CHAPTER

Simics fundamentals

2

We may change the name of things; but their nature and their operation on the understanding never change.
—David Hume

This chapter 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.

SIMICS† ARCHITECTURE AND TERMINOLOGY

Simics is a modular *full-system simulator* (FSS), which means that it is capable of simulating the user's entire system, while running the real software taken from the real system. The systems simulated with Simics go all the way from individual processor cores, through processors and SoCs, to boards, racks of boards, or even networks of machines, to systems of systems based on arbitrary network topologies. Because of the complexity of the models and the simulation environment it is important to clearly define some concepts and the terminology that is used throughout this book.

Simics itself runs on the simulation *host*, typically a desktop or server in the user's development environment. The host runs a host operating system, such as Microsoft† Windows† or a Linux distribution.

Inside of the Simics process, we find one or more simulation *targets*. A simulation target corresponds to a specific configuration, including software, of a *virtual platform* (VP) model. The *target hardware* is what Simics simulates, on top of which the *target software* is running. The target software typically includes a target operating system, which may differ from that running on the simulation host, and from other targets within the same simulation.

Simics is modular in the sense that it provides a small simulation core, an API, and the ability to load additional functionality, including models, at runtime. Such additional functionality is loaded from a Simics *module*, which is basically

Full-System Simulation with Simics. DOI: http://dx.doi.org/10.1016/B978-0-12-800725-9.00002-0

a dynamically linked object file. A module can, for example, contain a single device or processor model, or all the models of an SoC or a set of simulator features relevant for a specific use case. This fine-grained structure makes it possible to supply the exact set of models and features needed for any specific user. An overview of the Simics architecture is shown in Figure 2.1.

Simics modularity enables short rebuild times for large systems, because only the modules that are actually changed have to be recompiled. The rest of the system is unaffected, and each Simics module can be updated and upgraded independently. The modularity also allows for changing the simulation system while it is running, such as adding features and reconfiguring the target hardware.

Simics modules are always distributed in binary form, ensuring that a user does not need to build anything to use Simics. Python modules are distributed as compiled `.pyc` files, with source code being optional. In practice, a large part of the module set delivered with Simics is also made available in source-code form to help a user customize and extend their Simics environment.

Even central functions of Simics like the Python interpreter and the command-line interface (CLI) are built as modules and loaded dynamically. This means that the actual Simics executable can be kept very small, at just a few hundred kilobytes. As Simics starts and more functions and models are brought into use, they are loaded dynamically as needed.

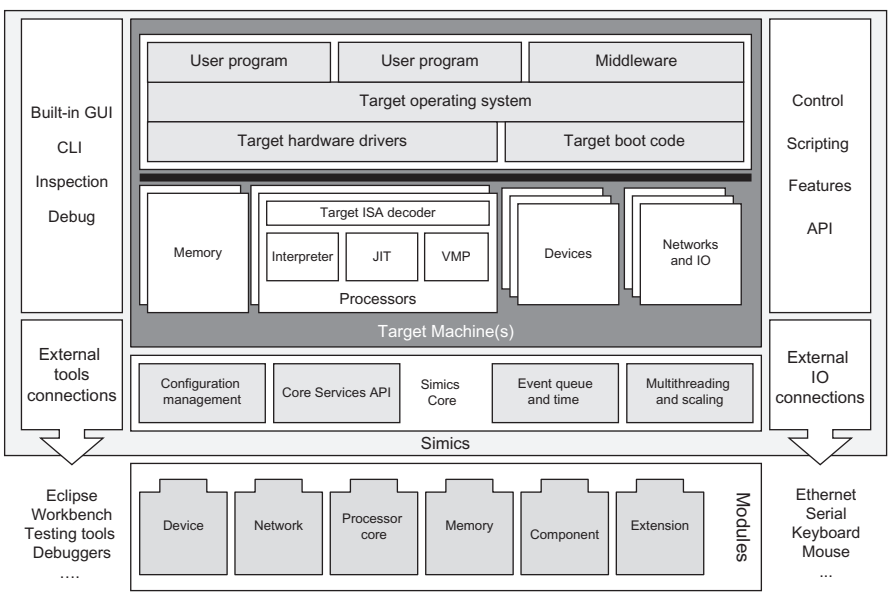

FIGURE 2.1

Overview of the Simics architecture.

RUNNING REAL SOFTWARE STACKS

A key design goal of Simics has always been to run the real software stack, as found on the target system. This includes the boot code or BIOS, operating system, drivers, and the applications and middleware running on top of that. Over the years, Simics has run most types of software, including hypervisors with guest operating systems, small MMU-less embedded operating systems and bare-metal code, as well as Windows, Linux, mainstream RTOSs like VxWorks[†], and major server operating systems.

