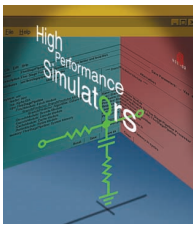
Simics: A Full System Simulation Platform



A full system simulator attempts to strike a balance between accuracy and performance by modeling the complete final application and providing a unified framework for hardware and software design within that context.

Peter S. Magnusson
Magnus Christensson
Jesper Eskilson
Daniel Forsgren
Gustav Hållberg
Johan Högberg
Fredrik Larsson
Andreas Moestedt
Bengt Werner
Virtutech AB, Stockholm

That all computers can simulate each other is an immediate consequence of the theoretical work of Alan Turing and Alonzo Church. Computer architects made direct use of this property as early as the EDSAC project in the 1950s,[1] and simulation in its various shapes and guises has been used to support the design of computers ever since. Simulation offers the traditional benefits of software construction: It can arbitrarily parameterize, control, and inspect the system it is modeling—the *target* system. Its measurements are nonintrusive and deterministic. Further, it provides a basis for automation: Multiple simulator sessions can run in parallel, and sessions can be fully scripted.

Naturally, we wish to simulate an entire system and to do so with total accuracy—a perfect model. There are obvious problems with seeking perfection, including cost, time to completion, specification inaccuracies, and implementation errors. But most important is the problem of workload realism. In most cases, we do not know how to implement an accurate model with performance sufficient to run realistic workloads. So, in practice, models that attempt to be highly accurate end up running very small "toy" workloads. The result is accurate answers to irrelevant questions.

Simics is a platform for full system simulation, which attempts to strike a balance between accuracy and performance. That is, it is sufficiently abstract to achieve tolerable performance levels with, at the same time, sufficient *functional* accuracy to run commercial workloads and sufficient *timing* accuracy to interface to detailed hardware models. Simics was one of the first academic projects in this area. It is the first commercial full system simulator, and it is just beginning to demonstrate the possibilities for system development.

FULL SYSTEM SIMULATION

Increasingly, we must design computer hardware or software within the context of the final application. A component's value lies in its contribution to that application. For example, no particular signal or executed instruction provides value at Amazon.com's Web site. Instead, the value lies in letting a customer expediently execute a search and make a purchase decision. An end-user service like this is usually built from a mixture of different manufacturers' equipment, in turn running a mixture of standard software and proprietary components. The availability, performance, and reliability of that end-user service motivates the entire digital value chain up to that point.

Large projects aimed at developing high-end digital systems employ a variety of simulation-oriented tools and methodologies. We can classify these along two dimensions: scope (what is being modeled) and level of abstraction (the abstraction level at which it is modeled). Abstraction level, in turn, is best viewed from two perspectives: the functional behavior ("what") and the timing behavior ("when").

If the goal is to model realistic workloads, then the scope must be the full system or we will not be able to represent modern scenarios at all. The abstraction level must be functionally low enough to boot and run unmodified commercial operating systems and industry benchmarks, and temporally low enough to support hardware engineering. However, the descent into more detailed levels of abstraction must not result in an overall simulation

0018-9162/02/$17.00 © 2002 IEEE

Figure 1. Simics simulation of target systems based on several processor architectures. Each simulated system can run unmodified operating systems and applications.

performance that precludes realistic workload scale, in terms of data set sizes and execution lengths. Today, a high-end workload scenario has a total code base of 10^5 to 10^8 lines, with execution lengths of 10^9 to 10^{12} instructions operating on a physical memory of 10^8 to 10^{11} bytes, with backing storage of 10^{10} to 10^{13} bytes.

Full system simulation supports the design, development, and testing of computer hardware and software within a simulation framework that approximates the final application context. In this case, "system" does not mean some arbitrary subset of digital components running simple test code. Referring to the Amazon.com example, it would include multiple Windows/Linux desktop clients connected over a network to a cluster of workstations and servers running Web software, databases, and various application-specific tasks related to the value proposition.

SIMICS OVERVIEW

Simics provides such a simulation platform. We designed it from the ground up to be sufficiently detailed to run unmodified operating systems (including both embedded systems such as VxWorks and general-purpose desktop/server systems such as Solaris, Linux, Tru64, and Windows XP). It is fast enough to run realistic workloads, including the SPEC CPU2000 benchmark suite, database benchmarks such as TPC-C, interactive desktop applications, and games. Simics is also sufficiently generic to model embedded systems, desktop or set-top boxes, telecom switches, multiprocessor systems, clusters, and networks of all these items. At the same time, Simics is flexible enough to support a broad variety of tasks throughout the product development cycle, including such seemingly disparate activities as microprocessor design, operating system development, fault injection studies, and hardware design verification.

