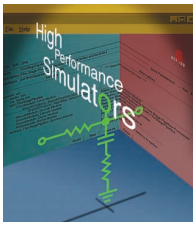
SimpleScalar: An Infrastructure for Computer System Modeling



Developed to provide an infrastructure for simulation and architectural modeling, the SimpleScalar toolset offers an open source distribution especially suited to the needs of researchers and instructors.

Todd Austin
Eric Larson
Dan Ernst
University of Michigan

To accelerate hardware development, designers often employ software models of the hardware they build. They implement these models in traditional programming languages or hardware description languages, then exercise them with the appropriate workload. Designers can execute programs on these models to validate the performance and correctness of a proposed hardware design.

Programmers can use software models to develop and test software before the real hardware becomes available. Although software models are slower than their hardware counterparts, programmers can build and test them in minutes or hours rather than in the months needed to build real hardware. This fast mechanism for design and test provides shorter time to market and much higher quality first silicon.

Three critical requirements drive the implementation of a software model: performance, flexibility, and detail. *Performance* determines the amount of workload the model can exercise given the machine resources available for simulation. *Flexibility* indicates how well the model is structured to simplify modification, permitting design variants or even completely different designs to be modeled with ease. *Detail* defines the level of abstraction used to implement the model's components. A highly detailed model will faithfully simulate all aspects of machine operation, whether or not a particular aspect is important to any metric being measured.

In practice, optimizing all three model characteristics in tandem is difficult. Thus, most model implementations optimize only one or two of them, which explains why so many software models exist, even for a single product design. Research models tend to optimize performance and flexibility at the expense of detail.

The SimpleScalar toolset provides an infrastructure for simulation and architectural modeling. The toolset can model a variety of platforms ranging from simple unpipelined processors to detailed dynamically scheduled microarchitectures with multiple-level memory hierarchies. For users with more individual needs, SimpleScalar offers a documented and well-structured design, which simplifies extending the toolset to accomplish most architectural modeling tasks.

SimpleScalar simulators reproduce computing device operations by executing all program instructions using an interpreter. The toolset's instruction interpreters support several popular instruction sets, including Alpha, Power PC, x86, and ARM.

MODELING BASICS

The typical approach to computer system modeling leverages a simple approximate model with good simulation performance and a modular code structure. This simulator style suits researchers and instructors well because the simple model focuses on the design's primary components, leaving out

0018-9162/02/$17.00 © 2002 IEEE

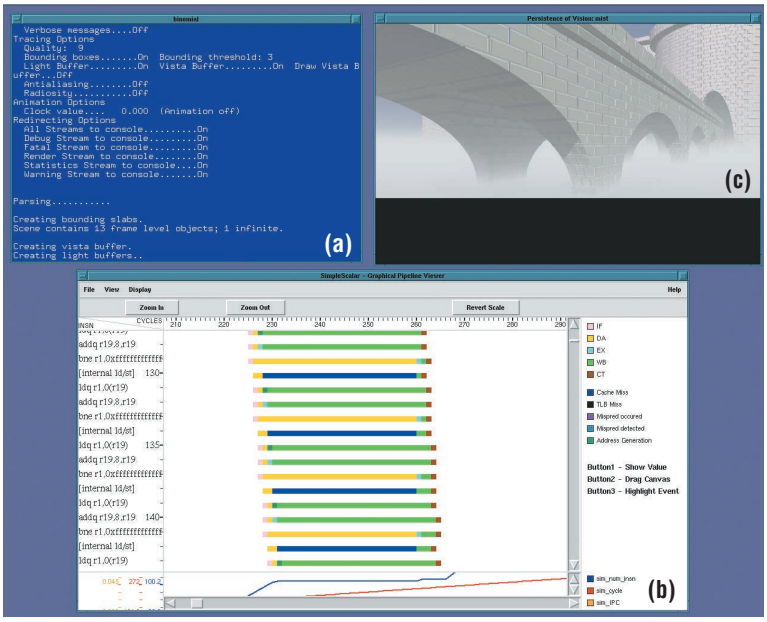


Figure 1. Sample SimpleScalar user session. (a) The console window displays simulator-generated messages, (b) the graphical pipeline viewer provides a representation of the simulated program's execution, and (c) the simulated graphical display shows program output.

the minutiae of detail that might otherwise hinder the model's performance and flexibility. Industrial users, on the other hand, require very detailed models to minimize design risk. Detailed modeling assures that a design has no faulty components or acute performance bottlenecks.

