CHAPTER

Introduction

1

Fools ignore complexity. Pragmatists suffer it. Some can avoid it. Geniuses remove it.
—Alan Perlis, *Epigrams on Programming*, 1982

In just a few years, electronic systems have become significantly more complex. Now, even comparatively simple designs include multiple processors, a mixture of CPU types, digital signal processing (DSP), application-specific integrated circuits (ASICs), field-programmable gate arrays (FPGAs), and other devices. Complementing the diverse combinations of hardware, today's systems employ a variety of operating systems and application stacks that until recently would not have been combined within a single product or solution.

Unfortunately, however, as these systems have grown in complexity, the development tools and processes that were refined when single processors and basic client−server architectures were the rule have not kept pace. As a result, today's system developers are challenged to find new ways to define system architectures, develop and integrate millions of lines of code, and deploy such complex systems. They must do this in ways that reduce risk and shorten the schedule while simultaneously resulting in a higher-quality product that is easier to support and maintain.

In addition to the growing complexity, the market also expects new systems to be delivered at a much higher pace. The product development lifecycle of most electronic systems has been significantly shortened over the last decade. Thus, today's system developers are faced with two significant challenges: deliver new solutions faster, and develop, debug, and maintain ever more complex systems. Virtual platforms can help in addressing these two challenges.

The goal of this book is to inspire and educate the reader to find new ways to leverage the power of virtual platforms and full system simulation to improve their systems' design and development activities. With this book we seek to share our experience, gathered over more than a decade, from working with our customers to help them realize the advantages of working in a simulation environment. This book is focused on virtual platforms created in Wind River Simics[†], and although Simics offers many unique features, many of the techniques and challenges discussed apply to other virtual platform solutions as well.

At one level the book will address how to use Simics simulations to achieve your development goals as a leader of an organization. At another level, the book will discuss how to use Simics simulations to get actual tasks done. The book offers best practices along with real-life examples to help you understand how to get the most out of your Simics implementation. Design patterns and architectures that have been proven to work when building complex simulation systems involving many separate components are described. While the book is not intended to be a user manual, it is a comprehensive book on simulation using Simics, and we have tried to provide enough details for the book to be useful for someone trying to implement the concepts described.

This chapter introduces the reader to why virtual platforms and full-system simulation like Simics is a critical tool for developing today's complex computer-based systems. The chapter defines the basic terminology and provides a high-level overview of why and where Simics is being applied to solve problems for software and system developers. The chapter concludes with an outline of the remaining chapters of the book.

VIRTUAL PLATFORMS

A *virtual platform* is a model of a hardware system that can run the same software as the hardware it models. The virtual platform is simulated on a *host* computer that may be different from the hardware modeled by the virtual platform. For example, a big-endian Power Architecture system with a controller area network (CAN) bus and other peripherals running VxWorks† can be simulated on a typical little-endian Intel® Architecture PC running a Linux† or Windows† operating system. A virtual platform is not limited to modeling a single processor or board, but can represent anything from a basic board with only a processor and memory to a complete system made up of network-connected boards, chassis, racks, and models of physical systems.

The key property of a virtual platform is its ability to run unmodified binaries of the software that will finally run on the real system, and run it fast enough to be useful for software developers. Such software includes low-level firmware and boot loaders, hypervisors, operating systems, drivers, middleware, and applications. Therefore, the virtual platform accurately models the aspects of the real system that are relevant for software, such as CPU instruction sets, device registers, memory maps, interrupts, and the functionality of the different devices. On the other hand, the virtual platform is typically not concerned with modeling the detailed implementation of the hardware, such as internal buses, clocks, pipelines, and caches.

By focusing the model on the hardware−software interface and functionality it is possible to achieve good performance and produce a virtual platform very early in the product lifecycle—two critical features required to address the aforementioned challenges.

TERMINOLOGY

There are many terms in use for the kind of technology that Simics represents. This section defines some of the terminology the reader may come in contact with.

Simulation is a very broad term, used in many different fields. At its core, it means that you use computer software to build a *model* of some phenomenon you want to study and then run this simulator to understand the behavior of the modeled system. A simulation provides more flexibility than the real system, allows parameters to be set freely, provides better insight into the internal workings, and allows for the replay and repetition of scenarios. It also fundamentally avoids the need to build physical prototypes or experiments, which speeds up development. Simulation is used in every field of science and engineering. Simulations are used to predict weather, crash-test cars, design aircraft, understand economic mechanisms, and find new medicines. This book is primarily concerned with the simulation of a digital computer system (the target) using another digital computer system (the host).

Full-system simulation (FSS) is a term commonly used to describe Simics, and it captures the fact that the simulation targets an entire target system. Originally, the point of a full system was that the digital computer hardware model was sufficiently complete to run a real operating system (Magnusson et al., 1998). Over time, it has grown in scope, and today a full system often includes factors external to the digital computer hardware, such as models of the surrounding world and inputs and outputs from the outside. It also includes the use of the simulator to model collections of digital computer systems, such as multiple machines in a network or multiple boards in a rack. A simulation that cannot simulate more than a single system-on-chip (SoC) or board is not really a FSS today.

Virtual platform is the established term in the world of electronic design automation (EDA) for a piece of software that works like a piece of hardware and is capable of running software in lieu of the real hardware. Virtual platforms are used at many levels of abstraction, from cycle-accurate models that correctly emulate all pins and signals on buses and inside devices, to programmer's view (PV) and transaction-level models (TLMs) that essentially work like Simics does. Virtual platforms are considered to be development tools.

