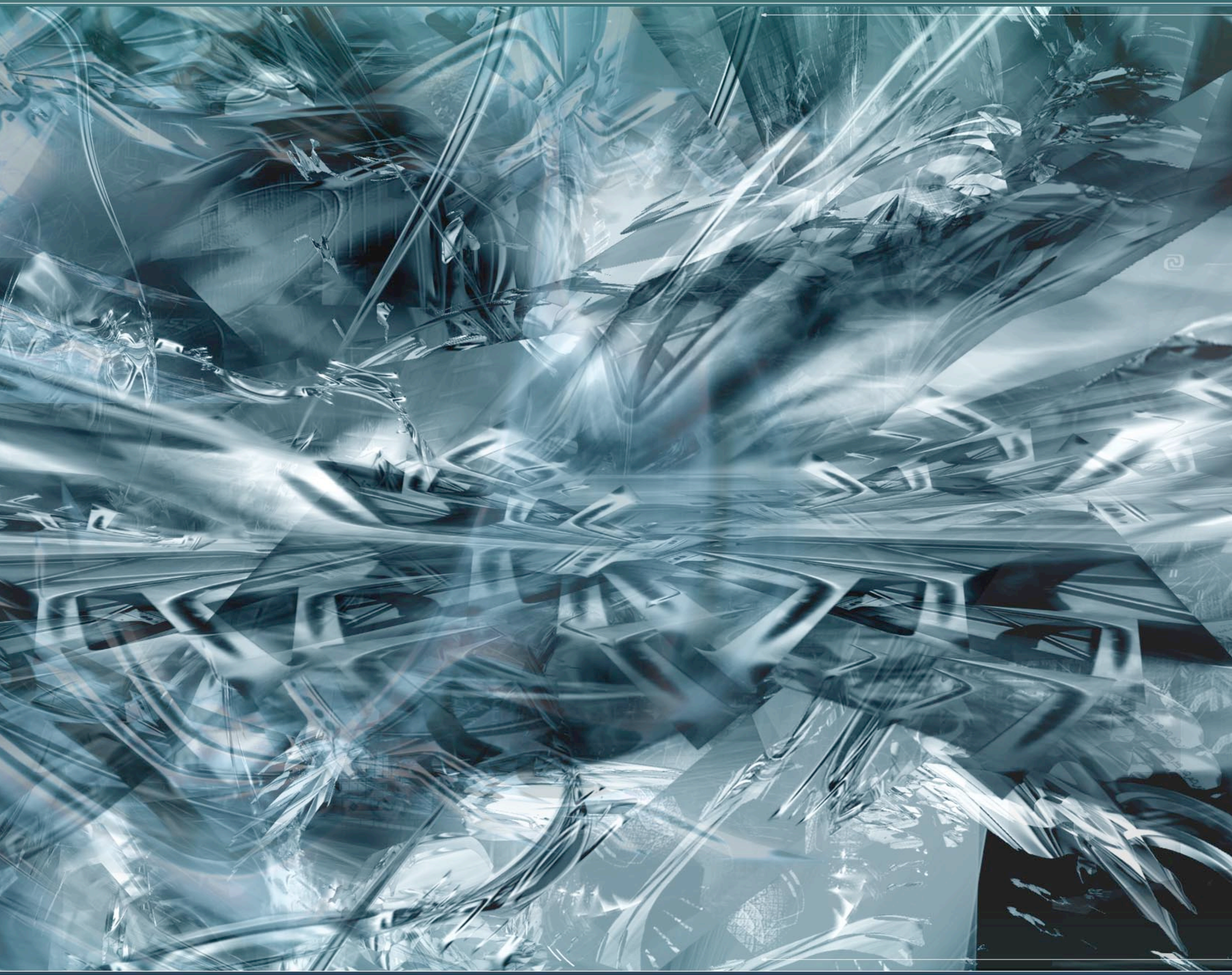

FOCUS
Virtual Machines

Has the time come to kiss that old iron goodbye?