The additional detail necessary to implement these models usually comes at the expense of model performance. In industrial applications, individual model performance often takes a backseat because companies have the resources available to stock large simulation server pools. Hardware simulation tends to be a throughput-bound task: Designers want to simulate multiple design configurations running several benchmarks. Adding more machines to the simulation pool decreases the overall runtime to the maximum runtime of any single experiment. For example, Intel's Pentium 4 design team used a simulation pool that contained more than 1,000 workstations.[1]

In some cases, designers optimize a model for performance and detail at the expense of flexibility. Designers typically employ these models when they need to faithfully represent a device at speeds capable of executing large workloads, but don't need to change the model.

Software performance analysis is an example of this type of application. Software developers need accurate depictions of program performance, but rarely require changes to the model because they are only concerned with program performance for a particular processor.

The FastSIM simulator[2] microarchitecture uses memoization to record internal simulator states and the actions taken—such as statistical variable update—upon entering those states. This permits microarchitectural models of arbitrarily high detail to quickly process instructions. However, the implementation sacrifices significant flexibility because the approach requires all microarchitecture components to provide internal-state hashing mechanisms and recording of per-cycle actions.

MODELING WITH SIMPLESCALAR

SimpleScalar was written in 1992 as part of the Multiscalar project at the University of Wisconsin, under Gurindar Sohi's direction. In 1995, with Doug Burger's assistance, the toolset was released as an open source distribution freely available to academic noncommercial users. SimpleScalar LLC now maintains the tool, which is distributed through SimpleScalar's Web site at http://www.simplescalar.com.

Since its release, SimpleScalar has become popular with researchers and instructors in the computer architecture research community. For example, in 2000 more than one-third of all papers published in top computer architecture conferences used the SimpleScalar tools to evaluate their designs. SimpleScalar provides an infrastructure for computer system modeling that simplifies implementing hardware models capable of simulating complete applications. During simulation, model instrumentation measures the dynamic characteristics of the hardware model and the performance of the software running on it.

Figure 1 shows a typical SimpleScalar user session, with the persistence-of-vision raytracer (POV-ray) graphics application running on a detailed Alpha processor microarchitecture model. The simulated graphical display in the screen's upper-left corner shows the program I/O. The simulator console window in the screen's upper-right corner displays simulator-generated messages plus *stdout*/*stderr*, the simulated program's standard output. At the bottom of the screen, the graphical pipeline view window provides a graphical representation of the simulated program's execution on the detailed microarchitecture model.

GPV shows the execution of instructions from fetch until retirement, displaying each instruction's state throughout the pipeline. In the example, the blue lines in the display represent long-latency I-cache misses. GPV forms part of SimpleScalar's visualization infrastructure and provides a useful tool for identifying hardware and software bottlenecks.[3]

SimpleScalar includes several sample models suitable for a variety of common architectural analysis tasks. Table 1 lists the simulator models included with SimpleScalar version 3.0. The simulators range from *sim-safe*, a minimal SimpleScalar simulator that emulates only the instruction set, to *sim-outorder*, a detailed microarchitectural model with

dynamic scheduling, aggressive speculative execution, and a multilevel memory system.

All the simulators have fairly small code sizes because they leverage SimpleScalar's infrastructure components, which provide a broad collection of routines to implement many common modeling tasks. These tasks include instruction-set simulation, I/O emulation, discrete-event management, and modeling of common microarchitectural components such as branch predictors, instruction queues, and caches. In general, the more detailed a model becomes, the larger its code size and the slower it runs due to increased processing for each instruction simulated.

Figure 2 shows the SimpleScalar hardware model's software architecture. Applications run on the model using a technique called *execution-driven simulation*, which requires the inclusion of an instruction-set emulator and an I/O emulation module. The instruction-set emulator interprets each instruction, directing the hardware model's activities through callback interfaces the interpreter provides.

SimpleScalar includes instruction interpreters for the ARM, x86, PPC, and Alpha instruction sets. The interpreters are written in a target definition language that provides a comprehensive mechanism for describing how instructions modify registers and memory state. A preprocessor uses these machine definitions to synthesize the interpreters, dependence analyzers, and microcode generators that SimpleScalar models need. With a small amount of extra effort, models can support multiple target instruction sets by implementing the full range of callback interfaces the target definition language defines.