Simics simulates processors at the instruction-set level, including the full supervisor state. Currently, Simics supports models for UltraSparc, Alpha, x86, x86-64 (Hammer), PowerPC, IPF (Itanium), MIPS, and ARM. Simics is pure software, and current ports include Linux (x86, PowerPC, and Alpha), Solaris/UltraSparc, Tru64/Alpha, and Windows 2000/x86.

Figure 1 shows multiple instances of Simics simulating target systems based on a variety of different processor architectures, each running a corresponding operating system:

- An x86 (Pentium II) machine running Red Hat 6.2 and a KDE desktop (large window in the center), showing two Netscape sessions connected to actual, live Web servers;
- A second x86 machine (top right) showing the Windows NT login screen;
- An UltraSparc II machine running Solaris 8 and MySQL (middle left);
- A Simics command line for an UltraSparc III model before "powering on" (bottom left);
- An IPF (Itanium) model running Red Hat 7.2 (top left);

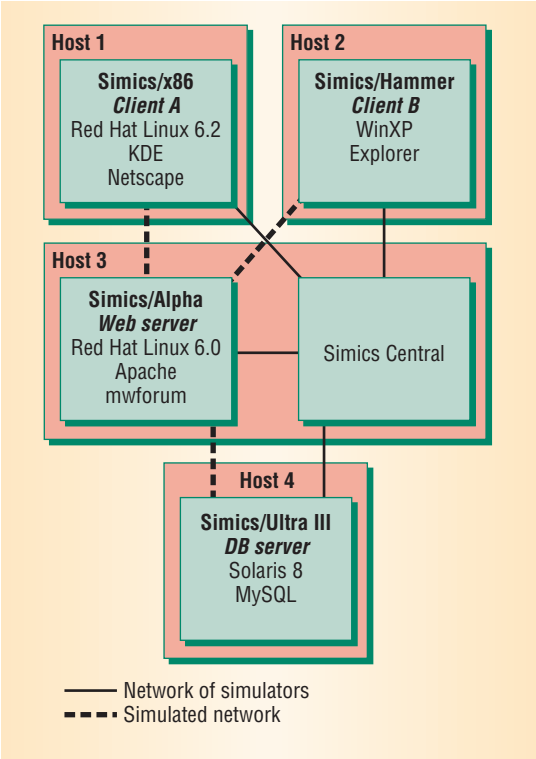


Figure 2. Sample network setup for Simics simulation. The simulation is distributed over four host workstations, and two clients are talking to the Web server, which has a database back end.

- A PowerPC machine running VxWorks (top center); and
- An x86-64 (Hammer) machine running Windows XP (the simulated processor is running in 32-bit legacy mode), in the bottom right window.

The window in the bottom-left corner of Figure 1 shows the Simics command line. All other Simics command windows are hidden. The screen shot is taken from a dual-processor 933-MHz Pentium III system with 512 Mbytes of memory, running Red Hat Linux 7.2. All the Simics processes are running on that same system.

In addition to processor models, Simics includes device models accurate enough to run the real firmware and device drivers. For example, Simics/UltraSparc III will run the real Open Boot PROM, and Simics/x86 will correctly install and run Windows XP from the installation disks.

Simics views each target machine as a node, representing a resource such as a Web server, a database engine, a router, or a client. A single Simics instance can simulate one or more nodes of the same basic architecture. Heterogeneous nodes can be connected into a network controlled by a tool called Simics Central. In Figure 1, Simics Central is used to connect the two Netscape sessions in the central desktop to real Web servers.

Simics Central is a key component and allows the creation of full-scale distributed systems. Figure 2 shows a sample network setup, demonstrating a Web-based online discussion forum using a three-tier database solution. The simulation is fast enough to use interactively. Through the mouse or a keyboard, users can give input to clients A or B, which run within windows on the respective host window systems. Simics Central acts as a router that lets users traceroute into the simulated network from the host environment (and vice versa).

With this setup, we can interactively browse the different discussion groups on the Web server and write new messages with acceptable response. Retrieving the first Web page that includes a list of all discussion groups takes approximately 30 seconds.

The point here, of course, is that this is a fully simulated setup. For example, any Simics session can be stopped to single step, inspect state, and so on, in which case the other Simics processes automatically pause pending simulated global time progress. The simulation can access memory traffic anywhere, set breakpoints anywhere, and modify any of the systems (such as adding new instructions or caches). It can record and timestamp all user input—for example, keyboard and mouse—and play back the entire session. The simulation can also save the entire setup to a checkpoint and bring it up again in repeat sessions.