Running real unmodified software stacks has many benefits. Because Simics is primarily used for software development, running the actual software that is being developed makes eminent sense. The software is compiled using the same tools and compilers that are used with the hardware target, avoiding inconsistencies and deviations introduced by host compilation or other approximations.

Unmodified software also means unmodified build systems, and thus there is no need for users to set up special builds or build targets for creating software to run on Simics. There may also be portions of the system where only machine code is available, such as proprietary libraries, drivers, or operating systems.

Using unmodified software also means that software can be managed and loaded in the same way as on a real system, making Simics useful for testing operations and maintenance of the target system.

To be able to inspect and debug the target software running on Simics, a feature known as *OS awareness* is used. OS awareness is available from the Simics CLI, scripting interface, and debugger. OS awareness looks at the contents of memory and registers in the target hardware and figures out the target software's state, such as the currently running process and thread on a certain processor and all the processes and threads currently active in the target operating system.

Note that there are cases where small modifications to the software stack are appropriate. For example, taking shortcuts early in the development of a target system to get software running without a full boot system in place, or adding drivers to the target software to access simulator features like host file system access. This is discussed in more detail in Chapter 3.

INTERACTING WITH SIMICS

From the very beginning, Simics was designed as an interactive tool that could also be used in automated batch runs. Given the wide range of users and usage scenarios, both CLIs and graphical user interfaces (GUIs) are needed. Today, the primary user interface for new users to Simics is the Eclipse-based GUI, but the CLI is still there for more advanced tasks. Figure 2.2 shows a screenshot of the Simics 4.8 Eclipse GUI, running two simultaneous but separate simulation sessions.

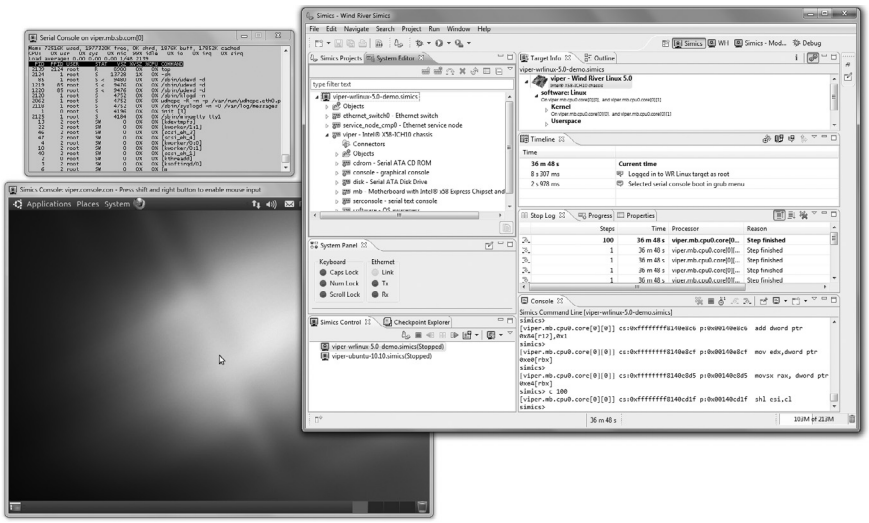

FIGURE 2.2

Simics 4.8 Eclipse GUI, with two Simics simulation sessions running (clockwise from top left: simulated serial text-terminal console, Simics Eclipse GUI, and simulated graphical console).

The currently selected session is running Wind River Linux and uses a serial text-terminal console to interact with the target system (top left). The other session is running Ubuntu Linux on a graphical console (bottom left). In the Eclipse window (right), a number of Simics views that allow inspection and control of the target system can be seen. It is worth noting that the Simics CLI is available and fully integrated into Eclipse, and that actions taken on the CLI are reflected in the state of the GUI and vice versa.

Simics can also be run from a normal command-line shell, on both Linux and Windows hosts. This makes it possible to run Simics without invoking the Eclipse GUI, and this is useful when it comes to automating Simics runs from other tools. Simics behaves just like any other UNIX-style command-line application when needed.

As illustrated in Figure 2.1, the Simics architecture separates the function of the target hardware system from the connections to the outside world. The target consoles shown in Figure 2.2 are not part of the device models of the serial ports and graphics processor unit, but rather they are provided as generic functions by the Simics framework. This means that all consoles behave in the same way, and provide support for command-line scripting, record and replay of inputs, and reverse execution.