The I/O emulation module provides simulated programs with access to external input and output facilities. SimpleScalar supports several I/O emulation modules, ranging from system-call emulation to full-system simulation. For system-call emulation, the system invokes the I/O module whenever a program attempts to execute a system call in the instruction set interpreter, such as a `callpal syscall` instruction in the Alpha instruction set. The system emulates the call by translating it to an equivalent host operating-system call and directing the simulator to execute the call on the simulated program's behalf. For example, if the simulated program attempts to open a file, the I/O emulation module translates the request to a call to `open()` and returns the resulting file descriptor or error number in the simulated program's registers.

Other I/O targets provide finer-grain emulation of actual hardware devices. For example, the SimpleScalar/ARM release includes an I/O emulator for Compaq IPaq hardware devices. This emulator is detailed enough to boot the ARM Linux operating system. Device-level I/O emulation has the added advantage of analyzing the operating system portion of an application's execution. This additional fidelity proves especially useful with server applications, where networking and file system services demand much of the workload's runtime.

At the center of each model, the simulator core code defines the hardware model organization and instrumentation. Figure 3 lists the code for a hardware timing model of a simple microarchitecture in which all instructions execute in a single cycle except for loads and stores. These instructions execute in two cycles if they hit in the data cache, or in 10 cycles if they miss.

The simulator core defines the simulator's main loop, which executes one iteration for each instruction of the program until finished. For a timing model, the main loop must account for the progression of execution time, measured in clock cycles for this model. The `cycle` variable stores the execution time, which counts the total number of clock cycles required to execute the program up to the current instruction. To determine the relative performance of programs, the model compares the

Table 1. SimpleScalar baseline simulator models.

Simulator	Description	Lines of code	Simulation speed
sim-safe	Simple functional simulator	320	6 MIPS
sim-fast	Speed-optimized functional simulator	780	7 MIPS
sim-profile	Dynamic program analyzer	1,300	4 MIPS
sim-bpred	Branch predictor simulator	1,200	5 MIPS
sim-cache	Multilevel cache memory simulator	1,400	4 MIPS
sim-fuzz	Random instruction generator and tester	2,300	2 MIPS
sim-outorder	Detailed microarchitectural timing model	3,900	0.3 MIPS

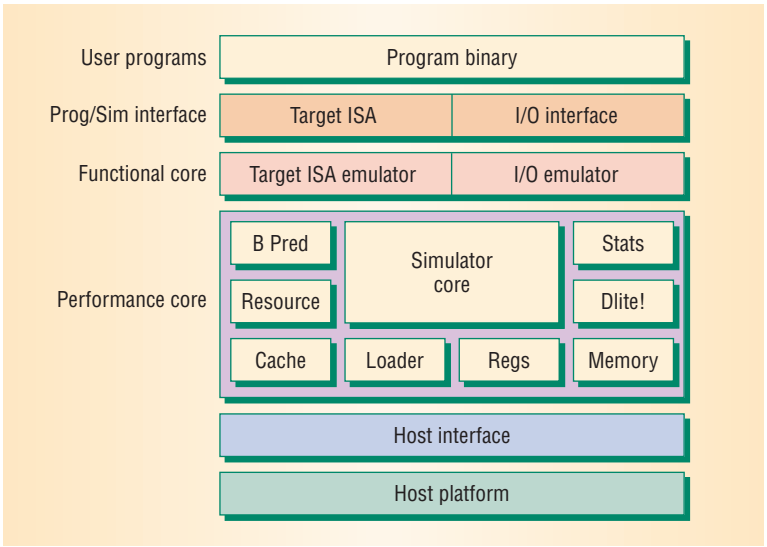


Figure 2. SimpleScalar simulator software architecture. Applications run on the simulator using execution-driven simulation, which requires the inclusion of an instruction set emulator and an I/O emulation module.