APPLICATIONS FOR FULL SYSTEM SIMULATION

Figure 3 shows the major task dependencies in developing a complete high-end digital system. Note that full system simulation relaxes many dependencies by providing a single platform across the development cycle. Each task can move from an existing system model toward the intended next-generation model at its own pace. In other words, each task can begin within an abstract and—by design—incorrect context, which gradually becomes more representative of the real future system. The time-to-market gains and risk reductions apply across the electronics industry.

Simics is currently being used commercially in all the areas listed in Figure 3.

Microprocessor design

Next-generation processor design was an early application area for simulation. Traditional trace-based simulation techniques have some well-known limitations that full system simulation tries to resolve. Most importantly, multiprocessor workloads interact with the memory subsystem and the operating system memory management and sched-

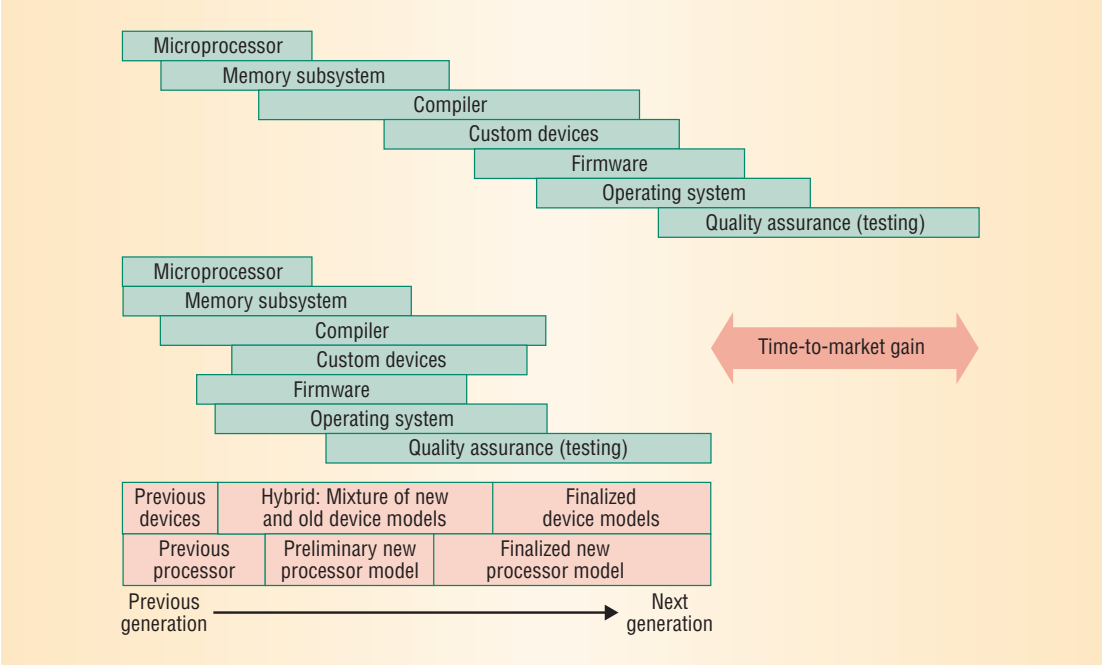


Figure 3. Task dependencies for developing a high-end digital system (top) are relaxed in full system simulation (middle), reducing time to market.

uling in a manner that is difficult to capture in a trace from a dissimilar setup.

Simics facilitates the inclusion of approximate cache and I/O timing models, allowing a first-order approximation of the interleaving of memory operations for a next-generation system. This approximation serves as a platform for generating traces for use as input to traditional cycle-accurate microarchitecture models. Simics can be scaled to full-blown server workloads such as TPC-C, but in practice the workloads are frequently scaled down.

Some recent Simics processor models include early support for out-of-order processing. These models have unlimited execution units and renaming registers, with a configurable reorder buffer and dispatch rate, but no pipeline. Simics handles data dependencies and can roll back when it encounters exceptions. This rudimentary support for out-of-order and multiple outstanding memory operations is more in line with what occurs on real systems and proposed future systems. We are extending this support to facilitate attaching a full-blown next-generation microarchitecture model, the philosophy being that Simics provides the functional model, and an investigator provides the timing model.

Memory studies

The simulated memory in Simics is represented by one or more *memory spaces*, which usually correspond to the address spaces found in a real system. Typical examples include physical cacheable memory, PCI bus spaces, and I/O address spaces.

