The M5 Simulator: Modeling Networked Systems

Nathan L. Binkert Ronald G. Dreslinski Lisa R. Hsu Kevin T. Lim

Ali G. Saidi Steven K. Reinhardt

Advanced Computer Architecture Laboratory, EECS Department
The University of Michigan
2260 Hayward Ave
Ann Arbor, MI 48109-2121

{binkertn,rdreslin,hsul,ktlim,saidi,stever}@eecs.umich.edu

1 Abstract

TCP/IP networking is an increasingly important aspect of computer systems, but a lack of simulation tools limits architects' ability to explore new designs for network I/O. We have developed the M5 simulator specifically to enable research in this area. In addition to typical architecture simulator attributes, M5 provides features necessary for simulating networked hosts, including full-system capability, a detailed I/O subsystem, and the ability to simulate multiple networked systems deterministically. Our experience in simulating network workloads revealed some unexpected interactions between TCP and the common simulation acceleration techniques of sampling and warm-up. We have successfully validated M5's simulated performance results against real machines, indicating that our models and methodology adequately capture the salient characteristics of these systems. M5's usefulness as a general-purpose architecture simulator and its liberal open-source license have led to its adoption by several other academic and commercial groups.

2 Keywords

computer architecture, simulation, simulation software, interconnected systems

3 Introduction

Network I/O is an increasingly significant component of modern computer systems. Software trends toward net-centric applications drive the commercial importance of TCP/IP networking, while hardware trends toward ever higher Ethernet bandwidths continually raise the level of technical challenge involved. Nevertheless, this aspect of system design has received relatively little attention, particularly in academia, due to the complexity of analyzing all the hardware and software components involved in even the simplest TCP/IP networking environments.

The key requirements for performance modeling of networked systems are the ability to execute operating system as well as application code (known as "full-system" simulation), as this is where the bulk of the network protocol implementation resides; detailed performance models of the memory and I/O subsystems, including network interface devices; and the ability to model multiple networked systems (e.g., a server and one or more clients) in a deterministic fashion. Traditional CPU-centric simulators, such as SimpleScalar [BAB96], lack nearly all of these features. Among the simulators available when we began

our research, only SimOS [RHWG95] provided the majority of the features necessary, missing only the ability to deterministically model multiple systems. While SimOS could be extended to add this capability, it does not have support from the authors or an active user community and its software design is not easily extensible. We were thus motivated to develop a new simulation infrastructure, known as M5, to satisfy our needs.

M5 continues to be, to our knowledge, the only simulator that meets all the requirements for studying architectures for network I/O. GEMS [MSB+05] and SimFlex [HSW+04], two recent simulators developed concurrently with M5, have detailed multiprocessor memory systems but lack detailed I/O models and multiple-system capability. In addition, they both rely on SimICS [MCE+02], a commercial functional simulator, to provide much of the I/O and privileged mode modeling. This approach reduces development effort, but the resulting black-box nature of the functional model restricts flexibility, and licensing restrictions raise barriers to widespread use, particularly for non-academics.

Because we use M5 to model complex multi-system networks with a variety of components, our design focused on providing modularity and flexibility to manage this complexity. M5's relatively clean software architecture and advanced feature set have made it a powerful and flexible simulation tool even for architects who are not focused on network I/O.

4 Simulation Core

At the core of M5 is a generic, object-oriented discrete-event simulation infrastructure. This foundation includes scaffolding for defining, parameterizing, instantiating, and checkpointing simulation objects; an interface for building system models from complex collections of objects; a powerful statistics collection package; and significant debugging support.

A simulation consists of a collection of objects that interact directly via method calls. Each object is responsible for managing its own timing by scheduling its own events on the global event queue. In contrast to other simulators such as ASIM [EAB+02] that integrate timing with inter-object communication, we find that M5's approach enables individual object models to make local tradeoffs between performance and detail by determining the granularity with which they schedule their internal events. External inter-object delays are easily added when desired by interposing bus or delay buffer objects.

M5's use of object-oriented (OO) programming techniques provides a clear interface between a simula-

tion object and the rest of the system. The benefits are threefold. First, researchers can modify a component's behavior using only localized code changes with a higher likelihood of not breaking seemingly unrelated parts of the simulator. Second, realistic modeling is encouraged because violations of software modularity often indicate violations of hardware modularity (and thus feasibility). Third, different models for a particular component, such as a CPU, can be substituted easily within a particular configuration. These interchangeable models may differ in level of detail (allowing simulation speed vs. accuracy trade-offs), in the component's functional behavior (to study design alternatives), or both.