BOB SUPNIK, SUN MICROSYSTEMS

Simulators

Virtual Machines of the Past (and Future)

Simulators are a form of "virtual machine" intended to address a simple problem: the absence of real hardware. Simulators for past systems address the loss of real hardware and preserve the usability of software after real hardware has vanished. Simulators for future systems address the variability of future hardware designs and facilitate the development of software before real hardware exists.

SIMH, the Computer History Simulation system, is a behavioral simulator for obsolete systems of historic interest. Originally intended as an educational project, it is increasingly being used in long-lived production environments as a substitute for real systems. SIMH is continuously being extended to simulate new machines.

A TAXONOMY OF VIRTUAL MACHINES

The term *virtual machine* has been used to describe at least three different types of computer programs:

1. The virtualized computer operating system (hypervisor) pioneered by VM/370 and today commercialized through such products as VMWare. Virtual hypervisors allow a single system to run multiple operating environments simultaneously.

2. The abstract computer system produced by interpreted environments such as Java and C# today (and many other projects in earlier times). Virtual machine interpreters allow code to be portable across incompatible instruction architectures.

3. A simulated or emulated computer system, like SIMH (http://simh.trailing-edge.com), which looks

Simulators

Virtual Machines of the Past (and Future)

back to the behavioral simulators of 35 years ago, such as MIMIC.[1] Simulators create virtual machines that can run code for other systems, typically because the other system no longer exists or does not yet exist.

All virtual machines share a common set of problems: (1) They must faithfully reproduce the target; (2) they must be able to faithfully reproduce the environment expected by the software (timing, I/O, etc.); and (3) they must deliver adequate performance in order to be a usable environment.

SIMH

SIMH is an open-source simulation system for obsolete computer systems. Its purpose is to preserve computing's legacy—both hardware and software—and to make systems of historic interest accessible to anyone with a personal computer. SIMH consists of a common framework known as SCP (Simulator Control Package) and individual simulators for more than 20 systems:

- DEC: PDP-1, PDP-4, PDP-7, PDP-8, PDP-9, PDP-10, PDP-11, PDP-15, VAX
- Data General: Nova, Eclipse
- IBM: 1130, 1401, 1620, System/3
- Interdata: 16- and 32-bit systems
- Others: Royal-Mcbee LGP-30, SDS 940, Honeywell H316, GRI-909

SIMH is portable and runs on Windows, Linux, and most other flavors of Unix, Mac OS X, and VMS (Virtual Memory System). The overall design of SIMH has been described elsewhere;[2] this article focuses on the practical implementation issues of writing a simulator, and the common problems that must be solved.

WRITING A SIMULATOR

Writing a simulator for an obsolete system requires historical research, software engineering, and detective work, in about equal parts.

HISTORICAL RESEARCH

The essential basis for any simulator is an accurate specification of the target system. While modern architectures are specified in great detail, descriptions of historical systems tend to be sketchy. In addition, time has created gaps in the historical record: manuals have been thrown out, schematics lost, and actual machine examples scrapped. As in most forms of historical research, primary sources (schematics, microcode listings, and maintenance documentation) are best; secondary sources such as handbooks, marketing material, textbooks, and even user manuals cannot be trusted.

The Internet provides multiple starting points for simulation research:

- Collectors, universities, and museums have transcribed and published their private archives of documentation.
- Search engines can provide pointers to Web pages and Web sites dedicated to historical computing or particular systems of interest.
- Documentation and, occasionally, complete systems are offered for sale on eBay.

Several newsgroups are devoted to obsolete systems and emulation.

Nonetheless, even for the most popular systems, survival of hardware and software is chancy. For example, Unix versions 1-4 appear to be irrevocably lost. On the other hand, the sources for PDP-15 XVM/DOS—not exactly a widely used operating system—turned up in a collection of DECtapes on eBay.