Users can extend a memory space by connecting a timing model to it, typically to affect how long memory accesses take, but also to collect input to cache and memory simulators or to generate a trace.

Device development

Simics has an interface for communicating with external programs. These programs can simulate a single device, a memory bus, or anything connected to a memory bus. Users can test a new device by connecting it to Simics and having the Simics' I/O traffic drive the device during the test.

Operating system emulation

Prior to the availability of a target operating system, developers can run user-level applications on a "bare bones" simulator by using an operating system emulation layer. The OS emulation layer does not need to execute on the simulated hardware itself, which means we can isolate the behavior of a single application. For example, in an emulated OS environment, interrupt and exception handlers will not interfere with the measurements of an application's cache behavior. The OS emulation layer can be implemented in a scripting language. For educational purposes, a kernel implementation in a scripting language legibly demonstrates the OS internals and provides an easy-to-modify platform for practical experiments and studies.

For example, in developing support for new processors, work on compilers requires a very large number of simulated cycles early in the project. Simple OS emulation lets compiler designers run user-level compiler regression tests independent of progress on the "real" operating system.

OS development

Bringing up firmware and operating systems prior to hardware availability is a classic use of full system simulation. For example, SuSE has ported Linux to AMD's x86-64 architecture using the Hammer version of Simics, and Wasabi Systems has

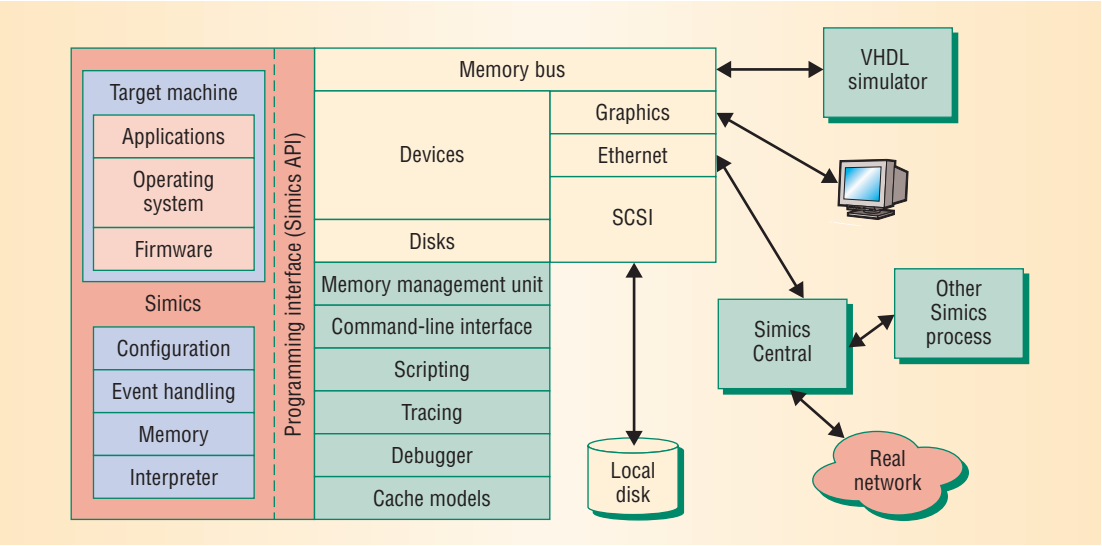


Figure 4. Simics architecture. A core module offers basic simulation features such as processor instruction set and memory. A broad application programming interface makes Simics a "platform." The API is used for adding specific device models and intrinsic components such as a command line.

ported NetBSD in the same manner. By using old devices together with a new processor architecture, the core porting work can ignore most driver issues.

Even when the hardware is available, a simulator offers classical benefits. For firmware development, the ability to implement very specific breakpoints is useful—such as stopping the execution upon reads from specific control registers.

Debugging

A simulator provides a powerful set of methods for locating bugs compared with traditional debuggers. Simics supports traditional debugging tasks, like loading symbolic information, setting breakpoints, and single stepping. Its access to the entire system state, however, also lets developers inspect the state of devices and the operating system.

Repeatability is a particularly useful feature for debugging. Interactive keyboard and mouse input, as well as network traffic, can be played back, causing the exact event flow that triggered a bug to be repeated. Combining this information with a checkpoint right before the point of failure greatly facilitates time-efficient debugging.

Developers use scripting support to implement advanced breakpoints. For example, write breakpoints can check lock semantics around data structures, and timing breakpoints can trigger when two program points are executed too far apart.