Another benefit of OO modeling is that an object encapsulates all of a component's state, eliminating nearly all global variables. Users can thus easily replicate both individual object models and entire collections of objects. Simple multiprocessors can be constructed by instantiating multiple CPU objects and connecting them to a shared bus. A simulated system is merely a collection of objects (CPUs, memory, devices, etc.), so we can model a client-server network by instantiating two of these collections and interconnecting them with a network link object. Note that all of these objects are in the same simulation process and share the same global event queue, satisfying our requirement for deterministic multi-system simulation.

M5 is implemented using two OO languages: Python for high-level object configuration and simulation scripting, where flexibility and ease of programming are of concern, and C++ for low-level object implementation, where performance matters. All simulation objects (CPUs, busses, caches, etc.) are represented as objects in both Python and C++. Using Python objects for configuration allows flexible script-based object composition to describe complex simulation targets. Once the configuration is constructed in Python, M5 instantiates the corresponding C++ objects, which provide good run-time performance for detailed modeling.

4.1 Scripting M5

The M5 executable is invoked as follows:

```
m5 [m5 options] [Python script] [script options]
```

The script, written in standard Python, controls the progress of the simulation. Figure 1 provides an example. The script's first action is to import the m5 package, which contains the interface for orchestrating the simulation. This step is followed by importing all of the objects. These two statements are all that is necessary to access the m5 framework in a script. The next block of code parses the script's options.

The following block of code defines a Linux based system object along with its parameters. Next,

```
import os, optparse, sys
import m5
from m5.objects import *

# handle script options
parser = optparse.OptionParser(option_list=m5.standardOptions)
parser.add_option("-t", "--timing", action="store_true")
(options, args) = parser.parse_args()
if args:
    sys.exit('too many arguments')
parser.parse_args()

# define the system
class System(LinuxSystem):
    if options.timing:
        cpu = DetailedCPU()
    else:
        cpu = SimpleCPU()
    membus = Bus(width=16, clock='400MHz')
    ram = BaseMemory(in_bus=Parent.membus, latency='40ns',
        addr_range=[Parent.physmem.range])
    physmem = PhysicalMemory(range=AddrRange('128MB'))
    tsunami = Tsunami()
    simple_disk = SimpleDisk(disk=Parent.tsunami.disk0.image)
    sim_console = SimConsole(listener=ConsoleListener(port=3456))
    kernel = '/dist/m5/system/binaries/vmlinux-latest'
    pal = '/dist/m5/system/binaries/ts_osfpal'
    console = '/dist/m5/system/binaries/console_ts'
    boot_osflags = 'root=/dev/hda1 console=ttyS0'

# build a network
root = Root()
root.client = System(readfile='/dist/m5/system/boot/netperf-stream-client.rcS')
root.server = System(readfile='/dist/m5/system/boot/netperf-server.rcS')
root.etherlink = EtherLink(int1=Parent.server.tsunami.etherint[0],
                           int2=Parent.client.tsunami.etherint[0])

# get the simulator going
m5.instantiate(root) # instantiate structure
exit_code = m5.simulate() # simulate until termination

print 'Exiting @ cycle', m5.curTick(), 'because', exit_code.getCause()
```

Figure 1: A sample configuration script

multiple systems are connected with an Ethernet link at the root of the hierarchy. The final code block instantiates the simulation hierarchy and tells M5 to begin simulating. Other methods provided by the m5 module cause the simulator to create a checkpoint or to swap a simple CPU for a detailed CPU model for warmup or sampling.

5 Object Models

M5 includes a variety of object models implemented on top of the core simulation engine. These models include CPUs, caches, busses, and I/O devices—everything necessary for modeling networks of complete systems. These models also define the interfaces used for inter-object communication, e.g., transferring requests between objects in the memory hierarchy. Researchers who find the supplied models inadequate can develop new objects that implement the same interfaces and incorporate them transparently into existing configurations.

5.1 CPU Models

M5 contains two primary CPU models, SimpleCPU and O3CPU. Both models derive from a base CPU class and export the same interface, allowing them to be used interchangeably. M5 can also switch between CPU models during run-time, allowing the use of SimpleCPU for fast-forwarding and warm-up phases and O3CPU for taking statistics.

SimpleCPU is an in-order, non-pipelined functional model that can be configured to execute one or more instructions per cycle, but can only have one outstanding memory operation. In addition to fast forwarding and warm-up, SimpleCPU is useful for modeling network client systems whose only goal is to generate traffic for a detailed server system under test.