SIMULATOR CORE

```
counter_t insn;
counter_t cycle;

sim_main()
{
  stat_register("insn", &insn, "total instructions");
  stat_register("cycle", &cycle, "total cycles");
  stat_formula("IPC", "insn/cycle", "inst per cycle");

while (!sim_done)
  {
    inst = sim_execute_insn();
      insn++;
      cycle++;
      if (inst.flags & F_MEMOP)
        cycle += cache_access(inst.addr)
    }
}
```

CACHE COMPONENT

```
time_t cache_access(addr_t addr)
{
  word_t index = cache_hash(addr)
  if (tag[index] == addr)
    { /* hit */
      cache_update_lru(index);
        return 1;
      }
    else
      { /* miss */
        cache_handle_miss(addr);
      }  return 9;
}
```

Figure 3. Sample code for a hardware timing model of a simple microarchitecture in which all instructions execute in a single cycle, except for loads and stores.

total number of cycles to complete their execution. The simulation model increments `cycle` once for each instruction, once again for loads and stores, and 10 more times for any access that missed in the data cache.

The cache.c module supplied with the SimpleScalar distribution implements the data cache. Figure 3 shows the relevant portion of the implementation. The cache module uses a hash table to record the cache blocks it contains. If an access address matches an entry in the hash table, the access returns the hit latency. If an access address does not match an entry in the hash table, the system calls the cache miss handler, which returns the number of clock cycles required to service the cache miss. The model builder specifies the miss handler, which may be another cache module or a DRAM memory model. The cache module does not return the value accessed in the cache because this value has no effect on cache access latency. For designs in which the cache contents affect access latency, such as compressed cache designs, the system can configure the cache module to store and return the value accessed.

In addition to standard component models, SimpleScalar provides a variety of helper modules that implement useful facilities that many models require. These modules include a debugger, program loader, symbol table reader, command line processor, and statistical package.

The sample code in Figure 3 uses the statistical package to manage model instrumentation. The `stat_register()` interface registers the simulator instrumentation variables *insn* and *cycle* with the statistical module. Once registered, the statistical package tracks updates to statistical counters, producing on request a detailed report of all model instrumentation. The `stat_formula()` interface allows derived instrumentation to be declared, creating a metric that is a function of other counters. In Figure 3, instruction per cycle (IPC) denotes a derived statistic equal to the number of instructions executed divided by the total execution cycles.

Simulators interface to the host machine via the host interface module, a thin layer of code that provides a canonical interface to all host machine data types and operating system interfaces. Simulators that use host interface types and operating system services can be easily run on new hosts by simply porting the host interface module. In Figure 3's sample code, `counter_t` and `word_t` are canonical types, exported by the host interface, that define the maximum-sized unsigned integer and 32-bit signed integers, respectively.

EXECUTION-DRIVEN SIMULATION

All SimpleScalar models use an execution-driven simulation technique that reproduces a device's internal operation. For computer system models, this process requires reproducing the execution of instructions on the simulated machine. A popular alternative, trace-driven simulation, employs a stream of prerecorded instructions to drive a hardware timing model. This method uses a variety of hardware- and software-based techniques—such as hardware monitoring, binary instrumentation, or trace synthesis—to collect instruction traces.

Advantages

Execution-driven simulation provides many powerful advantages compared with trace-based techniques. Foremost, the approach provides access to all data produced and consumed during program execution. These values are crucial to the study of optimizations such as value prediction, compressed-memory systems, and dynamic power analysis.

In dynamic power analysis, the simulation must monitor the data values sent to all microarchitectural components such as arithmetic logic units and caches to gauge dynamic power requirements. The hamming distance of consecutive data inputs defines the degree to which input bits change, which in turn causes transistor switching that consumes dynamic power.

Execution-driven simulation also permits greater accuracy in the modeling of speculation, an aggressive performance optimization that runs instructions before they are known to be required by predicting vital program values such as branch directions or load addresses. Speculative execution proceeds at a higher throughput until the simulation finds an incorrect prediction, which flushes the processor pipeline and restarts it with correct program values. Before this recovery occurs, however, misspeculated code will compete for resources with nonspeculative execution, potentially slowing the program. Trace-driven techniques cannot model misspeculated code execution because instruction traces record only correct program execution. Execution-driven approaches, on the other hand, can faithfully reproduce the speculative computation and correctly model its impact on program performance.