Emulation is a term commonly used to indicate a software layer that lets a piece of software run on a platform it was not initially targeted to run on. Well-known examples are the Mac† 68k emulator that Apple† used in the migration from the 68k-family of processors to the PowerPC† family, and the Rosetta emulator that allowed PowerPC binaries to run on Intel® Architecture in Apple's next architectural transition. Simulators for old videogame platforms, such as the Nintendo† Entertainment System (NES), are also known as emulators to the public. We thus consider emulation in the software realm to mean something that runs software by translating binaries and operating system calls, where the primary use is to run software, not to develop it.

Virtualization in the IT world means the use of virtual machines to run multiple software loads on a single host. Virtualization as a principle traces its beginnings back to the IBM System/360 line in the 1970s, and today there is a wealth of virtualization solutions available on standard Intel hardware such as KVM, VMware†, Xen, Hyper-V, Virtualbox, and many others. A virtual machine runs a real operating system, but often employs special drivers and input/output (I/O) mechanisms to optimize performance for disks and networking. The goal is to provide an isolated and manageable container for a particular workload. A key property of virtualization is that it provides virtual clones of the underlying host machine—a virtualization system cannot provide a target system that is fundamentally different from the host.

In EDA some of these terms have specific meanings. An *emulator* is a custom hardware system that runs the register-transfer level (RTL) of a new design without having to actually manufacture a chip. Emulators are optimized for execution speed, even if they also typically support some development. A *simulator* is a software program that simulates the RTL. This is very slow, but it also does not require any special hardware, and it provides very detailed insight into the execution of the system. For understanding and debugging a hardware design, a simulator is the gold standard. A *field-programmable gate array prototype* synthesizes the hardware design to run on an FPGA, rather than for ASIC production. The functionality is the same, but the detailed timing behavior is not. Still, it is much cheaper than using an emulator and runs much faster than a simulator. If seen in software terms, this is the equivalent of using the same source code, but compiling it for a different architecture and operating system.

SIMULATION AND THE SYSTEM DEVELOPMENT LIFECYCLE

Full-system simulation can be applied during the complete system development lifecycle as shown in Figure 1.1. It helps in designing and defining systems by providing an executable model of the hardware interface and hardware setup. FSS supports hardware and software architecture work, and it validates that the hardware can be efficiently used from the software stack. Full-system simulation is

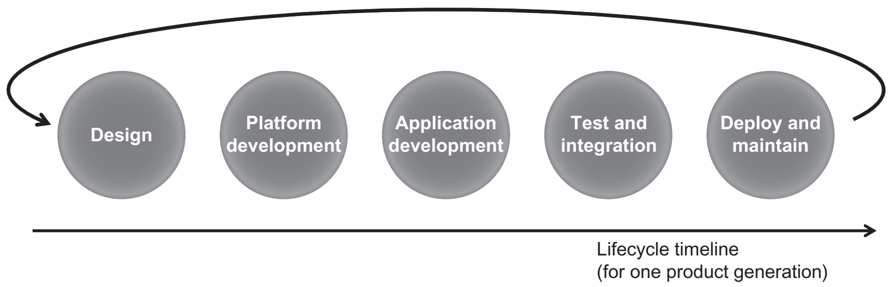

FIGURE 1.1

System development lifecycle.

used to develop low-level firmware, system software, and application-level software. Testing and integration can be performed on the simulator as well as on hardware, providing increased hardware flexibility and developer agility. The software development schedule can be decoupled from the availability of hardware. Using a simulator improves software development productivity by providing a better environment than hardware, especially for reproducing issues, debugging, and automated testing and execution.

The following sections describe various ways in which virtual platforms are being used to make developers more efficient throughout the product lifecycle.

HARDWARE DEVELOPMENT AND DESIGN

A virtual platform is a common tool in the design of new computer systems and new SoC designs. Early hardware design models tend to focus on performance modeling without much care for the actual functionality and what is being computed, which is not really a good match for the Simics-style fast functional simulation. Still, Simics-style virtual platforms are very useful during the hardware design, because Simics provides a means to define and test the functional design of the hardware system. It feeds into pre-silicon software development, as discussed in the next section.

It is also quite common to use fast virtual platforms with a few components swapped out for detailed cycle-accurate and bit-accurate models to perform component-level tests with real workloads and component-level verification and validation work. Chapter 9 discusses how such mixed-level simulations can be built by combining elements from multiple different simulation systems.

PRE-SILICON

When developing a new chip, FSSs like Simics are used to develop software long before the first silicon appears. This allows the entire project to have its schedule “shift left,” effectively reducing the time to market and time to revenue for a new product. In the traditional product development flow, hardware development, software development, and integration and testing more or less take place serially. Typically, software developers try to start as early as possible by using different techniques such as cross-compilation to the host machine, working with old revisions of a board, or using previous-generation hardware. These techniques offer significant challenges, especially for low-level code such as firmware and drivers. Using virtual platforms, the software and hardware can be developed more or less in parallel, significantly reducing the time to a releasable product. Additionally, because the schedule pressure is reduced by increased parallelism, there is the option to get more testing done before release, increasing product quality. These benefits from a “shift left” are illustrated in Figure 1.2.