O3CPU is an out-of-order, superscalar, pipelined, simultaneous multithreading (SMT) model. The O3CPU model simulates an out-of-order pipeline in detail. Individual stages such as fetch, decode, etc. are configurable in their width and latency. Both forward and backward communication between stages is modeled explicitly using time buffers (described below). The model includes detailed branch predictors, instruction queues, load/store queues, functional units, and memory dependence predictors. The O3CPU uses C++ templates to allow for easy switching of individual pipeline stages or structures, avoiding the performance overhead of virtual functions.

The O3CPU model has been developed with a strong focus on ensuring timing accuracy. To promote accuracy, we integrated both timing and functional modeling into a single "execute-in-execute" pipeline, in which functional instruction execution occurs in the execute stage of the timing pipeline. Most other models, such as our old detailed CPU model, are "execute-in-fetch" models, which functionally execute instructions as they are fetched, and then do the timing modeling afterwards. Our execute-in-execute design provides accurate modeling of timing-dependent instructions, such as synchronization operations and I/O device accesses. An execute-in-execute model also provides higher confidence in the realism of the timing model, as unrealistic timing is often manifested as incorrect functional operation as well.

Because M5 supports booting entire operating systems as well as running application binaries with system call emulation, all the CPU models support the full privileged instruction set, virtual address translation, and asynchronous interrupts. M5 currently boots unmodified versions of Linux 2.4/2.6, FreeBSD, HP/Compaq Tru64 Unix, and the L4Ka::Pistachio microkernel.

Both CPU models derive their ISA semantics from a single ISA description written in a powerful custom language. The language allows users to describe instruction behavior using simple statements such as `Rc = Ra + Rb`. A parser converts the ISA description to a set of instruction classes, which include methods that implement the execution semantics, indicate the number and identity of source and destination registers, the type of functional unit required, etc. The object interface is ISA independent, allowing CPU models to be implemented in a largely ISA-agnostic fashion. This approach minimizes the amount of work required to describe new ISAs or enhancements to existing ISAs. Although the initial M5 release implemented only Alpha, support for SPARC and MIPS ISAs is incorporated in the most recent version.

5.2 The O3CPU Pipeline and timebuffers

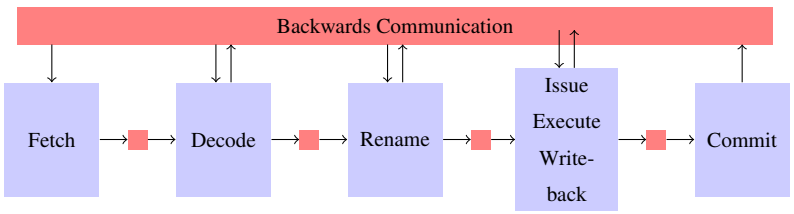

Figure 2: O3CPU Pipeline

The O3CPU model uses a data structure we call a "timebuffer" to model both communication delays be-

tween pipeline stages and processing delays of the stages themselves, as illustrated in Figure 2. A timebuffer is a fixed-size structure that is similar to a queue, where each entry represents a cycle of communication between two structures. Producers generally write into a timebuffer using index 0, and all other consumers index into the buffer using a cycle number relative to their distance from the producer. Timebuffers are usually advanced every simulator cycle at which point all indices are moved to one cycle earlier, and entries are cleared if they exceed the buffer's assigned cycle depth. For example, if there is a timebuffer between fetch and decode, and there is a 2 cycle delay between fetch and decode, fetch would write information into the buffer using index 0, and decode would read information from the timebuffer using index 2. The timebuffer would have to be advanced two times before information written by fetch could be read by decode, thus modeling the 2 cycle delay.

The O3CPU model uses timebuffers to provide both forward and backward communication between pipeline stages. Timebuffers used for forward communication generally include the instructions being sent between the two stages. Additionally a timebuffer is used to provide backward communication for information such as the squash signal or the number of free ROB entries. By limiting interaction between pipeline stages to go through these timebuffers, we avoid unrealistic instantaneous communication between pipeline stages.

5.3 Memory System

The memory system within M5 is divided into two main types of objects: devices and interconnects. Devices are components such as caches, memories, and I/O devices. Interconnects encapsulate communication mechanisms such as busses and networks. Although M5 currently provides only a bus model, the device-to-interconnect interface has been designed to easily allow extension to point-to-point networks as well.