SOFTWARE ENGINEERING

Once adequate documentation is in hand, the next stage is designing and implementing the simulator. Certain design decisions are critical: How will the hardware system architecture be mapped into software? How will memory and I/O devices be represented? How will asynchrony (simulated time) be handled?

SIMH handles these issues as follows:

SIMH maps the hardware architecture using a variation of PMS (processor-memory-switch) notation.[3] Simulators are collections of devices; the CPU is just a device that executes instructions. Devices consist of registers, which hold state, and units, which contain data sets. For example, a disk controller is a device. Its registers are the controller state. The units represent the disk drives, each of which has a data set.

SIMH represents memory and I/O data sets by a uniform mapping into host system containers. All containers are C integer data types. For example, a 12-bit memory would be represented by a C unsigned short array (typically 16 bits). Containers for I/O devices are typically disk files, although they can also be mapped to real devices such as floppy or CD drives.

SIMH models the asynchronous operation of a computer system explicitly. The simulator keeps track of simulated time in any convenient unit (nanoseconds, instructions). Devices that operate asynchronously schedule events in "future time." When the simulator reaches the appropriate time, it calls the device event handler to execute the asynchronous operation.

Once the high-level design issues are decided, the simulator can be detailed, designed, and coded, usually from the CPU out to the peripherals. The CPU design has the most complexity: How will instructions be decoded and executed? How will the CPU communicate with I/O devices? How will exceptions and interrupts be handled? What debugging facilities should be included? A typical SIMH simulator handles these issues as follows:

Instruction execution models the behavior of the real system, with a fetch phase, an address decode phase, and instruction execution. Often, the instruction breakout is simply a large case statement. This is fast and models the structure of microcode, but can be bulky and difficult to read.

- The CPU implements formal, configurable interfaces to I/O devices, usually with the same basic operations as the real system's I/O bus. These interfaces allow an I/O dispatch table to be built at runtime so that I/O devices can be included or excluded as desired.
- Exceptions are often handled in the same way they are in the hardware: by a global trap (C `longjmp`) to a central exception routine. This is anathema in modern object-oriented languages but accurately models how many real systems work.
- Interrupts are modeled to balance accuracy and simulation speed. For complex priority interrupt systems, the best approach is a central routine to evaluate the state of the interrupt system after any event that can change it (for example, an I/O instruction).
- All simulators provide symbolic assembly and disassembly for memory, execution breakpoints, single execution step, and a PC change history. Some provide deep instruction trace capabilities or multiple types of breakpoints.

The simulator is now ready for initial debugging with hand test cases, or diagnostics if available.

What Are These Systems?

SIMH simulates systems that are of historic or architectural interest—or, in many cases, of personal interest to the author. For example, the LGP-30 was the first computer I ever saw. Built in the mid-1950s, it used a drum for memory (all 4,000 words of it). It had less computing power than today's disposable calculators. The IBM 1620 was the first computer I ever programmed. Built in the late 1950s, it was popular with universities because of its low cost (only $64,000—in 1960 dollars). It implemented decimal arithmetic with table lookups, leading to the nickname Cadet—"Can't add, doesn't even try."

The PDP-1, an 18-bit computer delivered in 1960, was Digital Equipment's first computer. It was used to develop the world's first video game, Spacewar. DEC's PDP-8, a 12-bit computer delivered in 1966, was considered the first minicomputer, because it cost less than $20,000 and consisted of only half a rack of logic instead of multiple racks. The PDP-8 was also the first mass-produced computer; more than 50,000 were manufactured.

