verify its functional correctness. Opal (previously known as TFSim[18]) is described in more detail later in Section 4.

The first two drivers are part of a stand-alone executable that is independent of Simics or any actual simulated program. In addition, Ruby is specifically designed to support additional drivers (beyond the four mentioned above) using a well-defined interface.

GEMS’ modular design provides significant simulator configuration flexibility. For example,

our memory system simulator is independent of our out-of-order processor simulator. A researcher can obtain preliminary results for a memory system enhancement using the simple in-order processor model provided by Simics, which runs much faster than Opal. Based on these preliminary results, the researcher can then determine whether the accompanying processor enhancement should be implemented and simulated in the detailed out-of-order simulator.

GEMS also provides flexibility in specifying many different cache coherence protocols that can be simulated by our timing simulator. We separated the protocol-dependent details from the protocol-independent system components and mechanisms. To facilitate specifying different protocols and systems, we proposed and implemented the protocol specification language SLICC (Section 3.2). In the next two sections, we describe our two main simulation modules: Ruby and Opal.

3 Multiprocessor Memory System (Ruby)

Ruby is a timing simulator of a multiprocessor memory system that models: caches, cache controllers, system interconnect, memory controllers, and banks of main memory. Ruby combines hard-coded timing simulation for components that are largely independent of the cache coherence protocol (e.g., the interconnection network) with the ability to specify the protocol-dependent components (e.g., cache controllers) in a domain-specific language called SLICC (Specification Language for Implementing Cache Coherence).

Implementation. Ruby is implemented in C++ and uses a queue-driven event model to simulate timing. Components communicate using message buffers of varying latency and bandwidth, and the component at the receiving end of the buffer is scheduled to wake up when the next message will be available to be read from the buffer. Although many buffers are used in a strictly first-in-first-out (FIFO) manner, the buffers are not restricted to FIFO-only behavior. The simulation proceeds by invoking the wakeup method for the next scheduled event on the event queue. Conceptually, the simulation would be identical if all components were woken up each cycle; thus, the event queue can be thought of as an optimization to avoid unnecessary processing during each cycle.

3.1 Protocol-Independent Components

The protocol-independent components of Ruby include the interconnection network, cache arrays, memory arrays, message buffers, and assorted glue logic. The only two components that merit discussion are the caches and interconnection network.

Caches. Ruby models a hierarchy of caches associated with each single processor, as well as shared caches used in chip multiprocessors (CMPs) and other hierarchical coherence systems. Cache characteristics, such as size and associativity, are configuration parameters.

Interconnection Network. The interconnection network is the unified communication substrate used to communicate between cache and memory controllers. A single monolithic interconnection network model is used to simulate all communication, even between controllers that would be on the same chip in a simulated CMP system. As such, all intra-chip and inter-chip communication is handled as part of the interconnect, although each individual link can have different latency and bandwidth parameters. This design provides sufficient flexibility to simulate the timing of almost any kind of system.

A controller communicates by sending messages to other controllers. Ruby's interconnection network models the timing of the messages as they traverse the system. Messages sent to multiple destinations (such as a broadcast) use traffic-efficient multicast-based routing to fan out the request to the various destinations.

Ruby models a point-to-point switched interconnection network that can be configured similarly to interconnection networks in current high-end multiprocessor systems, including both directory-based and snooping-based systems. For simulating systems based on directory protocols, Ruby release 1.0 supports three non-ordered networks: a simplified full connected point-to-point network, a dynamically-routed 2D-torus interconnect inspired by the Alpha 21364 [19], and a flexible user-defined network interface. The first two networks are automatically generated using certain simulator configuration parameters, while the third creates an arbitrary network by reading a user-defined configuration file. This file-specified network can create

complicated networks such as a CMP-DNUCA network [5].

For snooping-based systems, Ruby has two totally-ordered networks: a crossbar network and a hierarchical switch network. Both ordered networks use a hierarchy of one or more switches to create a total order of coherence requests at the network's root. This total order is enough for many broadcast-based snooping protocols, but it requires that the specific cache-coherence protocol does not rely on stronger timing properties provided by the more traditional bus-based interconnect. In addition, mechanisms for synchronous snoop response combining and other aspects of some bus-based protocols are not supported.