M5 supports configurable caches with parameters for size, latency, associativity, replacement policy, coherence protocol, etc. Additionally, several variants of hardware prefetchers, ranging from simple next line prefetching to more complex history-based approaches, are available as plug-in additions to any of the cache models.

Bus objects model a split-transaction bus which is configurable in both latency and bandwidth. A simple bus bridge object is available to connect busses of different speeds, e.g. the PCI bus and the system bus.

Figure 3 is a diagram of a possible M5 memory hierarchy. The enlarged portion shows the modular port interface, supported by all memory objects, that allows arbitrary interconnections based on the runtime

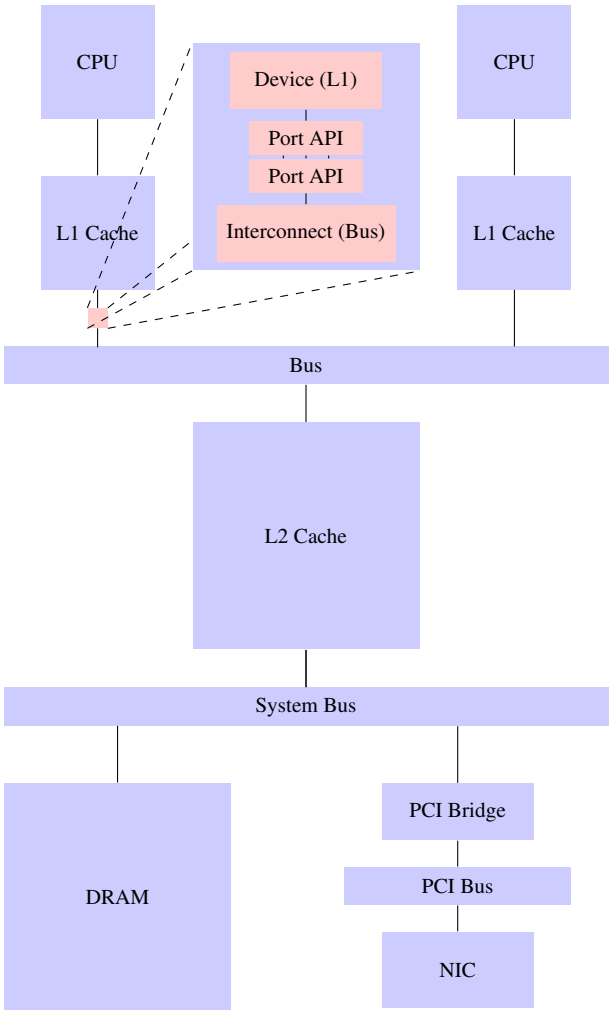

Figure 3: A sample system hierarchy

configuration. The port model allows two device objects to connect without an intervening bus, e.g., to connect a SimpleCPU directly to a memory object for fast functional simulation.

5.4 I/O Devices

I/O devices are first-class participants in the memory hierarchy, responding to programmed I/O accesses at configurable address ranges and issuing DMA transactions. These I/O memory accesses are indistinguishable from CPU accesses in the memory hierarchy, and their timing is modeled with equal fidelity.

Because of our focus on networking workloads, we paid particular attention to the accuracy of our network components. We chose to model a National Semiconductor DP83820 network interface controller (NIC), as it is the only commercial gigabit Ethernet interface for which we could find public documentation. The model is complete with configuration and status registers which can be accessed by the device driver through programmed I/O, as well as a DMA engine to transfer packets to/from main memory. This NIC is modeled with enough fidelity to support the unmodified Linux device driver.

In addition to other regular I/O devices, such as serial ports and disk controllers, M5 also models system chipsets (e.g., memory and interrupt controllers) with sufficient fidelity to boot unmodified OS kernels.

5.5 Ethernet

The NICs of individual systems are connected using an Ethernet link object, which we model as a lossless full-duplex channel of configurable bandwidth and delay. The time on the link is calculated by dividing the packet size by the link bandwidth. If the result is less than the wire delay, only a single packet is allowed on the link at a time. If the delay is large, then it is possible for the link to have multiple packets in flight. If insufficient buffer space is available at the receiver when a packet exits the link, the packet is dropped.

6 Pitfalls of Simulating Networking Workloads

When we began simulating TCP/IP workloads with M5, we soon discovered that the self-tuning nature of the TCP protocol can interact with conventional architecture simulation techniques in unexpected ways. We provide a brief description of this phenomenon here; further details can be found elsewhere [HSBR05].