It has been shown many times that by using virtual platforms the time to create a board support package (BSP) for a new design can be pulled in from several

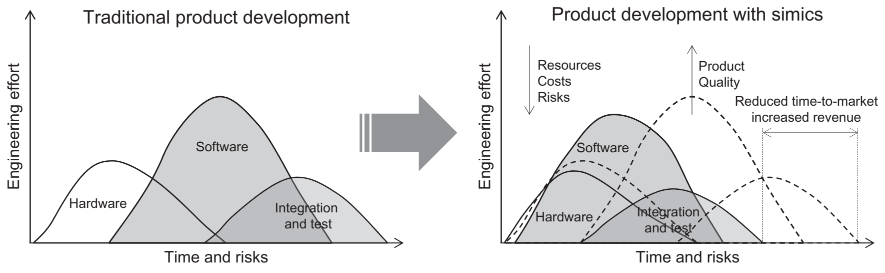

FIGURE 1.2

Product “shift left”.

months to only days after the first hardware is available. In the ideal case, the hardware and software teams work closely together, allowing the software team to provide feedback to the hardware designers already before the design is frozen. This can help to avoid costly mistakes in terms of overly complex programming models and performance bottlenecks that appear because of a lack of system optimization.

The software most commonly developed on the pre-silicon virtual platform are boot loaders and basic input/output systems (BIOSs) (Carbonari, 2013), silicon verification and test software (Veseliy and Ayers, 2013), drivers, firmware, and operating system support. Even though the Simics abstraction level hides the detailed timing and implementation of a system, developing software on a functional virtual platform has been proven to work very well. Compared to not using a virtual platform, system developers save weeks and months of time (Koerner et al., 2009).

A variant of pre-silicon development that might not be obvious is the development of software for a new *board*. Even if a new board is based on a familiar SoC and existing network chips, memory, and other functions, a Simics model can still be provided ahead of the arrival of the board and offer the same benefits as for a new silicon chip. Just like a new chip, a new board needs custom boot code and drivers to enable software to use the capabilities of the board.

PLATFORM DEVELOPMENT

Platform development refers to the development of the fundamental software that makes hardware work and that provides a platform for application development. As discussed before, this includes the development of firmware, boot loaders, and BIOS, as well as operating system kernels and BSPs. In addition to such hardware-interface code, it also usually involves integrating various forms of middleware software on top of the operating system. The middleware provides the crucial domain-specific specialization of the generic operating system platform, such as distributed communications systems, fault-tolerance mechanisms, load balancing, databases, and virtual machines for Java, C#, and other languages. The complete software stack can be developed and run on Simics.

Debugging low-level code in Simics is a much nicer experience than using hardware, especially compared to early unstable prototype hardware. As discussed in depth in Chapter 3, Simics enables the debugging of firmware and boot code from the first instruction after power on, and makes it easy to debug device drivers and interrupt handlers. When drivers and the operating system are up, Simics can be used to integrate middleware and services on top of the operating system, taking the setup all the way to a complete running platform, ready for application developers (Tian, 2013).

In larger organizations, there is usually a dedicated platform team who is responsible for developing and delivering ready-to-use integrated platforms for application developers. Virtual platforms can be used to efficiently deliver the platform to application developers, containing both hardware and software, booted, configured, and ready to go. With a virtual platform, a nightly build can become a nightly boot, using checkpoints as discussed in Chapter 3 to deliver a ready-to-use platform to the application development teams.

APPLICATION DEVELOPMENT

Applications provide the software that makes a system useful for its end users. An application can be a single standalone process like a traditional desktop application. More often, an application actually consists of multiple cooperating processes, running on the same machine or spread out across machines to form a distributed application. In the embedded world, there is often an element of hardware involved, interfacing to the world outside of the computer. Fault-tolerant applications containing multiple redundant software and hardware systems are also commonly seen in the embedded world.

Application development with Simics means giving application developers access to virtual hardware, which lets them test their code on the same platform the code will run on in the end. Often, application software development is performed using development boards that only partially match the target system, or by using some form of emulation layer compiled to the host. With Simics, target hardware availability is not an issue, and application developers can work on their standard PCs while still compiling their code for the actual target and running it as part of the real software stack. Simics can simulate networks of machines and the interface between computers and their environment to provide a realistic system for application developers.

As the time available for development gets shorter and continuous integration and continuous deployment are being applied even to traditionally slow-moving embedded systems, the value of working with the actual target hardware increases. The goal is to have every build of the software ready to deploy to customers, and this means that it has to be built with the actual release compilers and get tested on the hardware that is used in the field. This is a very good match for virtual platforms, because they can be built and configured to precisely match the real-world platforms, enabling fast and agile software development while still only using standard laptops, workstations, and servers.

Application development can be supported by various simulation-powered shortcuts to make the work more efficient, such as using back doors to load software and scripts to automate a load-configure-run cycle.

For applications built on top of distributed, parallel, and virtualization-based systems, Simics provides an ideal debug and test platform, because it offers the ability to control and debug the entire system and all parts of the application using a single debugger, regardless of whether the actual system has debug access built into its hardware or software.

DEBUGGING

While not really a part of the product lifecycle, debugging is one of the most time-consuming parts of software development. Even worse, a really bad bug can potentially hold up a release, and cause customer pain, manufacturer embarrassment in case they are discovered post-release, and even force a recall of a product.

Software debugging involves three fundamental activities: provoking the bug, finding and isolating the bug, and fixing the bug. Traditionally, successful debugging requires a high degree of developer skill and experience, often combined with patience and luck. Simics removes luck from the equation by simplifying efforts to repeat and isolate the bug. Several of Wind River's customers previously struggled for months to repeat and isolate bugs on physical hardware only to find them in hours with Simics.

Simics's usage and value as a debugger applies to post-silicon as well as pre-silicon use cases (Tian, 2013). When hardware is available, Simics complements the use of hardware for debugging. Users who test-run their code on Simics can easily debug it using Simics, and Simics can also be used to replicate and debug issues from the field and tricky hard-to-find bugs.