The topology of the interconnect is specified by a set of links between switches, and the actual routing tables are re-calculated for each execution, allowing for additional topologies to be easily added to the system. The interconnect models virtual networks for different types and classes of messages, and it allows dynamic routing to be enabled or disabled on a per-virtual-network basis (to provide point-to-point order if required). Each link of the interconnect has limited bandwidth, but the interconnect does not model the details of the physical or link-level layer. By default, infinite network buffering is assumed at the switches, but Ruby also supports finite buffering in certain networks. We believe that Ruby's interconnect model is sufficient for coherence protocol and memory hierarchy research, but a more detailed model of the interconnection network may need to be integrated for research focusing on low-level interconnection network issues.

3.2 Specification Language for Implementing Cache Coherence (SLICC)

One of our main motivations for creating GEMS was to evaluate different coherence protocols and coherence-based prediction. As such, flexibility in specifying cache coherence protocols was essential. Building upon our earlier work on table-driven specification of coherence protocols [23], we created SLICC (Specification Language for Implementing Cache Coherence), a domain-specific language that codifies our table-driven methodology.

SLICC is based upon the idea of specifying individual controller state machines that represent system components such as cache controllers and directory controllers. Each controller is conceptually a per-memory-block state machine, which includes:

- States: set of possible states for each cache block,
- Events: conditions that trigger state transitions, such as message arrivals,
- Transitions: the cross-product of states and events (based on the state and event, a transition performs an atomic sequence of actions and changes the block to a new state), and
- Actions: the specific operations performed during a transition.

For example, the SLICC code might specify a "Shared" state that allows read-only access for a block in a cache. When an external invalidation message arrives at the cache for a block in Shared, it triggers an "Invalidation" event, which causes a "Shared x Invalidation" transition to occur. This transition specifies that the block should change to the "Invalid" state. Before a transition can begin, all required resources must be available. This check prevents mid-transition blocking. Such resource checking includes available cache frames, in-flight transaction buffer, space in an outgoing message queue, etc. This resource check allows the controller to always complete the entire sequence of actions associated with the transition without blocking.

SLICC is syntactically similar to C or C++, but it is intentionally limited to constrain the specification to hardware-like structures. For example, no local variables or loops are allowed in the language. We also added special language constructs for inserting messages into buffers and reading information from the next message in a buffer.

Each controller specified in SLICC consists of protocol-independent components, such as cache memories and directories, as well as all fields in: caches, per-block directory information at the home node, in-flight transaction buffers, messages, and any coherence predictors. These fields consist of primitive types such as addresses, bit-fields, sets, counters, and user-specified enumerations. Mes-

sages contain a message type tag (for statistics gathering) and a size field (for simulating contention on the interconnection network). A controller uses these messages to communicate with other controllers. Messages travel along the intra-chip and inter-chip interconnection networks. When a message arrives at its destination, it generates a specific type of event determined by the input message control logic of the particular controller (also specified in SLICC).

SLICC allows for the specification of many types of invalidation-based cache coherence protocols and systems. As invalidation-based protocols are ubiquitous in current commercial systems, we constructed SLICC to perform all operations on cache block granularity (configurable, but canonically 64 bytes). As such, the word-level granularity required for update-based protocols is currently not supported. SLICC is perhaps best suited for specifying directory-based protocols (e.g., the protocols used in the Stanford DASH [13] and the SGI Origin [12]), and other related protocols such as AMD's Opteron protocol [1, 10]. Although SLICC can be used to specify broadcast snooping protocols, SLICC assumes all protocols use an asynchronous point-to-point network, and not the simpler (but less scalable) synchronous system bus. The GEMS release 1.0 distribution contains a SLICC specification for an aggressive snooping protocol, a flat directory protocol, a protocol based on the AMD Opteron [1, 10], two hierarchical directory protocols suitable for CMP systems, and a Token Coherence protocol [16] for a hierarchical CMP system [17].