Drawbacks

Execution-driven simulation has two potential drawbacks: increased model complexity and inherent difficulties in reproducing experiments. Model complexity increases because execution-driven models require instruction and I/O emulators to produce program computation, while trace-driven simulators do not. SimpleScalar mitigates this additional complexity through the use of a target definition language and a set of internal tools that synthesize the emulators that SimpleScalar models require. The target definition provides a central mechanism for specifying the complexities of instruction execution, including operational semantics, register and memory side effects, and instruction faulting semantics. The same facility makes it straightforward for SimpleScalar models to support multiple instruction sets.

Because execution-driven simulators interface directly to input devices through I/O emulation, reproducing experiments that depend on real-world external events may be difficult. For example, changes in response to network latency and the contents of incoming requests affect a Web server application running on an execution-driven model. Trace-driven experiments do not experience these difficulties because instruction traces record the effects of external inputs within the instruction trace file. To overcome this reproducibility problem, SimpleScalar provides an external input tracing feature. Such traces record external device inputs to a program during its live execution. Replaying traced inputs to the simulated program recreates the original execution. Since these traces only contain external inputs, they are small and can be easily shared with other SimpleScalar users.

SYSTEM MODEL INFRASTRUCTURE

Our primary impetus for releasing SimpleScalar stemmed from our desire to reduce the effort required to become a productive researcher in the computer architecture research community. By its very nature a quantitative engineering discipline, computer architecture modeling requires access to tools capable of measuring program characteristics and performance. In the past, no such tools were widely available. Thus, much of the early work in computer architecture could be performed only at large universities and corporations where resources were available to develop the necessary tools. Building computer system modeling tools from scratch requires a significant development effort that consists mostly of writing mundane software components such as loaders, debuggers, and interpreters. Constructing these components requires great effort. Making them reliable is even more difficult, taking time that could be better spent on innovation.

Computer architecture modeling requires access to tools capable of measuring program characteristics and performance.

Although SimpleScalar can be thought of as a simulator collection, we view it as an infrastructure for computer system modeling. The differences between a tool and an infrastructure lie in the care taken in designing the internal modules and interfaces. An infrastructure must organize these components to permit their reuse over a wide range of modeling tasks. Moreover, the module interfaces must be expressive and well documented. Computer architecture researchers can use SimpleScalar's performance-analysis infrastructure to evaluate complex design optimizations by specifying them as changes and comparing their relative impact on baseline models.

Figure 2 shows the baseline modules that comprise SimpleScalar's software architecture. These modules export functions ranging from statistical analysis, event handlers, and command-line processing to implementations of modeling components such as branch predictors, caches, and instruction queues.

Open source and academia

Academic noncommercial users can download and use SimpleScalar tools free of charge. In addition, researchers can use SimpleScalar source code to build new tools and release them to other academic noncommercial users. The only restriction regarding redistribution is that the code must

include the SimpleScalar academic noncommercial use license.

An open source distribution model gives users maximum flexibility in how they can modify and share the infrastructure. If the interfaces exported within the infrastructure prove insufficient for easily implementing a new model, users can extend the sources as required to implement their ideas. If these additions are generally useful, users can choose to distribute these enhancements to others. The Wattch model (http://citeseer.nj.nec.com/brooks00wattch.html), a framework for analyzing and optimizing microprocessor power dissipation, provides an excellent example of this capability. Wattch required significant changes to the baseline models, including an infrastructure to model physical device characteristics such as area and energy. The "Architecture-Level Power Modeling with Wattch" sidebar describes these challenges and their solutions in detail.

An open distribution model has potential drawbacks, however. Once the source code becomes available, the tool is likely to undergo a higher level of inspection than typical for research software. Given such scrutiny, it behooves researchers to make available only their highest quality work.

Architecture-Level Power Modeling with Wattch

Margaret Martonosi and David Brooks, Princeton University
Vivek Tiwari, Intel

Power dissipation and thermal issues have assumed increasing significance in modern-processor design. As a result, making power and performance tradeoffs more visible to chip architects and even compiler writers has become crucial. To support this work, many researchers have begun developing tools to estimate the energy consumption of architectures and system software.

Before the late 1990s, most power analysis tools operated below the architecture or microarchitecture levels and achieved high accuracy by calculating power estimates for designs only after developers completed layout or floorplanning. In contrast, architecture-level power modeling seeks to provide reasonably accurate high-level power estimates at useful simulation speeds much earlier in the design process.