To repeat a bug on physical hardware, developers may have to restart the system or application hundreds or thousands of times, using a new set of input parameters, data streams, or operator actions each time, or hoping for some random fluctuation that will provoke the bug. Simics virtual platforms are different. They operate in a virtual world where the entire system state and all inputs are controllable and recordable. As a result, any simulation can be trivially reproduced. Once a bug is seen inside a Simics simulation, it can be reproduced any number of times at any time or any place in the world. Thus, Simics makes it possible to transport bugs with guaranteed replication.

Once a bug can be reliably repeated, the developer must find the source of the bug. Traditional hardware-centric debug methods require an iterative approach where breakpoints are set, the system is run, registers are reviewed, and the application is restarted or execution is resumed to the next breakpoint. Using this technique, developers can eventually find the precise offending lines of source code. However, attaching a debugger to a hardware system will affect the execution of the system, leading to so-called *Heisenbugs*, whereby the act of debugging changes the observed system and makes the bug disappear. In particular, stopping individual threads or putting in breakpoints will often cause a complex software system to break entirely. In contrast, a simulation-based debugger is nonintrusive, and the system will run exactly the same regardless of whether it is under the inspection of a debugger or not.

With Simics, developers can run the system in reverse, watching the sequence of steps that led up to an issue. Simics will trigger breakpoints in reverse, making it possible to stop at the previous change to a variable or memory location. Such an approach does away with the need to start the debug session over and over again and try to reproduce a bug and plant different sets of breakpoints. Instead, Simics allows debuggers to continue from finding the bug directly to debugging and unearthing the cause of it. Simics can observe all parts of the system state and trace all interactions without disturbing the target execution, which means that it is easy to understand just what the system is doing.

Once a bug has been repeated and isolated, the effort to resolve it may range from trivial to extensive. With Simics, developers may apply standard features such as checkpointing, reverse execution, run-to-run repeatability, and full-system visibility and control while finding the precise bug fix. For complex systems, Simics will make it easier to replicate the particular hardware–software setup involved with a bug report to test fixed code in a relevant environment.

TESTING AND INTEGRATION

Testing and integration are crucial parts of any large-scale software development project. Modules from many sources have to be built, integrated, and tested to make sure they are working together. Hardware has to be integrated with software, and networks and racks configured, brought up, and tested. Using a simulator like Simics for this phase brings great benefits to the development workflow (Magnusson, 2005). As discussed in more detail in Chapter 5, Simics can scale up to truly large systems, making system testing and integration work in simulation a realistic option.

When Simics is used to enable software development before silicon or boards are available, it is natural to also perform *system integration* ahead of hardware. Because Simics models cover the whole system, all the system software and hardware can be integrated in the simulation before the hardware is available. A particularly interesting case is when the new hardware is part of a bigger system containing older hardware, such as rack-based systems where new and old boards coexist. In such cases, Simics makes it possible to virtually integrate the new hardware with the old hardware, allowing system integration and testing to happen before the hardware becomes available.

Creating and managing multiple system and network configurations for testing is often difficult in hardware. The number of hardware lab setups is limited by hardware availability, and reconfiguring a hardware setup with different boards and network connections is time consuming and error-prone. With Simics, it is possible to write scripts and save setups as software, making configuration an

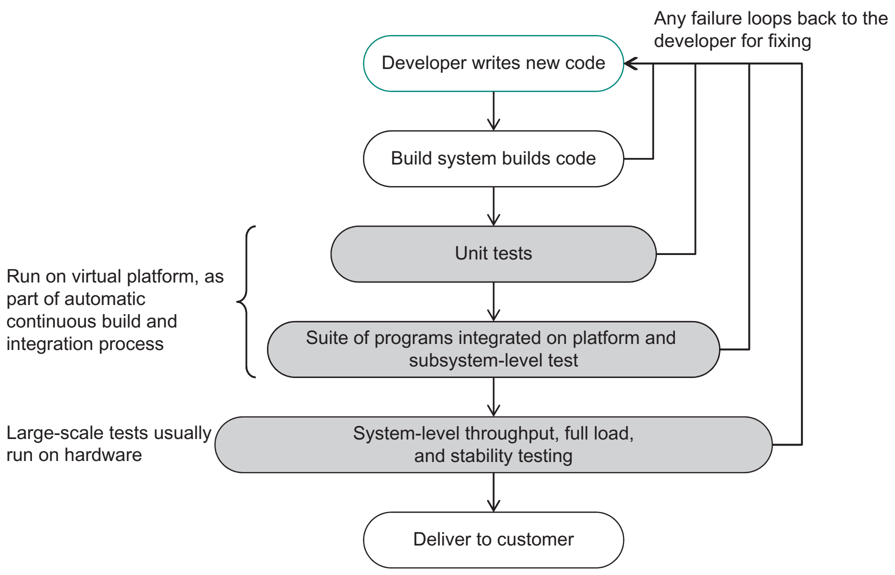
Any failure loops back to the developer for fixing
Developer writes new code
Build system builds code
Unit tests
Suite of programs integrated on platform and subsystem-level test
System-level throughput, full load, and stability testing
Deliver to customer
Run on virtual platform, as part of automatic continuous build and integration process
Large-scale tests usually run on hardware

FIGURE 1.3

Continuous integration with Simics.

instant process. Configurations can also be saved in version control systems, allowing hardware and software configurations to be managed together.

Testing can naturally be performed in *parallel*, because virtual platform availability is only limited by the number of servers that can be used to run Simics. This increases the amount of testing that can be performed within a given time, compared to only using hardware setups. Using techniques like *checkpointing*, it is possible to shorten test execution time by starting from booted setups rather than rebooting the test system for each test.