The first 16-bit minicomputers were delivered in the mid- to late-1960s. Hewlett-Packard's 2100, Interdata's Model 3, and Honeywell's H516 were all early examples of 16-bit minicomputers. The H516 was used to implement the ARPAnet IMP (Advanced Research Projects Agency Network interface message processor), in effect the first router for the predecessor to the Internet. In 1970, the DEC PDP-11 and Data General Nova revolutionized minicomputer architecture. The elegant and complex PDP-11 was the most popular minicomputer ever and influenced most late-1970s systems, including the Intel x86; the x86's "little endian" byte order derives from the PDP-11. The radical simplicity of the Nova foreshadowed the RISC (reduced instruction set computer) architectures of the 1980s.

Thirty-two-bit computing broke out of the mainframe category with the introduction of the "supermini" Interdata 7/32 in the mid-1970s and then the VAX in 1977. The 7/32 hosted the first port of Unix, as well as the first port to a 32-bit system. The VAX was the most popular 32-bit computer of the 1980s, until it was overtaken first by RISC and then by PCs.

Simulators

Virtual Machines of the Past (and Future)

DETECTIVE WORK

The last stage is bringing up the operating system and other operational software. In theory, a simulator that runs diagnostics should run any operating system. In practice, this isn't the case for a number of reasons:

The software may be incomplete. Attempts to bring up the PDP-15 ADSS (Advanced System Software) were stymied by lack of the proper paper-tape bootstrap. Eventually, a PDP-9 bootstrap turned up in France, but no paper-tape reader was available to transcribe it. The collector in France scanned the tape in sections on a flatbed scanner and then wrote a program to recognize and transcribe the holes and splice the transcribed sections together. The PDP-15 simulator writer then discovered, through debugging the boot process of ADSS, what changes had occurred between the PDP-9 and PDP-15.

The software path may be untested. Attempts to bring up PDP-10 TOPS-10 7.04 uncovered a bug in the installation routine's handling of magnetic tapes. The "new system" installation path for 7.04 had never been tested, because by the time 7.04 shipped in 1988, the PDP-10 had been out of production for six years, and all installations were upgrades.

The simulator configuration may be untested. Simulators are much cheaper than real hardware, and it's easy to build a simulated configuration that would have been utterly impractical, in financial or physical terms, in real life. For example, initial bring-up of PDP-15 DOS failed when the software attempted to "size" the fixed-head disk. The simulator implemented a maximum-size disk of 2 million words (eight platters). In the real world, however, no such configuration had ever been built because it was too expensive. The sizing code contained a bug that looped indefinitely on a maximum configuration.

The software may depend on details of timing or implementation that are not simulated accurately. DEC's MSCP disk controllers proved particularly difficult to get right. RSTS/E and RSX11M on the PDP-11, and NetBSD on the VAX, had timing dependencies in their MSCP drivers (and different timing dependencies at that). As of this writing, there is still a timing problem in the MSCP driver for OpenBSD/VAX, although not for VMS, Ultrix, or BSD 4.3.

In all of these cases, the debug vehicle was an operating system. Operating systems are great at finding bugs in simulators, but their reporting mechanisms—hang, crash, or loop—leave something to be desired. Hence the need for powerful debug tools in the simulator. The debug tools are not there to debug software but to debug the simulator itself.

KEY ISSUES

Any form of virtual machine must deal with three key issues: accuracy, software dependencies, and performance.

ACCURACY

For SIMH, accuracy of simulation has been a key requirement. This stems from a simple observation: the more accurate the simulation, the more software it will run correctly. The critical choices in implementation are level of simulation (behavioral, register-transfer); detail of simulation (instruction-accurate, cycle-accurate); and specificity of implementation (architectural accuracy, specific model accuracy).

Level of simulation. Most simulators are behavioral: they reproduce the behavior of a computer system rather than its internal implementation. Though there are likely to be similarities (for example, register files abstracted as arrays), these are coincidental. No attempt is made to reproduce the "blow-by-blow" internal operation of the system. For example, a typical 1960s minicomputer would access memory through specific buffer registers:

```
MAR <- memory_address;
MBR <- memory_access ();
memory destination <- MBR;
```

A behavioral simulator abstracts out the intermediate register transfers:

```
memory_destination <- memory_access (memory address);
```

In contrast, a register-transfer-level simulator does include the details of the hardware intermediate steps.