Developers can also attach external debuggers. The Simics "gdb-remote" module implements the TCP/IP remote debugging protocol of the GNU Debugger (gdb). Since the remote gdb session works concurrently with Simics' command line, users can combine a well-known debugger interface with other Simics features.

High-availability testing

Simics supports tests of system characteristics such as reliability, performance, and fault tolerance that could not be tested, in any practical way, without simulation. A simulator can modify conditions almost arbitrarily to introduce failure behaviors. This has obvious benefits compared with testing the physical hardware. It provides a higher level of information about error paths, and it can replicate discovered errors. In addition, the testing can be automated by using checkpoints and scripts, and the cost is much lower.

SIMICS IMPLEMENTATION

Figure 4 shows an overview of the Simics architecture, which has been under development for more than a decade. The current version has absorbed more than 50 person-years of development and constitutes close to one million lines of code.

Simics Central

Simics Central synchronizes the virtual time between Simics simulators and distributes simulated traffic between the nodes. It imposes a minimum latency on every message passed, allowing the entire distributed simulation to be fully deterministic. To overlap host network latency with simulation, Simics Central uses a two-phased clocking scheme to pass synchronization messages.

Simics Central currently supports Ethernet networks, but other types of networks can be added using a modular infrastructure. The network adapter modules (such as AM79C960) connect to the Ethernet-central module in Simics Central.

Simics runs simulations as fast as it can, but Simics Central will halt the simulation if one process consumes cycles slower than the rest. In other words, the network simulation speed is equivalent to the speed of the slowest Simics process.

Configuration

Simics uses a simple object-oriented configuration language to describe the target system. An object corresponds to a processor or device in the target machine or to a "virtual" object such as vir-

```
from sim_core import *
import conf

def break_handler(id):
   if conf.cpu0.eax > conf.cpu0.ecx:
      raise SimExc_Break

id = SIM_breakpoint(conf.phys_mem0,Break_Physical, Break_Execute,
       0x000f2501, 1, 0)

SIM_hap_register_callback("Core_Breakpoint" ,break_handler, id)
```

Figure 5. Sample conditional breakpoint written in Python. The code installs a callback handler, which is called when the instruction at 0x000f2501 executes.

tual-to-physical memory mappings and disk images. The objects are instantiated from classes that are defined by runtime-loadable modules. To add a device, developers write a loadable module using the Simics application programming interface (API) to implement a class, then they define an object of that class in the configuration file.

For example, the following listing shows parts of a configuration file for a future desktop PC with 256 Mbytes of memory.

```
OBJECT cpu0 TYPE x86-hammer
{
   freq_mhz: 3500
   physical_memory: phys_mem0
}
OBJECT phys_mem0 TYPE memory-space
{
   map: ((0xa0000, vga0, 1, 0,
         0x20000),
      (0x100000, mem0, 0, 0x100000,
         0xff00000),
       ...

}
OBJECT con0 TYPE gfx-console
{
   queue: cpu0
   x-size: 720
   y-size: 400
   keyboard: kbd0
   mouse: kbd0
}
```

The file defines objects for the processor, physical memory space, and graphical console. The configuration system implements checkpoints by saving all objects and their attributes to disk in a mostly human-readable text format.

CLI and scripting

Simics is controlled primarily through the command line interface (CLI), which is similar to the front end of a debugger. Simics also has a built-in Python runtime environment, which loads Python scripts and executes them directly from the CLI. In fact, the CLI is written in Python, using the Simics API.

Scripts can be tied to certain events, such as translation look-aside buffer (TLB) misses and I/O operations. The code in Figure 5 is a Python example of a conditional breakpoint. It installs a callback handler, which is called when a breakpoint is triggered (in this case, when the instruction on address 0x000f2501 executes). If register EAX is greater than register ECX, the handler signals a break, and the simulation will stop; otherwise, the simulation continues. Note how all objects reside in the `conf` module and are mapped to Python objects.

Devices

For each target, Simics supports a device set that enables firmware and operating systems to boot and run. For the x86 ("PC") target, for example, Simics supports legacy ISA devices such as a timer (8254), a floppy controller (82077), a keyboard/mouse controller (8042), direct memory access (8237), an interrupt controller (8259), and a real-time clock/nonvolatile RAM (DS12887). Other Simics/x86 devices include an interrupt controller (APIC, I/O-APIC), a host-to-PCI bridge (82443BX), an IDE controller, a VGA adapter, an accelerated 3D graphics (Voodoo3-based) card, and Ethernet adapters (AM79C960 and DEC21140A).