Simics and simulation are enablers for *continuous integration* (Duvall et al., 2007) and automated testing of embedded code. Using hardware is much more difficult than simulators, especially for quick short tests. As illustrated in Figure 1.3, a typical continuous integration workflow starts with a developer submitting new code to the build system. If the build fails, they have to fix it. Once the code actually builds, quick unit tests and other smoke tests are typically run to make sure the code is not totally broken. Such tests should run very fast—no more than a few minutes—to quickly return a reply to the developer.

Once code passes unit testing, it can be subjected to larger-scale tests. First, some form of subsystem test is run where the code is tested in a real context but typically with quite small inputs. The goal is to get the subsystem-level tests done in hours. Code that passes subsystem tests is finally used in system-level tests where it is run along with all other code and functionality of the system, and subjected to long hard tests under high load and lots of traffic.

Simics is a suitable platform for unit tests and subsystem tests, but system-level tests are usually run on hardware. At some point, it is necessary to test what is actually going to be shipped. The maxim is always to "test what you ship and ship what you test." Thus, the physical hardware that will be shipped to the customer must be used for final testing.

Still, using a virtual platform like Simics can drastically reduce the amount of hardware labs needed. If most unit tests and subsystem tests are run on Simics, most developers will be independent of hardware and can run the tests whenever needed, regardless of hardware availability. It is very easy to integrate Simics as an automated testing component in build automation systems like Jenkins. Integration also covers the integration of a computer system with its physical environments. By combining Simics with other simulators, as discussed in Chapter 9, simulation-based testing can cover both a control computer and its environment.

DEPLOYMENT

The pre-silicon use case is easy to appreciate—when there is no hardware available a simulator is a good solution and often the only solution. However, many Simics users find that the benefits of virtual platforms carry on long into the deployment and maintenance phases. For example, some customers embed Simics into their complete virtual environment, allowing them to carry out system-level development and testing in a flexible and powerful environment, at the point in time where systems are actually available in the market and deployed to customers.

In the deployment phase Simics can be used to perform demos for customers. It is easy to bring a fully configurable Simics model to a customer to showcase an application that would otherwise require custom hardware or a large set of hardware to be brought to the customer or maintained in a separate demo lab with obvious resource limitations. A related topic is that of training, which is covered in more detail later in this chapter.

Virtual platforms can also be used to simulate faults that have appeared in deployed systems. For example, if the target system is flying through space somewhere, a virtual platform on Earth can be used to model various faults that have appeared in the physical system during its service life. Software workarounds and patches for hardware issues can then be tested on the ground, in the simulated environment of the virtual platform, before being uploaded to the live system.

MAINTENANCE

Once development is complete and a product version is released, it goes into maintenance. In maintenance the focus is typically on providing incremental improvements and to resolve bugs that were not found during QA testing. The test automation systems discussed previously for testing and integration should still be used to make sure no new errors creep back into the system.

When issues come back from the field, as they invariably will, virtual platforms support the reproduction and analysis of the issues. With a virtual platform, it is possible to reproduce a customer's setup even if the precise hardware needed is not available in physical form. Once a bug has been reproduced on the virtual hardware, it can then be analyzed at leisure.

REAL-WORLD STORY: DEBUGGING A CORRUPTED FILE SYSTEM

One Simics customer had a system that handled large amounts of network data. Every once in a while the system would crash with a corrupted file system. The crash happened in a catastrophic way so the customer was not able to recover any useful information about the crash from the logs on the hard drive. Because this system was deployed in a situation where downtime was very costly, the customer was desperately looking for a solution.

After months of debugging the problem using real hardware, the customer decided to try a different approach. There was already a Simics model available for the system that had been used during development, so the customer cloned the setup of the system in Simics and began trying to reproduce the bug. They realized that the bug was most often triggered when the hard drive was close to full, so they replicated this scenario and started running traffic through the Simics model. By varying timing parameters and traffic data they eventually managed to reproduce the bug in Simics. Because Simics is deterministic they could now reproduce the bug at their leisure.

The next step was to write a script that checked the consistency of the file system automatically. Using this script and checkpoints the customer bisected the workload to pinpoint the time when the file system was corrupted. Attaching the debugger they found that a routine in the operating system was corrupting the stack and overwriting the return address. This was a commercial real-time operating system (RTOS) and the customer did not have access to source code. However, they could go to the OS vendor and pinpoint exactly the routine that was causing the issue. The problem was then identified and fixed by the OS vendor.

Another aspect of maintenance is generational change: once a system is delivered and deployed, it is often the basis for tweaks and upgrades. Components can be upgraded to increase capacity or fix hardware issues, and the virtual platform used to develop the original software can easily be updated to model the updated hardware, enabling another round through the lifecycle.

A virtual platform for a deployed system is often also used as the basis for the development of a next-generation system, especially for SoC, chip designs, and system designs. A next-generation platform can be developed by starting with a model of the current platform and then changing one component at a time from old to new as they become available. The software is then updated to work with the new hardware one component at a time, always maintaining a working hardware–software system that is gradually changing from all-old to all-new. Such a gradual change from one generation to another is very hard to do in hardware, because there is no practical way to build a series of part-way designs (Magnusson et al., 2002).

TRAINING

Virtual platforms can be used for training on the system being simulated. The main benefit of using a virtual platform is that training can be performed without the need to access the real hardware system. Providing large classes with sufficient hardware is often prohibitively expensive. The virtual platform runs the actual software, which means that the behaviors seen are just like the real thing.