The SLICC compiler translates a SLICC specification into C++ code that links with the protocol-independent portions of the Ruby memory system simulator. In this way, Ruby and SLICC are tightly integrated to the extent of being inseparable. In addition to generating code for Ruby, the SLICC language is intended to be used for a variety of purposes. First, the SLICC compiler generates HTML-based tables as documentation for each controller. This concise and continuously-updated documentation is helpful when developing and debugging protocols. Example SLICC code and corresponding HTML tables can be found online [15]. Second, the SLICC code has also served as the basis for translating protocols into a model-

checkable format such as TLA+ [2, 11] or Murphi [8]. Although such efforts have thus far been manual translations, we are hopeful the process can be partially or fully automated in the future. Finally, we have restricted SLICC in ways (e.g., no loops) in which we believe will allow automatic translation of a SLICC specification directly into a synthesizable hardware description language (such as VHDL or Verilog). Such efforts are future work.

3.3 Ruby's Release 1.0 Limitations

Most of the limitations in Ruby release 1.0 are specific to the implementation and not the general framework. For example, Ruby release 1.0 supports only physically-indexed caches, although support for indexing the primary caches with virtual addresses could be added. Also, Ruby does not model the memory system traffic due to direct memory access (DMA) operations or memory-mapped I/O loads and stores. Instead of modeling these I/O operations, we simply count the number that occur. For our workloads, these operations are infrequent enough (compared to cache misses) to have negligible relative impact on our simulations. Those researchers who wish to study more I/O intensive workloads may find it necessary to model such effects.

4 Detailed Processor Model (Opal)

Although GEMS can use Simics' functional simulator as a driver that approximates a system with simple in-order processor cores, capturing the timing of today's dynamically-scheduled superscalar processors requires a more detailed timing model. GEMS includes Opal—also known as TFSim [18]—as a detailed timing model using the timing-first approach. Opal runs ahead of Simics' functional simulation by fetching, decoding, predicting branches, dynamically scheduling, executing instructions, and speculatively accessing the memory hierarchy. When Opal has determined that the time has come for an instruction to retire, it instructs the functional simulation of the corresponding Simics processor to advance one instruction. Opal then compares its processor states with that of Simics to ensure that it executed the instruction correctly. The vast majority of the time Opal and Simics agree on the instruction execution; however, when an interrupt, I/O operation, or rare

kernel-only instruction not implemented by Opal occurs, Opal will detect the discrepancy and recover as necessary. The paper on TFSim/Opal [18] provides more discussion of the timing-first philosophy, implementation, effectiveness, and related work.

Features. Opal models a modern dynamically-scheduled, superscalar, deeply-pipelined processor core. Opal is configured by default to use a two-level gshare branch predictor, MIPS R10000 style register renaming, dynamic instruction issue, multiple execution units, and a load/store queue to allow for out-of-order memory operations and memory bypassing. As Opal simulates the SPARC ISA, condition codes and other architectural state are renamed as necessary to allow for highly-concurrent execution. Because Opal runs ahead of the Simics functional processor, it models all wrong-path effects of instructions that are not eventually retired. Opal implements an aggressive implementation of sequential consistency, allowing memory operations to occur out of order and detecting possible memory ordering violations as necessary.

Limitations. Opal is sufficient for modeling a processor that generates multiple outstanding misses for the Ruby memory system. However, Opal's microarchitectural model, although detailed, does not model all the most advanced features of some modern processors (e.g., a trace cache and advanced memory-dependence speculation predictors). Thus, the model may need to be extended and further validated in order to be used as a basis for experiments that focus on the microarchitecture of the system. Also, the first release of Opal does not support hardware multithreading, but at least one of our internal projects has preliminarily added such support to Opal. Similar to most microarchitecture simulators, Opal is heavily tied to its target ISA (in this case, SPARC), and porting it to a different ISA, though possible, would not be trivial.

Perhaps Opal's biggest constraints deal with its dependence on the Simics functional execution mode. For instance, because Simics uses a flat memory image, Opal cannot execute a non-sequentially consistent execution. Also we developed Opal to simulate systems based on the SPARC ISA, and therefore it is limited to the specific TLB configurations supported by Simics and the simulated operating system.