The TCP protocol uses a feedback mechanism to tune the sender's transmission rate such that data is sent as quickly as possible without overwhelming the network or the receiver. This feedback comes in two forms:

the receiver's advertised buffer space, transmitted in acknowledgment packets, and packet drops, assumed to be due to network congestion, which are detected only indirectly when the receiver fails to properly acknowledge a sent packet. Both of these sources of feedback operate on time scales that are typically on the order of the network round-trip latency.

Common simulation speedup techniques, such as fast forwarding and sampling, can exhibit unexpected interactions with TCP's self tuning. These techniques use simple functional processor and memory models to advance machine state quickly to a point of interest in the workload, at which time more detailed models are used. Because these simple functional models ignore performance-sapping events such as pipeline stalls and cache misses, they typically result in very high effective performance. The switchover from these models to the detailed timing models is equivalent to swapping out a high-performance machine for a much slower system, forcing TCP to retune the transmission rate to this new bottleneck. Simulation statistics collected before this retuning process is complete will not reflect the system's actual steady-state behavior. In some situations, the amount of warm-up time required for TCP to stabilize can be significantly larger than is practical to simulate.

As one example, consider a two-system configuration where the receiver switches from a simple functional CPU model to a detailed CPU model. A possible packet flow is shown in the diagram in Figure 4. The initial configuration will have achieved some steady state, shown at the top of the diagram. The CPU model switchover increases the receiver's processing time, delaying its acknowledgments. When the server fails to receive an acknowledgment within some multiple of its expected round-trip time (which it estimated based on previous observations), it will time out and assume a packet was dropped. TCP then reacts to this supposed packet loss by restricting the number of unacknowleged packets the sender is allowed to have outstanding. Eventually this limit can go far below the number of packets the sender has already transmitted, meaning that the sender cannot transmit any packets until the packet queue at the receiver has drained. When studying high-bandwidth networks, these queues can be very large. As a result, periods of near-zero bandwidth may be observed after a switch from functional to detailed models.

To obtain meaningful bandwidth results, the waiting (warm-up) period between switching to a more detailed model and collecting statistics must be extended to allow TCP retuning as well. This latency is primarily a function of the network round-trip time. Figure 5 shows the difference in tuning delay when the link latency between two systems is changed from zero to 400μs. The graph begins at the point where a purely functional system model is in one of the systems is replaced by a functional CPU driving a realistic