For the case of certified software, such as avionics, using exactly the same binary ensures that the training setup can be updated and kept in sync with the real system. In the past, simulators for systems containing avionics systems often relied on porting the software to run on a standard machine, or simply building a behaviorally correct emulation of system components. With increasing system complexity and reliance on software to implement system functionality, these approaches tend to fail. Keeping software ports or behavioral emulators up-to-date with the latest released real-world software is an extra cost and schedule risk, which can be entirely avoided by running the real binary on a virtual platform.

Virtual platforms can also be used in lieu of real systems to simplify training in general concepts. Simics has been used to teach networking, multicore programming, and operating systems at Wind River (Guenzel, 2013). In academia, one particularly interesting area is teaching operating system concepts. With a simulator at the base, it is much easier to debug and understand the behavior of an operating system, enabling much more advanced labs than would be possible on hardware. Once the operating system is up and running on the simulator, it can be moved to real hardware and tested to show the students that what they did actually works in the real world (Blum et al., 2013).

REAL-WORLD STORY: TEACHING OPERATING SYSTEM WITHOUT SIMICS

In the mid-1990s, before Simics was available, one of the authors of this book was a teaching assistant on a course in operating systems. The course was taught using a set of MIPS R3000-based boards with a small amount of memory and standard serial port for output. Getting an operating system up and running was not entirely easy, and most students ended up pulling a few all-night hack sessions in the computer lab to get their software to work. In the end, most of them did.

However, one very ambitious group of students decided that they would try to make use of the memory management unit (MMU) of the MIPS processor. After all, a real operating system should have memory protection. This turned out to be very hard indeed—setting up the translation look-aside buffer (TLB) entries and moving to a protected memory model is nontrivial. Test run after test run was made, with debug printouts scrolling by on the 24-line terminals in use, each time ending in a complete freeze of the target system. With no hardware debugger available and very limited scroll-back on the terminal, debugging was basically guesswork.

In the end, the students were forced to give up. Had they had Simics around, debugging would probably have been trivial. Check the MMU state, trace the MMU setup operations, and check where the code froze. Still, the students were given passing grades for the course and went on to quite illustrious careers in programming. The lab assistant later worked as an associate professor in the department and introduced Simics as the basis for the operating systems lab. It did make it much easier to debug the students' OS kernels.

LONGEVITY SUPPORT

Full-system simulation has been proven to have tremendous value in the support of *really old hardware*. Indeed, once a system gets old enough, the value of having a simulator for it tends to go up, as the hardware starts getting scarce.

In telecom and other fields, hardware sold a decade ago is often still used in field installations—customers do not upgrade their hardware unless they absolutely have to, and hardware tends to last longer than expected (or planned). Such hardware tends to be built from racks containing lots of boards, and there are usually a large variety of boards with several generations of each board. With a virtual model of the older boards in place, all developers can have their own immediately accessible hardware to work on. In the physical world, these older boards are often in very limited supply, limiting user productivity.

The practice of extending the life of older systems by software upgrades to old hardware is common in the military, aerospace, and transportation fields. The development of a software upgrade requires development hardware, but typically there are very few or no physical units available. Development boards tend to go bad and become unstable or useless over time. Even if a large supply of boards were procured at project start, their half-life tends to be only a few years, and after a decade or two it is rare to have many development boards available at all. Taking electronics units from systems in the field is not a realistic option due to hardware cost, the fact that making systems unavailable is often unacceptable, and that the boards being used in production are not exactly being designed for development tasks.

A virtual platform is thus a very nice solution that provides developers with the rare luxury, in these fields, of ample hardware access. A virtual platform is also stable and available virtually forever, as illustrated in Figure 1.4. The virtual platform will be available as long as Simics continues to be ported to new generations of hosts. And as long as the virtual platform is available, there is the ability to run the decades-old software stack and to test and integrate new software.

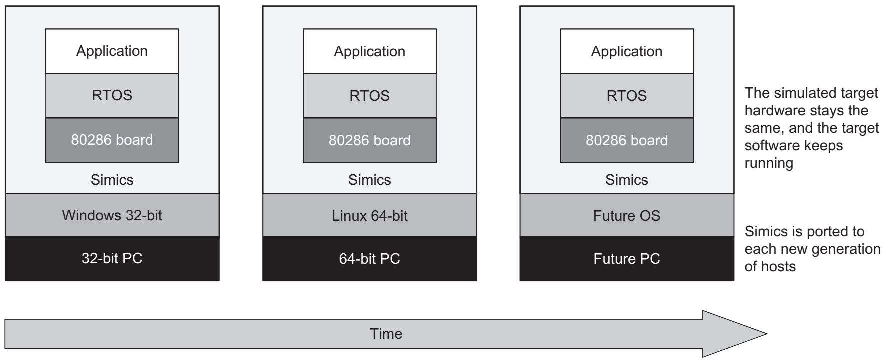

FIGURE 1.4

Virtually infinite platform life.

CERTIFIABLE AND SAFETY-CRITICAL SYSTEMS

Simics is commonly used to help develop certifiable and safety-critical systems. While Simics is not a qualified tool, it can still add tremendous value to the development of such systems, across all the lifecycle phases.

In the aerospace world, Simics is not usually used to actually test software for certification credit directly, but instead it is used to debug and develop the certification tests. By making sure that the certification tests are solid before they are run on hardware, significant effort and schedule time can be saved.