SIMH is a behavioral system. Even so, most SIMH simulators follow the hardware flows with great precision and model system structure accurately.

Detail of simulation. The old saw that "the devil is in the details" is particularly applicable to simulators. Processor and peripheral behavior needs to be reproduced at a very fine level of detail, or software will fail to run. Some examples:

1. Every published manual on HP's 16-bit computer systems (the 2100 family, later renamed the 1000 family) states that the SFS and SFC instructions do not implement the "clear flag" option. The schematics, however, clearly show that these instructions do implement the option; HP's RTE-IV operating system depends on this undocumented feature.

2. PDP-11 interrupts were supposed to follow a well-defined model, but as more and more peripherals were implemented, deviations from the model arose as a result of "improvements" or optimizations. The idiosyncratic interrupt behavior of the PDP-11's Ethernet and Massbus controllers—which will not disable a pending interrupt even if the interrupt enable flag is cleared—must be reproduced "bug-for-bug," or PDP-11 Unix will not run.

3. The Honeywell 316 simulator specifies only a small number of the possible bit combinations in the "generic A" operate instruction class. To reproduce the undefined but not unpredictable behavior of the other bit combinations, the simulator must reproduce the decode logic since the generic A operates signal-by-signal.

In this last case, the simulator predicted different results from the only published article on the subject; and testing on the real hardware proved the simulator to be correct.

Specificity of simulation. Finally, a simulator must reproduce a specific and complete system, rather than an idealized architectural model. This is clear enough for early computer systems, which predate the concept of a compatible computer family, but it is equally true for well-specified systems. SIMH doesn't simulate "the PDP-11" or "the VAX"; it simulates the PDP-11/73 and the MicroVAX 3900. Deviations can be perilous:

- The PDP-11 architecture did not specify the ordering of decoding and fetching operands in a double-operand instruction, and different PDP-11s implemented different orderings. PDP-11 software is not supposed to depend on this ordering. A PDP-11 simulator implementing a specific model, however, must implement double-operand processing in the exact right order; otherwise, the "model identification" software in a major operating system (RSX11M+) will not get the right answer.
- Different models are needed to accommodate different software environments; for example, early PDP-11 Unix systems run only on the PDP-11/45 and must be invoked by specific controls as alternatives to the main simulation environment.

REPRODUCING THE MACHINE ENVIRONMENT

SIMH has explicit mechanisms for handling software dependencies on the real machine environment, such as I/O formats, I/O timing dependencies, and real-world timing dependencies. I've already described handling of I/O formats and asynchronous operations; this section focuses on real-world timing dependencies.

Wall clocks versus simulated clocks. Simulated performance varies with the speed of the host processor. If the simulator keeps track of time by counting instructions or cycles, then simulated time will run faster on fast machines and will run slower on slow ones. This makes it difficult for simulated software to keep track of "wall time."

SIMH provides calibrated timers. A periodic event (such as the ticking of the realtime clock) is initially scheduled by guesstimate. As simulation progresses, SIMH calibrates the clock against wall time and automatically adjusts the simulated delay between clock ticks to approximate realtime.

Timing loops. Some programs, particularly games, require even greater timing accuracy. For example, PDP-1 Spacewar, the world's first video game, runs in a gigantic loop that is speed-matched to the performance of a real PDP-1. If the loop runs too fast—and on modern hardware a simulated PDP-1 runs 100 times faster than the original—the spaceships will zoom across the screen and fall into the sun before the user can react.

Simulators can use the calibrated timers and the simulator-specific timing mechanism to calculate the instruction execution rate. If this is too high, the simulator can run idle loops or go to sleep to slow down simulated execution to real-world rates.