Target processors usually imply a family of CPUs—for example, the x86 includes 486sx, Pentium, Pentium MMX, Pentium Pro, and the Pentium II. Simics supports multiprocessor system models for all targets.

Interfacing to other simulators

As the memory bus in Figure 4 indicates, Simics can interface to a clock-cycle-accurate model written in a hardware description language (HDL) such as Verilog.

Verilog has defined interfaces, which allows it to link C functions that are called from HDL components. Thus, Verilog can link the Simics communication layer into HDL, and the HDL program can drive Simics by telling it to advance simulation a fixed time unit, typically one clock cycle at a time. When signals pass between Simics and the HDL sim-

```
// IA32/x86-64 add to left instruction
instruction ADD_L({REG}, {REG_OR_MEM})
   pattern
      op_h2 == 0 && op1 == 0 && d == 1
         && opm == 0
   syntax
      "add {REG},{REG_OR_MEM}"
   semantics
      #{
      ireg_t op1 = {REG};
      ireg_t op2 = {REG_OR_MEM};
      ireg_t dst = op1 + op2;
      EFLAGS_ADD(dst,op1,op2,w,os);
      SET({REG_W}, dst);
      #}
   attributes
      type = IT_ALU
```

Figure 6. SimGen spec for an IA32/x86-64 add-to-left instruction.

ulator, their respective abstraction levels must be translated. For example, Simics models memory reads as atomic, so if the HDL simulator models a split-transaction memory bus, it must break the read into a read request and a data-reply bus transaction.

Also, Simics supports multiple outstanding memory transactions to generate realistic traffic patterns by keeping a list of the instructions it is currently executing. When data arrives at Simics from the HDL model, it triggers execution of previously stalled instructions in the same way a modern out-of-order processor does. Simics uses this out-of-order support to generate a reasonable stream of memory traffic to the external simulator.

Simics application programming interface

A major feature of Simics is its extensibility, allowing users to write new plug-in device models, add new commands, or write control and analysis routines. The Simics API has more than 200 exported functions, several data types, and more than 50 predefined interfaces. These interfaces are collections of function pointers, similar to method tables, that Simics uses for all interobject communication. The API is written in C, but it is also mechanically exported to Python.

Memory

Memory operations are the biggest performance challenge for a full system simulator. Simics uses a simulator translation cache (STC) to speed up loads and stores and instruction fetches. These caches store pointers to simulated memory and are indexed by virtual addresses. A hit in the STC guarantees that there are no side effects, such as an alignment exception, TLB miss, cache miss, or simulator breakpoint. For instruction fetches, the STC stores pairs of addresses, representing branch arcs that can be safely traversed.

Essentially, the STC works as a cache for the interpreter, with the common case (that is, an STC hit) sufficiently simple to be inlined in the interpreter kernel. The need for generality is a significant complication: Simics needs to handle various combinations of host-target endianity and address space sizes, for example. Altogether, the STC design may be the most complex construct in the simulator.

Threaded-code interpreter

At the center of any full system simulator is an interpreter kernel. The simulated CPU models include the entire software-visible spectrum—exception-interrupt models, control registers, and so on. Some processors include microcode-like features; thus Simics/Alpha supports PAL (Privileged Architecture Library) code.

There are many efficient ways to write interpreters, including threaded-code interpreters and variations of runtime code generation. Implementing an efficient simulator by hand is a labor-intensive and error-prone task. We developed a specification language, SimGen, to encode various aspects of the target instruction-set architecture. SimGen includes the syntax and encoding of instructions, as well as C code for the semantics and high-level attributes used with timing models. Optimizations include a sequence of partial evaluations, some suggested in the specification and some made automatically.

Figure 6 shows the spec for an IA32/x86-64 add-to-left instruction. This example omits all macro definitions, which would total more than 100 lines. SimGen uses special macros to express the repetitive, combinatorial, and contextual nature of instruction-set architectures. The SimGen tool generates all permitted combinations, and it uses instruction-frequency statistics to guide the generation of specialized service routines, the output being an interpreter in C. Essentially, the Simics kernel is synthesized from a high-level specification.

Event handling

Simics supports a general event-handling mechanism. Each processor object has two event queues: a step queue and a time queue.

In the step queue, events appear after a number of *program counter steps*. The step count is the sum of successfully completed instructions, issued instructions that caused exceptions, and interrupts handled at the system level.