Certified hardware is also usually both expensive and rare, and using Simics to augment hardware availability can remove many hardware-dictated bottlenecks from the development process. For example, with Simics, it is possible to run automated tests in parallel on regular servers, rather than relying on particular hardware. This can enable daily regression testing instead of weekly, reducing the chance of bugs sneaking back into the code base. Certified hardware and software stacks also tend to have poor debug support, because back doors are not a good thing on a critical system. Using a Simics model along with the unintrusive Simics debugger makes debugging much easier.

Safety-critical systems also tend to contain error-handling code. Testing error handlers is about the hardest thing possible, because forcing errors on hardware is very difficult. With a simulator like Simics, fault injection is much simpler, allowing for testing, debugging, and validation of error handlers. In industrial systems, validating fault-handling code is a requirement, and using a simulator like Simics makes it much easier to systematically inject particular states in the system directly into the virtual hardware. The alternative method of using a debugger to control the target system and overwrite values is much more intrusive.

REAL-WORLD STORY: NASA GO-SIM

The NASA IV&V Independent Test Capability (ITC) team joined forces with NASA Goddard Space Flight Center (GSFC) to develop a software-only simulator for the Global Precipitation Measurement (GPM) Operational Simulator (GO-SIM) project. The GPM mission is an international network of satellites providing next-generation global observations of rain and snow. GO-SIM includes the GPM ground system and database, flight software executables, and spacecraft simulators.

GO-SIM was designed as a high-fidelity simulator with no hardware dependencies. Its functions include loading and running unmodified flight software binaries, executing flight scripts, performing single-step debugging, injecting errors via the ground system, stressing the system under testing, and validating findings from other analyses.

Part of GO-SIM is a Simics model of the RAD750[†] processor, which enables the target software to run on the virtual platform the same way it does on physical hardware. Along with Simics' capabilities of scripting, debugging, inspection, and fault injection, it enables users to define, develop, and integrate their systems without the constraints of physical target hardware.

Simics allowed NASA's ITC team to simulate their target hardware, ranging from a single processor to large, complex, and connected electronic systems, and build its GO-SIM product with all the desired features.

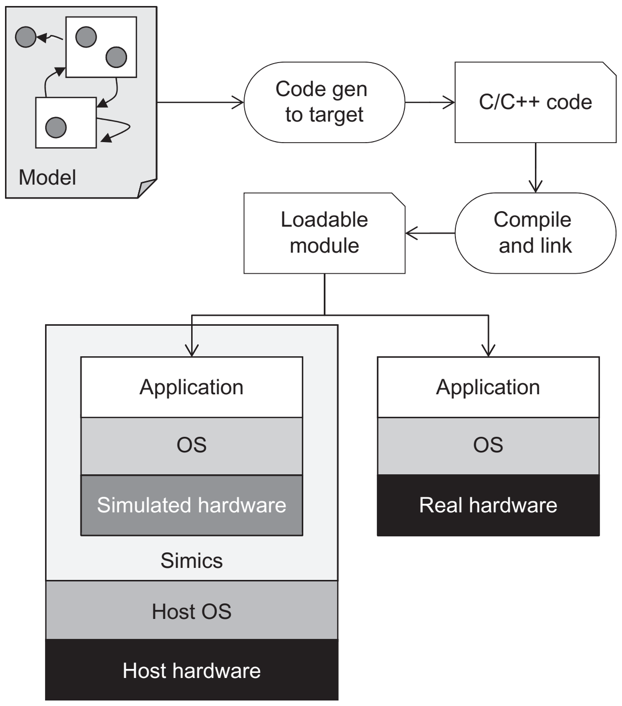

FIGURE 1.5

Simics and autogenerated code.

MODEL-DRIVEN DEVELOPMENT

Model-driven development (MDD) is widely applied in the domain of control systems and is the standard development methodology for automotive, aerospace, avionics, and defense systems. A key part of MDD is to generate code from the model, as illustrated in Figure 1.5, rather than writing it manually. For Simics, whether code is generated or handwritten does not matter—it will run the same on Simics.

PROCESSOR-IN-THE-LOOP TESTING

In a typical model-driven workflow, a model is first tested using model-in-the-loop (MIL) testing within the modeling tool (e.g., Simulink, Labview, MATLAB, or SCADE). In MIL testing, the model of the code to be generated is tested against a model of the world it interacts with. Next, simulation-in-the-loop (SIL) testing is performed, where code is generated to run on the development host, testing the code part against the world model. This makes sure that code generation from the model works. Once this is proven, processor-in-the-loop (PIL) testing is performed where the code is generated to the target system and tested using the actual processor it will run on in the final system. PIL testing makes sure that code generation and compilation for the target system does not introduce any errors, such as those involving word length, endianness, floating-point implementation details, or other target properties. In PIL testing, the model of the world is still used as the counterpart to the code.

PIL testing makes sense to do on Simics, because Simics provides a model of the final target system and thus a model of the final target processor. With Simics used as a PIL target, it is possible to automate the execution of tests from within the model-driven tool (see Chapter 9 for more information on such integration work) and to provide easy and widespread access to target hardware. The alternative to using Simics is to use a development board or even the final hardware, which is always a logistical and practical issue.

HARDWARE-IN-THE-LOOP TESTING

After a control program is proven in PIL testing, it is time to test it for real. This is done by generating code for the real target and running the code on the real target with the actual physical system being controlled instead of the world model used for MIL, SIL, and PIL testing.

Simics can be used for hardware-in-the-loop (HIL) testing in the same way that a physical board can. This requires that the Simics model is connected to the outside world using some form of real-world connection from a connection on the simulated board to a connection on the host machine. As discussed in Chapter 5, Simics provides such connections for a variety of common buses and networks.