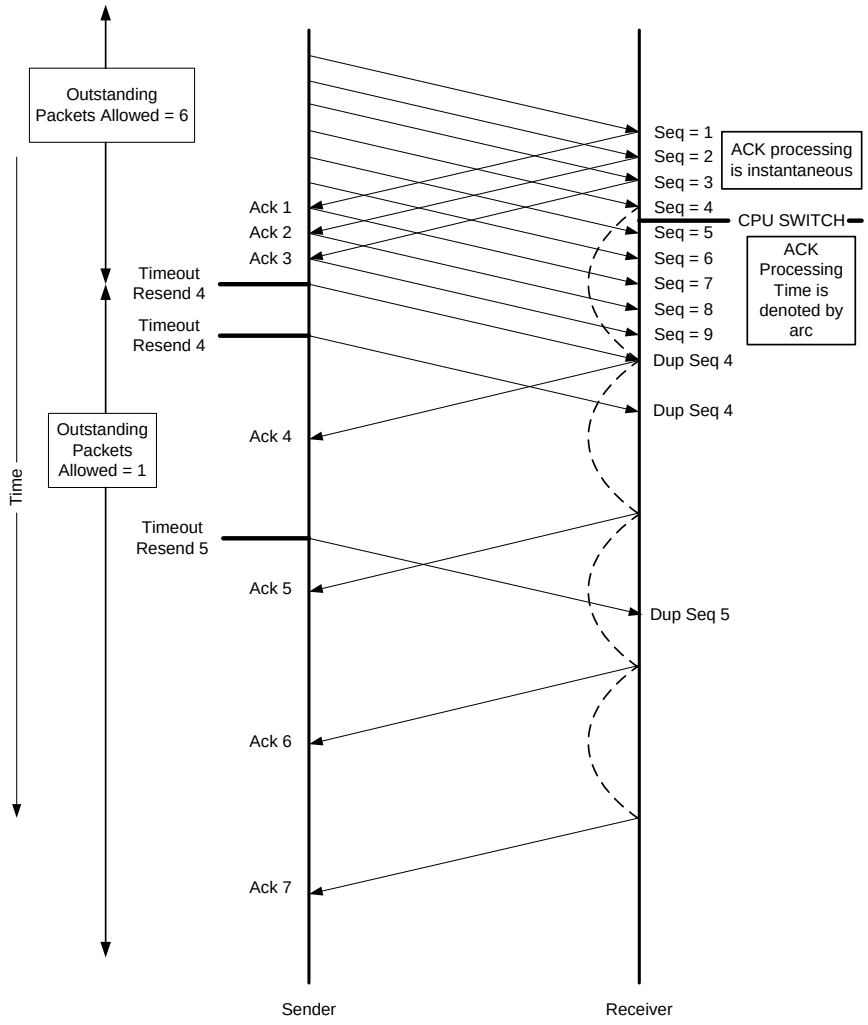

Figure 4: TCP flow disruption due to CPU model change. The arrows between the vertical lines represent packet transmissions.

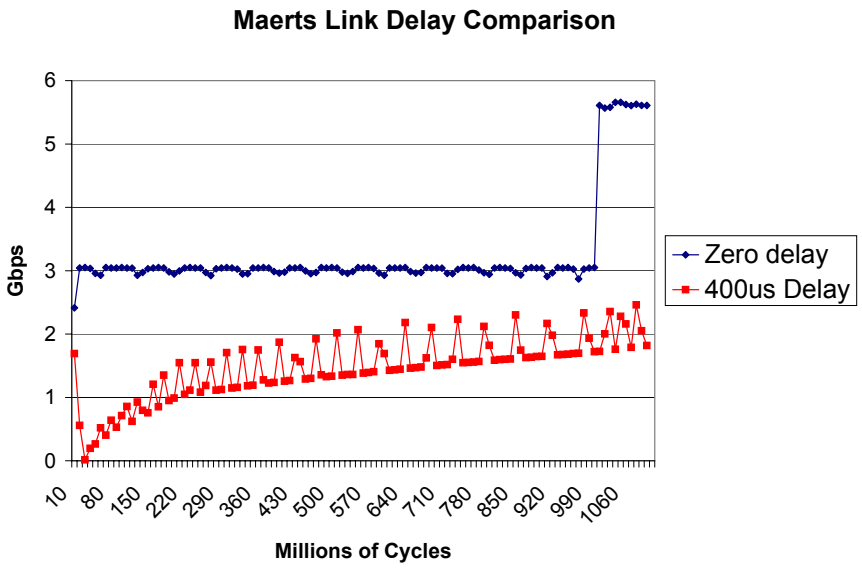

Figure 5: A Comparison of TCP tuning times with respect to round trip times

cache hierarchy, a standard configuration for warming up the cache contents. The zero-latency link quickly reaches steady state, but the long-latency link drops briefly to zero bandwidth before entering a gradual ramp-up phase. After 1 billion cycles, the functional CPU is replaced by a detailed timing model. Because the detailed timing model supports out-of-order execution and multiple outstanding cache misses, it has higher effective performance than the simple model, and the zero-latency link quickly adjusts to reflect that. However, the long-latency link simulation still has not recovered from the original transition, even after several hours of simulation time.

In our experiments to date, we have been careful to allow TCP retuning time after model changes, and to minimize this time by using low-latency links and by matching the effective performance of our fast functional, warm-up, and detailed timing configurations as much as possible.

7 Validating M5

To provide meaningful results, execution-driven architectural simulators must both be functionally correct and model performance accurately. Functional correctness, though often challenging, is typically straightforward to test. In many cases, a lack of functional correctness has catastrophic consequences that cannot be ignored. Performance accuracy, however, is much harder to verify and much easier to neglect. As a result,

it generally gets short shrift from the research community. This situation is ironic: given that the primary output of these simulators is performance data rather than simulated program output, performance accuracy is at least as important as functional correctness, if not more so.

To validate M5, we ran several benchmarks, including receive and transmit microbenchmarks and a web server benchmark, and compared the performance agains our real reference machines—two Compaq Alpha XP1000 systems running at 500MHz and 667MHz. We do not strive to model any particular system precisely, but merely tuned our generic component models to approximate the XP1000.