The time queue has a resolution of a processor clock cycle, which is a fixed time unit set in the configuration (such as in object *cpu0* mentioned above). Simics can schedule multiple events at the same cycle-step, and it handles the queue in FIFO

order. When posting an event to the time queue, Simics can synchronize the time queues of all processors for events that affect global state. This dual-queue design allows Simics to mix event-driven and time-driven components.

PERFORMANCE

Table 1 summarizes Simics performance. For simplicity, we have chosen a variety of OS boot workloads that model seven different processor architectures: Alpha EV5, UltraSparc II and UltraSparc III, Intel Pentium II, AMD x86-64 Hammer, Intel IPF (Itanium), and PowerPC 750. For comparability, we performed all measurements on an Intel P-III 933-MHz host with 512 Mbytes of RAM running Linux.

To demonstrate scalability, Table 2 shows times to boot Solaris 8 on simulated Ultra II Enterprise server systems to multiuser login prompt for various configuration sizes. The host is a 750-MHz UltraSparc III system. The time of a Solaris boot depends on the OS version, what devices are present, the amount of memory, the clock frequency, system services, and so on. Millions of instructions per second (MIPS) on multiprocessor models are higher because we disable idle CPUs during the first phase of the boot.

For the out-of-order versions of the UltraSparc, performance is obviously much lower. In the Solaris 8 boot example, the out-of-order version simulating a 16-Kbyte data cache runs at 0.3 MIPS compared to the 6.62 MIPS in Table 2.

Table 1. Simics performance of target systems for a variety of operating-system boot workloads.

Target	Boot workload	Instructions	Time (sec)	MIPS
Alpha-ev5	Tru64	2,112,119,247	354	5.9
Alpha-ev5	Linux	1,201,600,120	164	7.3
Sparc-u2	Solaris 8[1]	1,597,537,438	284	5.6
Sparc-u3	Solaris 8[1]	6,155,835,717	987	6.2
x86-p2	Linux[2]	1,299,639,608	227	5.7
x86-p2	Windows XP	3,129,351,000	1,518	2.1
x86-64	Linux[2]	1,299,639,608	285	4.5
Itanium	Linux	4,644,372,142	1,470	3.2
PPC-750	VxWorks	1,179,516,468	136	8.7
PPC-750	Linux[3]	498,836,969	53	9.3

[1] The configurations for the UltraSparc machines differ in more ways than just the CPU. The Sparc-u2 was configured to simulate an enterprise server, while the Sparc-u3 was configured as one of Sun's new "Sun Fire" servers (Serengeti). The different CPU clock frequencies configured on the machines (750 MHz on Sparc-u3 and 168 MHz on Sparc-u2) account for most of the difference in the number of simulated instructions.

[2] These are the same benchmarks, but Simics/x86-64 runs in "legacy mode," which means that it is IA32 compatible.

[3] This is a minimal Linux installation; no services are started.

Table 2. Times to boot Solaris 8 on simulated Ultra II server systems.

No. CPUs	Instructions/CPU	Time (sec)	MIPS	MIPS/CPU
1	3,032,350,964	443	6.62	6.62
2	2,957,823,001	505	11.35	5.68
4	2,952,203,000	610	19.17	4.79
8	2,989,656,000	931	25.12	3.14
16	3,068,869,000	1,340	36.29	2.27
30	3,212,321,000	2,554	37.41	1.25

RELATED SYSTEM SIMULATION WORK

IBM developed the first modern emulator,[2] which permitted programs written for the IBM 7070 to run on one of the larger System/360 computers (for a good summary of early work on emulation and simulation in industry, see the 1979 article by Michael Canon and colleagues[3]).

Early work in academia included the PDP-11 emulator developed by John Doyle and Ken Mandelberg[4] and the implementation of g88 by Robert Bedichek.[5] The g88 implementation was subsequently placed in the public domain, and the design details were published. This implementation modeled a uniprocessor M88100-based system with a mixture of real and pseudo devices, and it could boot an operating system (specifically, Unix). A predecessor of Simics, gsim, begun in 1991, was based on g88 and extended to include support for multiple processors with shared physical memory.[6] In 1994, the gsim simulator was rewritten as a multiprocessor Sparc V8 model, resulting in the first version of Simics.

More recently, SimOS has modeled large parts of a MIPS-based multiprocessor,[7] and it has booted and run a modified Irix kernel. SimOS and Simics have pursued similar goals and have arrived at similar solutions on many issues. Indeed, they have been developed partly in parallel. For example, the current scripting platform in Simics originated in the SimOS work on annotations.[8]

Simics is distinguished from earlier simulation work by implementing "best practices." Many of the possibilities of full system simulation have been obvious to practitioners in both academia and industry for quite some time, perhaps decades, but Simics supports more of these possibilities within a single framework than other tools.