Tracking Data Activity

With these issues in mind, in 1998 we began to develop Wattch, an architecture-level power modeling framework.[1] Wattch performs power analysis by tracking, on a per-cycle basis, the usage and data activity of microarchitectural structures such as the instruction window, caches, and register files. We use the unit-level usage statistics to scale appropriate power models that correspond to these structures. These fully parameterizable power models are based on capacitance estimates of the major internal nodes within these structures.

Wattch can be used for several types of architectural-level studies. Power-performance design tradeoff studies can be performed by simply varying microarchitectural parameters such as issue width, instruction window size, cache size, and so on. Studying the power and performance effect of additions to the microarchitecture can also be explored by modeling performance issues and instantiating power models for the additional hardware structures. Finally, compiler techniques and software energy profiling experiments can be performed.

Choosing an Infrastructure

When choosing a performance estimation infrastructure on which to base Wattch, we could choose to modify one of a handful of existing infrastructures or write our own. In the end, we found the SimpleScalar toolset attractive because of its parameterizable microarchitecture, wide user support, and well-established code base.

On the other hand, using SimpleScalar out of the box for power modeling presented some downsides. For example, the Register Update Unit structure is not representative of most modern microarchitectures. We based Wattch on the original RUU version of SimpleScalar so that its differences from the original code base would be minimized. Fortunately, users can fairly easily modify SimpleScalar or Wattch's microarchitecture to look more appropriate.

Wattch was one of the first attempts to demonstrate that power analysis can be performed at the architectural level with reasonable accuracy and speed. It has also provided our group and others with a useful measurement platform for doing power-aware research studies. Perhaps most importantly, Wattch and its first-generation counterpart tools from Penn State[2] and Intel[3] may serve as first steps toward future power-and-performance estimation tools with even better tradeoffs between accuracy and performance.

References

1. D. Brooks, V. Tiwari, and M. Martonosi, "Wattch: A Framework for Architectural-Level Power Analysis and Optimizations," *Proc. 27th Ann. Int'l Symp. Computer Architecture*, IEEE CS Press, Los Alamitos, Calif., 2000, pp. 83-94.
2. N. Vijaykrishnan et al., "Energy-Driven Integrated Hardware-Software Optimizations Using SimplePower," *Proc. 27th Ann. Int'l Symp. Computer Architecture*, IEEE CS Press, Los Alamitos, Calif., 2000, pp. 95-106.
3. G. Cai and C.H. Lim, "Architectural-Level Power/Performance Optimization and Dynamic Power Estimation," Cool Chips Tutorial in *Proc. MICRO32*, http://huron.cs.ucdavis.edu/Micro32/presentations/cool_chips.pdf, 1999.

Many researchers have studied the internals of the SimpleScalar code, including Doug Burger and colleagues, who compared SimpleScalar's *sim-outorder* model to a validated Alpha 21264 microarchitecture model.[4] Although this study showed that SimpleScalar successfully predicted the performance trends of programs running on the modeled hardware, it also uncovered several inaccuracies in the *sim-outorder* model that required code updates.

An open source distribution makes it more difficult to commercialize the effort later, because having access to the source makes it easier for potential customers to use the technology without a license or to recreate a similar product based on the open source.

Proactive user support

SimpleScalar's user base currently includes approximately 500 researchers and about twice as many students. The support load manifests in two forms: direct support for user questions and code maintenance.

Direct user support creates by far the largest demand. SimpleScalar's primary developers answer about 10 e-mail messages per week, with many more arriving just before popular conference deadlines or code updates. Most questions come from users who experience difficulty installing or running the model.

Experienced users often write to SimpleScalar's developers seeking advice on how to implement a particular new feature. A smaller portion of the support load addresses the continual process of code maintenance. The code must be routinely updated to accommodate host operating system and target compiler updates that affect the SimpleScalar I/O emulators and host interface. In addition, the developers must fix code bugs and model inaccuracies, then promptly disseminate the updates to the user community. Locating the resources to support SimpleScalar thus presents a perennial problem.

To keep the support load as light as possible, we have devised several proactive mechanisms that support SimpleScalar users. Like most other open source projects, we make available detailed documentation, including a user tutorial, hacker's guide, and FAQ database.