PERFORMANCE

Simulated processor and I/O performance. Simulators typically take a large number of instructions to execute one simulated instruction: up to 1,000:1 for a complex simulator such as the VAX. On modern computers, the disparity in clock rates between simulator host and simulated target (3 gigahertz for a modern PC, versus .8 megahertz for a PDP-8 or 5 megahertz for a MicroVAX II) is usually more than sufficient to overcome the adverse simulation ratio. In addition, I/O on a modern computer is much faster than on historic systems, as a result of improvements in peripherals and the copious use

of memory for caches and buffers. Taken together, the improvements can be dramatic: one SIMH user reported that the running time of a complex program build was reduced from 2.5 hours on a real MicroVAX 3100-38 to 14 minutes on the simulator.

Simulators benefit from improvements in microprocessor performance. Cache hit rates are high because the instruction stream is relatively small, and the data sets are bounded by historic memory sizes. On the other hand, branches are frequent, and branch predictability is poor. In general, simulators run best on microprocessors with high clock rates, shallow pipelines, large level-one instruction stream caches, and large level-two data stream caches.

Scale. For historic systems, simulator scale is not an issue because the scale is bounded by real system limits. Simulating a 512-megabyte PDP-11 is not an issue because real PDP-11s were limited to 4 megabytes. Simulators are also used to re-create systems of relatively recent provenance, however, such as the Hercules 370 simulator or the SIMH VAX simulator. For simulators of modern architectures, scale is certainly an issue.

In the last year, users of SIMH VAX requested greater memory and storage capacity than provided by the original target system, the MicroVAX 3900. Fortunately, the VAX architecture abstracts the details of memory and storage representation to some extent. In the VAX architecture, memory size is presented to an operating system by bootstrap code; the operating system isn't required to understand the details of the underlying memory system. Likewise, in MSCP, disk-drive size is presented to an operating system by the drive itself; the operating system is expected to handle drives of arbitrary size. These abstractions allowed expansion of memory from 64 megabytes to 512 megabytes with changes only to the simulator and the boot firmware, and expansion of drive capacity to 1 terabyte (!) with changes only to the simulator's I/O routines to handle files larger than 2 gigabytes.

AVOIDING PITFALLS

SIMH has successfully re-created machines covering a 40-year span of computing history, using a common design and control framework as well as common implementation techniques. SIMH can be readily extended to simulate additional systems if three simple rules are observed:

- Research the system to be simulated thoroughly.
- Work through the mapping of the hardware architecture to software structures before starting implementation.
- Run as much real software as possible to debug simulator operations.

With a little luck, and a lot of debugging, anyone can open another locked treasure from computing's past. Q

REFERENCES

1. Supnik, R. "Debugging Under Simulation," in *Debugging Techniques in Large Systems*, edited by R. Rustin. Prentice-Hall, 1971.
2. Supnik, B. Writing a Simulator for the SIMH System, Revised February 11, 2004, http://simh.trailing-edge.com.
3. Siewiorek, D., Bell, G., and Newell, A. "PMS Notation." Chap. 13 in *Computer Structures: Principles and Examples*. McGraw-Hill, 1982.

LOVE IT, HATE IT? LET US KNOW
feedback@acmqueue.com or www.acmqueue.com/forums

BOB SUPNIK joined Sun Microsystems via the acquisition of Nauticus Networks, where he served as chief technology officer and vice president of engineering. For the bulk of his career, he was a senior manager and technical leader at Digital Equipment Corporation, where he led the development of DEC's VAX microprocessors; started and program managed the entire Alpha engineering program (chips, systems, and software); and managed the creation of the Palo Alto Internet Exchange, the Personal Jukebox, and other innovative research products. He has a B.S. in mathematics and a B.S. in history from MIT, and an M.A. in history from Brandeis University. Supnik holds seven patents in silicon and systems architecture. He is the principal developer of SIMH, the computer history simulation project.