For example, running as realistic code as possible is important. In contrast to similar tools that we are aware of, Simics can run actual firmware and completely unmodified kernel and driver code. In fact, researchers are using Simics to develop and

test firmware for several future desktop and server systems.

Simics is unique in being able to run a heterogeneous network of systems from different vendors within the same framework. Simics is a fast tool with an abstraction level that makes it easy to add new components and leverage older ones. Simics offers a practical platform, with a rich API and a powerful scripting environment for use in a broad range of applications.

We believe Simics marks the starting point for a different way of designing, testing, and implementing high-end digital systems. ■

References

1. S. Gill, "The Diagnosis of Mistakes in Programmes on the EDSAC," *Proc. Royal Society Series A, Mathematical and Physical Sciences*, vol. 206, Cambridge Univ. Press, Cambridge, UK, 1951, pp. 538-554.
2. S. Tucker, "Emulation of Large Systems," *Comm. ACM 8*, pp. 753-761. As cited in E.W. Pugh, *Building IBM*, MIT Press, Cambridge, Mass., 1995.
3. M.D. Canon et al., "A Virtual Machine Emulator for Performance Evaluation," *Comm. ACM*, Feb. 1979, pp. 71-80.
4. J.K. Doyle and K. Mandelberg, "A Portable PDP-11 Simulator," *Software Practice and Experience*, Nov. 1984, pp. 1047-1059.
5. R.C. Bedichek, "Some Efficient Architecture Simulation Techniques," *Proc. Winter 90 Usenix Conf.*, Usenix Assoc., Berkeley, Calif., 1990, pp. 53-63.
6. P.S. Magnusson, "A Design for Efficient Simulation of a Multiprocessor," *Proc. Modeling, Analysis, and Simulation of Computer Systems* (MASCOTS 93), IEEE CS Press, Los Alamitos, Calif., 1993, pp. 69-78.
7. M. Rosenblum et al., "Using the SimOS Machine Simulator to Study Complex Computer Systems," *ACM Trans. Modeling and Computer Simulation* (TOMACS), Special Issue on Computer Simulation, Jan. 1997, pp. 78-103.
8. S.A. Herrod, *Using Complete Machine Simulation to Understand Computer System Behavior*, doctoral dissertation, Stanford Univ., 1998.

Peter S. Magnusson *is CEO of Virtutech. His technical interests include simulation, virtual machines, emulation, binary translation, and advanced software engineering. He received an MS in computer science from the Swedish Royal Institute of Technology and an MBA from Stockholm School of Economics. He is a member of the ACM and the IEEE. Contact him at psm@virtutech.com.*

Magnus Christensson *is a member of the technical staff at Virtutech. His research interests include dynamic code generation and computer architecture. He received an MS in computer science from the Swedish Royal Institute of Technology. He is a member of the ACM and the IEEE. Contact him at mch@virtutech.com.*

Jesper Eskilson *is a member of the technical staff at Virtutech. His technical interests include programming language design, user interfaces, and constraint programming. He has an MS in computer science from Uppsala University. He is a member of the ACM. Contact him at jojo@virtutech.com.*

Daniel Forsgren *is a member of the technical staff at Virtutech. His interests include computer architecture and operating systems. He has studied applied physics and electrical engineering at Linköping Institute of Technology. He is a member of the ACM. Contact him at daniel@virtutech.com.*

Gustav Hållberg *is a member of the technical staff at Virtutech. His technical interests include artificial languages and application design. He studied engineering physics at the Swedish Royal Institute of Technology. Contact him at gustav@virtutech.com.*

Johan Högberg *is a member of the technical staff at Virtutech. He received a University Certificate in computer technology from Karlstad University. Contact him at johan@virtutech.com.*

Fredrik Larsson *is a member of the technical staff at Virtutech. His technical interests include computer architecture and programming language design. He received an MS in computer science from Uppsala University. Contact him at fla@virtutech.com.*

Andreas Moestedt *is a member of the technical staff at Virtutech. His interests include computer architecture and operating systems. He received an MS in computer science and engineering from Lund University. He is a member of the ACM. Contact him at am@virtutech.com.*

Bengt Werner *is CTO at Virtutech. His research interests include accurate modeling and efficient simulation of digital systems. He received an MS in computer science and engineering from Lund University. Contact him at werner@virtutech.com.*