INTEGRATION TESTING

The classic MDD flow does not really touch on the issue of system integration. The focus is on generating correct code for the core functionality of the system. That code will need an operating system and a hardware platform to run in the real world, and integration with that platform often happens quite late in the system design cycle. Indeed, even HIL testing is often performed using development hardware rather than the final system.

Simics can be used to move integration earlier and allow earlier testing of the integration. As shown in Figure 1.6, with a Simics model of the target system, the OS port to the target hardware and the integration of the operating system and applications can be tested without hardware.

Something that Simics makes possible is to test that the application works with the actual input and output as provided by the operating system and target hardware platform, while still using a model of the world. Thus, it is possible to create a fully virtual integration environment, where hardware, the operating system, the applications containing control algorithms, and the world can all be run together in simulation.

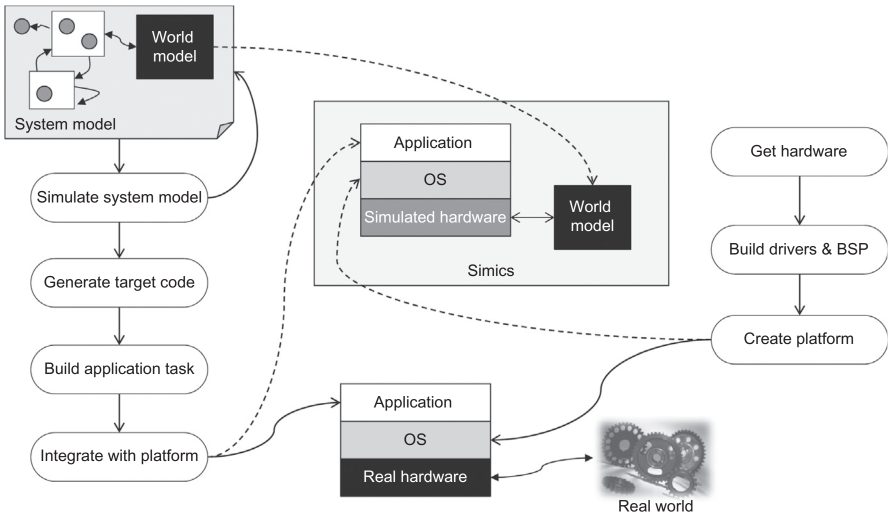

FIGURE 1.6

Model integration testing with Simics.

BOOK OUTLINE

Chapter 2 defines the basic terminology used throughout the book and introduces the reader to the Simics architecture, design, interface, and execution model. It describes how Simics works and why it works the way it does.

The core values of a fast virtual platform—developing, debugging, analyzing, and testing software—are covered in Chapter 3. A virtual platform like Simics lets users run software of all kinds, targeting all kinds of hardware, on a laptop or a development or test server. With the software running on Simics, the benefits of using simulation can be realized: determinism, checkpointing, reverse execution, full control, and insight. Chapter 3 describes how Simics is used to develop and debug software, including the features of the Simics system-level reversible debugger.

Simics structures a virtual platform into a hierarchical structure of reusable components with typed connectors. The components mimic the physical or logical breakdown of a system. The Simics system panel can be used to provide a visualization of a system that makes sense to the end user. Chapter 4 presents the Simics component system, the system panel, and script systems that are used to build systems from individual device and processor models.

Network simulation is an integral part of Simics, and many Simics target systems are networked in one way or another. Simics is used to simulate very large networks of systems, scaling up to several thousand target processors spread across dozens or even hundreds of networked boards. Chapter 5 shows how network simulation is done in Simics and how simulation can be scaled up to very large systems.

Chapter 6 introduces the reader to how to best perform transaction-level modeling of individual devices and how such models are built in Simics. It covers the Device Modeling Language (DML), as well as device modeling in C, C++, Python, and SystemC. Chapter 6 provides detailed step-by-step instructions for how to create a simple device model in Simics.

Following the introduction to modeling, Chapter 7 provides a tutorial-style example on how to develop a model of a direct memory access (DMA) controller, properly connect it to a virtual platform using PCI Express (PCIe), and to enable a device driver to interact with it. The example covers a wide range of important modeling concepts, such as handling, PCIe configuration and inbound/outbound accesses, interrupts, parsing of data structures, and how to model the passing of time.

Simics is designed to be an extensible and programmable system, allowing users to customize the tool to solve their particular problem in the best way possible. Over the years, Simics has been used for things and in situations that were not intended or even imagined by its developers. Chapter 8 discusses how Simics can be extended by its users, including aspects such as cache modeling and fault injection.

Simics users often need to model the physical world or look deep into the implementation of computer components. Rather than using Simics itself to create such models, it sometimes makes more sense to integrate Simics with other simulators, leaving each simulator to do what it does best. Chapter 9 addresses the reasons for, and the main issues involved in, creating such simulator integrations. The chapter provides a discussion on the general issues involved in integrating simulators and proven design patterns for such integrations.

Chapter 10 describes how Intel has used Simics for improving the system development workflow. At Intel, one of the major use cases of Simics is to help software and hardware bring-up, starting from the early pre-silicon stages. With the help of Simics, a functional model of future Intel hardware can be made available to BIOS and driver developers a year or even more ahead of engineering samples. This approach allows development of low-level software, which is very hardware-dependent, in parallel with the development of the hardware. In addition to that, close collaboration with hardware developers allows the team of Simics engineers to perform a certain amount of validation of early hardware specifications, thus speeding up the hardware development process as well. These practices lead to cost savings by reducing product time-to-market—that is, the "shift left."

TRADEMARK INFORMATION