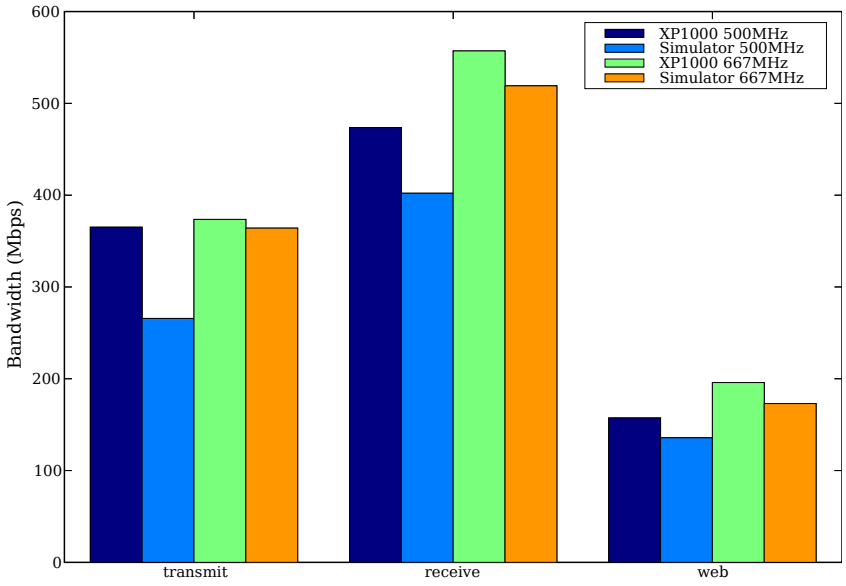

Figure 6: Absolute bandwidth in Mbps

The accuracy of M5 with respect to network bandwidth is depicted in Figure 6, which shows absolute achieved network bandwidth. Since nearly all data packets were of maximal size, bandwidth correlates directly with the packet rate as well. The difference between the simulated and actual systems is modest, deviating by no more than 20Mbps. This corresponds to a relative performance difference of less than 4% for the transmit and webserver benchmarks and 11.5% for the receive benchmark. We believe that the discrepancy in the receive benchmark is due to our inability to understand or model unexpected TLB performance issues on the 500MHz XP1000 system that are not present in the 667MHz system, possibly due to the different CPU stepping involved.

Among the performance problems we uncovered and remedied in the validation process involved TLB

fault handling. Our original execute-in-fetch out-of-order CPU model, which was used for this validation, did not drain the pipeline on TLB misses. As a result, the TLB miss instruction, the TLB miss handler, and application code from after the handler could reside in the pipeline simultaneously. Because there are no explicit register dependencies between these code blocks, the dynamic instruction scheduler could issue instructions from these different regions in parallel. The removal of the dependencies between instructions before and after the TLB miss meant that taking a TLB miss could in some cases lead to higher simulated performance! This discovery confirmed to us not only the importance of validation but also the advantages of an execute-in-execute CPU timing model. Under that approach, this bug could not have remained hidden, as it would have resulted in a functionally incorrect model as well.

Considering the complexity of the system and our use of generic, configurable components, we consider these validation results quite good. Validation of other pieces of the simulator and additional details about the methodology, benchmarks, and related work are available in a previous paper [SBHR05].

8 Conclusion

Though intended for simulating networked systems, M5's structured design, rich feature set, and library of predefined object models provide a powerful framework for building complex architectural simulations in a variety of domains. To foster its usage, and to reduce the duplication of effort in the research community, we have sought to make M5 readily accessible to others, including releasing all our code under liberal open-source licensing terms. As a result, M5 has already attracted a significant user community from around the world.

We are working to provide additional online resources that allow users to contribute back to M5, including wiki-based documentation and a public source repository. In the long term, a widely shared simulation infrastructure to which researchers contribute back their enhancements and additions would allow innovation to proceed at a quicker pace by improving the repeatability and comparability of experiments, lowering barriers to collaboration, and enabling architects to build on the research of others. As the community grows, the available modules will grow, allowing architects to focus on tough higher level problems—the necessary system and software changes required by tomorrow's chip-multiprocessor and high-bandwidth I/O infrastructures.

Acknowledgements

Korey Sewell and Gabe Black are new contributors to M5, developing the MIPS and SPARC ISAs respectively. Andrew Schultz, Miguel Vega, and Ben Nash contributed to the Tsunami platform model, necessary for Linux and FreeBSD. Steve Raasch and Erik Hallnor contributed significantly to the original CPU and memory models respectively.

M5 has been developed with generous support from several sources, including the National Science Foundation under grants CCR-0105503 and CCR-0219640, Hewlett-Packard, Intel, and IBM. Continuing development is being supported by Sun Microsystems and MIPS Technologies. Individuals working on M5 have also been supported by an Intel Fellowship (Nate Binkert), a Lucent Fellowship (Lisa Hsu), a Sloan Research Fellowship (Steve Reinhardt), and a University of Michigan Fellowship (Korey Sewell).