5 Constraints and Caveats

As with any complicated research tool, GEMS is not without its constraints and caveats. One caveat of GEMS is that although individual components and some entire system timing testing and sanity checking has been performed, a full end-to-end validation of GEMS has not been performed. For instance, through the random tester we have verified Ruby coherently transfers cache blocks, however, we have not performed exhaustive timing verification or verified that it strictly adheres to the sequential consistency memory model. Nevertheless, the relative performance comparisons generated by GEMS should suffice to provide insight into many types of proposed design enhancements.

Another caveat of GEMS is that we are unable to distribute our commercial workload suite [3], as it contains proprietary software that cannot be redistributed. Although commercial workloads can be set up under Simics, the lack of ready-made workloads will increase the effort required to use GEMS to generate timing results for commercial servers and other multiprocessing systems.

6 Conclusions

In this paper we presented Multifacet's General Execution-driven Multiprocessor Simulator (GEMS) as a simulation toolset to evaluate multiprocessor architectures. By using the full-system simulator Simics, we were able to decouple the development of the GEMS timing model from ensuring the necessary functional correctness required to run an unmodified operating system and commercial workloads. The multiprocessor timing simulator Ruby allows for detailed simulation of cache hierarchies. The domain-specific language SLICC grants GEMS the flexibility to implement different coherence protocols and systems under a single simulation infrastructure. The processor timing simulator Opal can be used to simulate a dynamically-scheduled superscalar processor, and it relies on Simics for functional correctness. To enable others to more easily perform research on multiprocessors with commercial workloads, we have released GEMS under the GNU GPL, and the first release is available at <http://www.cs.wisc.edu/gems/>.

7 Acknowledgments

We thank Ross Dickson, Pacia Harper, Carl Mauer, Manoj Plakal, and Luke Yen for their contributions to the GEMS infrastructure. We thank the University of Illinois for original beta testing. This work is supported in part by the National Science Foundation (CCR-0324878, EIA/CNS-0205286, and CCR-0105721) and donations from Compaq, IBM, Intel Corporation and Sun Microsystems. We thank Amir Roth for suggesting the acronym SLICC. We thank Virtutech AB, the Wisconsin Condor group, and the Wisconsin Computer Systems Lab for their help and support.