We also draw assistance from the large and active user community through SimpleScalar's mailing lists. We encourage users with problems to send e-mail messages to the mailing lists before contacting the developers. As a result, other SimpleScalar users answer nearly half of all support questions. We archive the mailing lists on the SimpleScalar Web site, where users can browse through old messages, often finding answers to their questions.

New SimpleScalar users typically require the most support. We encourage them to attend SimpleScalar tutorials, held occasionally in conjunction with popular computer architecture conferences. Each tutorial provides a detailed overview of how to use and extend SimpleScalar tools.

A self-test simulation compares binary output to reference results.

For users attempting to build their first model, we offer a minimal SimpleScalar model and tutorial. This version demonstrates the absolute minimum code required to build a SimpleScalar model, serves as a starting point for learning about SimpleScalar modeling, and doubles as a useful baseline for simple modeling projects.

Finding and building appropriate benchmarks to drive simulation experiments can present a hurdle for new SimpleScalar users. To assist them, we make available precompiled binaries for the SPEC, MediaBench, and MiBench benchmark suites. Making this resource available takes little additional effort because we use these benchmarks internally for testing.

Users run SimpleScalar tools on a wide variety of machines, so distributing code that builds cleanly and works on a broad selection of platforms significantly reduces support needs. SimpleScalar provides portability through the host interface layer, a collection of types and routines that implements machine-independent host abstractions. By simply providing definitions for a new machine, users can port SimpleScalar code to a new platform.

Currently, SimpleScalar supports a diverse collection of host platforms, ranging from Windows/NT to Linux/x86 to Sparc/Solaris. We distribute SimpleScalar with a self-test mechanism that validates its build. The self-test simulation includes binaries, comparing their output to reference results. If the tests succeed, users can be confident that their SimpleScalar build works properly.

We have found that instructors are early adopters of research infrastructure, which means that both they and their students need support. Although the number of student users can grow quickly, their support requirements are quite similar, and we have found that we can proactively service their needs by providing support materials to their instructors. Recently, we released an instructor's kit that provides the components necessary to employ SimpleScalar in the classroom. The toolset includes a small collection of popular benchmarks with both

SimpleScalar's software architecture has been improved and modularized, making the code easier to understand and modify.

inputs and reference outputs as well as teaching materials and suggested projects at all levels of instruction.

LOOKING FORWARD

During the decade that SimpleScalar has been in use, we have continually adapted the infrastructure to researchers' needs. We will continue to improve the toolset and move it into new research areas by extending its capabilities. Currently, we are working toward three major enhancements, reflective of trends in computer architecture research: support for embedded system modeling, enhanced modeling capabilities, and development of sustainable user support services.

Embedded system modeling

The computer architecture research community has expressed growing interest in exploring embedded targets that support diverse applications and specialized hardware such as digital signal processors. In response to this trend, we are implementing an ARM target for SimpleScalar. The distribution will include pipeline and memory system models for the Intel StrongARM SA-1110 and XScale SA-2 microprocessors.

The ARM target supports the ARM7 integer instruction set and FPA floating-point extensions. The target I/O emulator supports ARM Linux call emulation and complete device-level emulation of the Compaq iPAQ platform. The device-level iPAQ emulator implements a real-time clock, interrupt controller, flash memory, and serial devices. A test release of the SimpleScalar/ARM distribution is available at http://www.simplescalar.com.

For future releases, we are working with the University of Michigan's Trevor Mudge to implement a TI C30 digital signal processor target. The C30 is an embedded target device that performs audio, video, or signal processing. SimpleScalar's C30 target model can be combined with a general-purpose processor model, such as the ARM, to allow modeling of heterogeneous multiprocessors.

Enhanced modeling capabilities

To date, SimpleScalar development has concentrated primarily on creating facilities for performance modeling. This emphasis served users well throughout the 1990s, when improving performance was a primary research focus. Looking forward, however, other design constraints, such as power dissipation and system reliability, are growing in importance. Power concerns cut across all market segments, from high-end systems where power dissipation rates affect system cooling costs, to low-end embedded systems where power requirements define battery life. Reliability concerns are receiving more attention as deep submicron fabrication technologies compromise transistor reliability.

We are working with Mudge to develop PowerAnalyzer, an infrastructure capable of modeling power and energy requirements for a wide range of pipelines and memory systems. PowerAnalyzer incorporates high-fidelity physical microarchitecture models into SimpleScalar. These models can accurately measure dynamic and static power dissipation. In addition, we are adding reliability analysis tools to SimpleScalar, which will make it possible to inject faults into a model and monitor its response.