References

[BAB96] Doug Burger, Todd M. Austin, and Steve Bennett. Evaluating future microprocessors: the SimpleScalar tool set. Technical Report 1308, Computer Sciences Department, University of Wisconsin–Madison, July 1996.

[EAB+02] Joel Emer, Pritpal Ahuja, Eric Borch, Artur Klauser, Chi-Keung Luk, Srilatha Manne, Shubhendu S. Mukherjee, Harish Patil, Steven Wallace, Nathan Binkert, Roger Espasa, and Toni Juan. Asim: A performance model framework. *IEEE Computer*, 35(2):68–76, February 2002.

[HSBR05] Lisa R. Hsu, Ali G. Saidi, Nathan L. Binkert, and Steven K. Reinhardt. Sampling and stability in TCP/IP workloads. In *Proc. First Annual Workshop on Modeling, Benchmarking, and Simulation*, pages 68–77, June 2005.

[HSW+04] Nikolaos Hardavellas, Stephen Somogyi, Thomas F. Wenisch, Roland E. Wunderlich, Shelley Chen, Jangwoo Kim, Babak Falsafi, James C. Hoe, and Andreas G. Nowatzyk. SIMFLEX: A fast, accurate, flexible full-system simulation framework for performance evaluation of server architecture. *ACM SIGMETRICS Performance Evaluation Review*, 31(4):31–35, March 2004.

[MCE+02] Peter S. Magnusson, Magnus Christensson, Jesper Eskilson, Daniel Forsgren, Gustav Halberg, Johan Hogberg, Fredrik Larsson, Adreas Moestedt, , and Bengt Werner. Simics: A full system simulation platform. *IEEE Computer*, 35(2):50–58, February 2002.

[MSB+05] Milo M. K. Martin, Daniel J. Sorin, Bradford M. Beckmann, Michael R. Marty, Min Xu, Alaa R. Alameldeen, Kevin E. Moore, Mark D. Hill, and David A. Wood. Multifacet's general execution-driven multiprocessor simulator (gems) toolset. *SIGARCH Comput. Archit. News*, 33(4):92–99, 2005.

[RHWG95] Mendel Rosenblum, Stephen A. Herrod, Emmett Witchel, and Anoop Gupta. Complete computer system simulation: The SimOS approach. *IEEE Parallel & Distributed Technology*, 3(4):34–43, Winter 1995.

[SBHR05] Ali G. Saidi, Nathan L. Binkert, Lisa R. Hsu, and Steven K. Reinhardt. Performance validation of network-intensive workloads on a full-system simulator. In *Proc. 2005 Workshop on Interaction between Operating System and Computer Architecture (IOSCA)*, October 2005.

Bios

Nathan L. Binkert is a senior software engineer at Arbor Networks Inc. and principal developer of M5. He received a BSE in electrical engineering and an MSE and a PhD in computer science and engineering all from the University of Michigan. As an intern at Compaq VSSAD, he was involved in the initial development of the ASIM simulator, currently in use at Intel.

Ronald G. Dreslinski is a PhD candidate in the EECS Department at the University of Michigan, and the main developer of M5's memory system. He received a BSE in electrical engineering, a BSE in computer engineering, and an MSE in computer science and engineering all from the University of Michigan.

Lisa R. Hsu is a PhD candidate in the EECS Department at the University of Michigan, and was the developer of M5's Ethernet network interface model. She received a BSE in electrical engineering from Princeton University and an MSE in computer science and engineering from the University of Michigan.

Kevin T. Lim is a PhD pre-candidate in the EECS Department at the University of Michigan, and the developer of M5's detailed CPU model. He received a BSE in computer engineering and an MSE in computer science and engineering, both from the University of Michigan.

Ali G. Saidi is a PhD candidate in the EECS Department at the University of Michigan, and wrote much of the platform code for Linux full-system simulation of Alpha and SPARC. He received a BS in electrical engineering from the University of Texas at Austin and an MSE in computer science and engineering from the University of Michigan.

Steven K. Reinhardt is an associate professor in the EECS Department at the University of Michigan, a managing engineer at Reservoir Labs, Inc., and a principal developer of M5. He received a BS from Case Western Reserve University and an MS from Stanford University, both in electrical engineering, and a PhD in computer science from the University of Wisconsin-Madison. While at Wisconsin, he was the principal developer of the Wisconsin Wind Tunnel parallel architecture simulator.