References

- [1] Ardsheer Ahmed, Pat Conway, Bill Hughes, and Fred Weber. AMD Opteron Shared Memory MP Systems. In *Proceedings of the 14th HotChips Symposium*, August 2002.
- [2] Homayoon Akhiani, Damien Doligez, Paul Harter, Leslie Lamport, Joshua Scheid, Mark Tuttle, and Yuan Yu. Cache Coherence Verification with TLA+. In *FM'99—Formal Methods, Volume II*, volume 1709 of *Lecture Notes in Computer Science*, page 1871. Springer Verlag, 1999.
- [3] Alaa R. Alameldeen, Milo M. K. Martin, Carl J. Mauer, Kevin E. Moore, Min Xu, Daniel J. Sorin, Mark D. Hill, and David A. Wood. Simulating a \$2M Commercial Server on a \$2K PC. *IEEE Computer*, 36(2):50–57, February 2003.
- [4] Todd Austin, Eric Larson, and Dan Ernst. SimpleScalar: An Infrastructure for Computer System Modeling. *IEEE Computer*, 35(2):59–67, February 2002.
- [5] Bradford M. Beckmann and David A. Wood. Managing Wire Delay in Large Chip-Multiprocessor Caches. In *Proceedings of the 37th Annual IEEE/ACM International Symposium on Microarchitecture*, December 2004.
- [6] Nathan L. Binkert, Erik G. Hallnor, and Steven K. Reinhardt. Network-Oriented Full-System Simulation using M5. In *Proceedings of the Sixth Workshop on Computer Architecture Evaluation Using Commercial Workloads*, February 2003.
- [7] Harold W. Cain, Kevin M. Lepak, Brandon A. Schwartz, and Mikko H. Lipasti. Precise and Accurate Processor Simulation. In *Proceedings of the Fifth Workshop on Computer Architecture Evaluation Using Commercial Workloads*, pages 13–22, February 2002.
- [8] David L. Dill, Andreas J. Drexler, Alan J. Hu, and C. Han Yang. Protocol Verification as a Hardware Design Aid. In *International Conference on Computer Design*. IEEE, October 1992.
- [9] Free Software Foundation. GNU General Public License (GPL). <http://www.gnu.org/copyleft/gpl.html>.
- [10] Chetana N. Keltcher, Kevin J. McGrath, Ardsheer Ahmed, and Pat Conway. The AMD Opteron Processor for Multiprocessor Servers. *IEEE Micro*, 23(2):66–76, March–April 2003.
- [11] Leslie Lamport. *Specifying Systems: The TLA+ Language and Tools for Hardware and Software Engineers*. Addison-Wesley, 2002.
- [12] James Laudon and Daniel Lenoski. The SGI Origin: A ccNUMA Highly Scalable Server. In *Proceedings of the 24th Annual International Symposium on Computer Architecture*, pages 241–251, June 1997.
- [13] Daniel Lenoski, James Laudon, Kourosh Gharachorloo, Anoop Gupta, and John Hennessy. The Directory-Based Cache Coherence Protocol for the DASH Multiprocessor. In *Proceedings of the 17th Annual International Symposium on Computer Architecture*, pages 148–159, May 1990.
- [14] Peter S. Magnusson et al. Simics: A Full System Simulation Platform. *IEEE Computer*, 35(2):50–58, February 2002.
- [15] Milo M. K. Martin et al. Protocol Specifications and Tables for Four Comparable MOESI Coherence Protocols: Token Coherence, Snooping, Directory, and Hammer. http://www.cs.wisc.edu/multifacet/theses/milo_martin_phd/, 2003.
- [16] Milo M. K. Martin, Mark D. Hill, and David A. Wood. Token Coherence: A New Framework for Shared-Memory Multiprocessors. *IEEE Micro*, 23(6), Nov/Dec 2003.
- [17] Michael R. Marty, Jesse D. Bingham, Mark D. Hill, Alan J. Hu, Milo M. K. Martin, and David A. Wood. Improving Multiple-CMP Systems Using Token Coherence. In *Proceedings of the Eleventh IEEE Symposium on High-Performance Computer Architecture*, February 2005.
- [18] Carl J. Mauer, Mark D. Hill, and David A. Wood. Full System Timing-First Simulation. In *Proceedings of the 2002 ACM Sigmetrics Conference on Measurement and Modeling of Computer Systems*, pages 108–116, June 2002.
- [19] Shubhendu S. Mukherjee, Peter Bannon, Steven Lang, Aaron Spink, and David Webb. The Alpha 21364 Network Architecture. In *Proceedings of the 9th Hot Interconnects Symposium*, August 2001.
- [20] Mendel Rosenblum, Stephen A. Herrod, Emmett Witchel, and Anoop Gupta. Complete Computer System Simulation: The SimOS Approach. *IEEE Parallel and Distributed Technology: Systems and Applications*, 3(4):34–43, 1995.
- [21] Lambert Schaelicke and Mike Parker. ML-RSIM Reference Manual. Technical Report tech. report 02-10, Department of Computer Science and Engineering, Univ. of Notre Dame, Notre Dame, IN, 2002.
- [22] Jared Smolens, Brian Gold, Jangwoo Kim, Babak Falsafi, James C. Hoe, , and Andreas G. Nowatzky. Fingerprinting: Bounding the Soft-Error Detection Latency and Bandwidth. In *Proceedings of the Eleventh International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 224–234, October 2004.
- [23] Daniel J. Sorin, Manoj Plakal, Mark D. Hill, Anne E. Condon, Milo M. K. Martin, and David A. Wood. Specifying and Verifying a Broadcast and a Multicast Snooping Cache Coherence Protocol. *IEEE Transactions on Parallel and Distributed Systems*, 13(6):556–578, June 2002.
- [24] Systems Performance Evaluation Cooperation. SPEC Benchmarks. <http://www.spec.org>.
- [25] David A. Wood, Garth A. Gibson, and Randy H. Katz. Verifying a Multiprocessor Cache Controller Using Random Test Generation. *IEEE Design and Test of Computers*, pages 13–25, August 1990.