Many SimpleScalar users have made suggestions for improving the toolset's modeling capabilities, which we implement in the MicroArchitecture Simulation Environment.[5] MASE improves the speed and ease with which users can build complex microarchitecture models by providing higher-level abstractions for constructing models, implementing improved underlying simulation facilities, and introducing an infrastructure for validating complex models.

To simplify the modeling of advanced speculative microarchitectures, MASE includes a facility that can recover model state to any arbitrary instruction. The memory interface has been updated to model devices with nondeterministic latency, permitting the study of advanced memory systems. The baseline performance model has been updated to more accurately represent modern microarchitectures. A checker component provides extensive debug and validation support, which assists users in locating inaccurate or incorrect performance-model features. Finally, the software architecture has been improved and modularized, making the code easier to understand and modify.

Sustainable user-support model

As the SimpleScalar community continues to grow, we must explore new avenues for supporting additional users. To date, its developers, users, and funding sources—the National Science Foundation and the Defense Advanced Research Projects Agency—have generously supported SimpleScalar. Moving forward, we will rely on commercial users to build a sustainable support model. In 1999, we founded SimpleScalar LLC to provide a commercial licensing service for

SimpleScalar tools. Commercial licenses for the toolset are fairly inexpensive compared to similar CAD tools, and commercial users pay only a one-time fee. To ensure that commercialization does not interfere with the academic use of SimpleScalar, our company provides a low-cost commercial use license to implement technology transfer between academic researchers using the toolset and corporations funding their research.

We seek to grow SimpleScalar's commercial market so that it can fully sustain the user community's needs. To date, companies in the field have been eager to support this effort. We are optimistic that by building this revenue mechanism, SimpleScalar will continue to serve as a valuable resource to researchers and instructors interested in the design, evaluation, and optimization of computing systems. ■

Acknowledgments

This work was supported by the NSF CADRE program, grant no. EIA-9975286, and DARPA Award No. F33615-00-C-1678. Eric Larson is supported under a National Science Foundation Graduate Fellowship.

References

1. R.M. Bentley, "Validating the Pentium 4 Microprocessor," *Proc. Int'l Conf. Dependable Systems and Networks* (DSN-2001), IEEE CS Press, Los Alamitos, Calif., 2001, pp. 193-198.
2. E. Schnarr and J. Larus, "Fast Out-of-Order Processor Simulation Using Memoization," *Proc. 8th Int'l Conf. Architectural Support for Programming Languages and Operating Systems* (ASPLOS-VIII), ACM Press, New York, 1998, pp. 283-294.
3. C. Weaver et al., "Performance Analysis Using Pipeline Visualization," *Proc. IEEE Int'l Symp. Performance Analysis of Systems and Software* (ISPASS-2001), IEEE CS Press, Los Alamitos, Calif., 2001, pp. 18-21.
4. R. Desikan, D. Burger, and S.W. Keckler, "Measuring Experimental Error in Microprocessor Simulation," *Proc. 28th Ann. Int'l Symp. Computer Architecture* (ISCA-28), ACM Press, New York, 2001, pp. 266-277.
5. E. Larson, S. Chatterjee, and T. Austin, "MASE: A Novel Infrastructure for Detailed Microarchitectural Modeling," *Proc. IEEE Int'l Symp. Performance Analysis of Systems and Software* (ISPASS-2001), IEEE CS Press, Los Alamitos, Calif., 2001, pp. 1-9.

Todd Austin *is an assistant professor in the University of Michigan's Electrical Engineering and Computer Science Department. His research interests include computer architecture, computer system verification, and performance analysis tools and techniques. Austin received a PhD in computer science from the University of Wisconsin-Madison. Contact him at taustin@eecs.umich.edu.*

Eric Larson *is a PhD candidate in the University of Michigan's Electrical Engineering and Computer Science Department. His research interests include computer architecture, compilers, software development, and computer system simulation. Contact him at larsone@eecs.umich.edu.*

Dan Ernst *is a PhD candidate in the University of Michigan's Electrical Engineering and Computer Science Department. His research interests include computer architecture and VLSI design. Contact him at ernstd@eecs.umich.edu.*