In addition to the Simics console windows, a common way to interact with a Simics target machine is via a network connection. In this case, Simics opens up

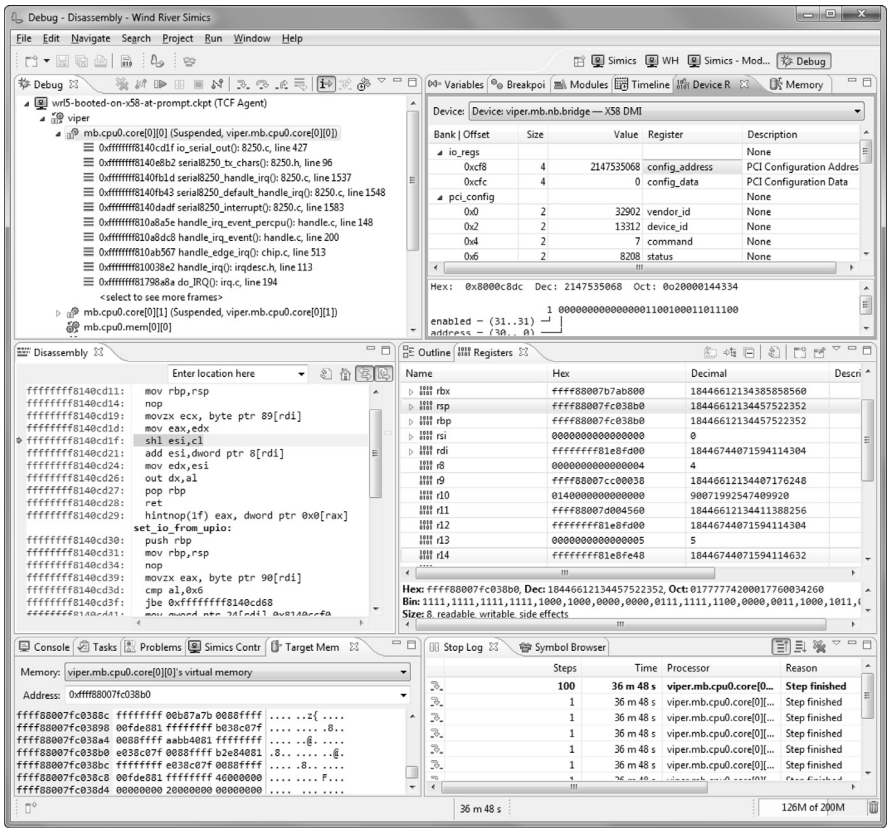

FIGURE 2.3

Simics debugger, looking at various hardware aspects of the target.

a network connection from the virtual network inside of Simics to the host machine or other machines on the network. This feature is known as *real network* in Simics. Users can then connect to Simics with the same tools as they would use to connect to a physical system on their network. Typically, `ssh` or `telnet` is used to get to a target command line, and remote debug protocols are used to control a target from an agent on the target machine. This process is described in detail in Chapter 5.

The Simics interface, the simulator scripting commands, and the Simics API provide rich access to the state of the target system and the simulation itself. It is easy to look inside any part of the system and check the state of hardware devices, processors, memory, and interconnects. Figure 2.3 shows an example of how the Simics GUI can be used to inspect various aspects of the state of the target system. The target software is executing inside a serial port driver in the

Linux kernel, as can be seen from the stack trace in the upper left portion of the window. Other views display the device registers, memory contents, processor registers, and disassembly at the point of current execution.

SOFTWARE DEBUGGING

The Simics interface also includes a very powerful debugger, based on the Eclipse CDT and some Wind River extensions. The debugger leverages Simics' OS awareness feature to allow debugging of individual applications and threads, as well as stepping up and down through the software stack layers. Symbolic debug information can be attached to processors for bare-metal debugging and to software stack context, such as a kernel or user application, for debugging at a certain abstraction level.

The debugger functionality is also accessible from the Simics command line, providing the ability to automate debug tasks and control the debugger from the CLI while looking at the state of the system in the GUI.

The Simics debugger can handle debugging both forward and backward in time, as well as user operations that arbitrarily change the target's state and time.

Simics has the ability to trace or put breakpoints on aspects of the target that are inaccessible on the hardware, such as hardware interrupts, processor exceptions, writes to control registers, device accesses, arbitrary memory accesses, software task switches, and log messages from device models. In Simics it is possible to single-step interrupt handling code and to stop an entire system consisting of multiple networked machines synchronously. The advantages of being able to synchronously stop an entire system are discussed in more detail in Chapter 3.

With checkpoints, it is easy to capture a bug in one location and then transfer a perfect reproduction of it to any developer, anywhere in the world, with no need to describe the system setup or bug scenario. Instead, users just provide a recording of the issue and the developer is able to perfectly reproduce it.

SCRIPTING

No matter how Simics is run, the full power of the CLI and its scripting engine is available. Simics scripts work the same way in a Simics simulation started from Eclipse, in an interactive command-line session, and in an automated batch run on a remote compute server. Basic scripts are written in the Simics command-line language, and for more complex tasks there is a full Python environment embedded in Simics. The Python engine has access to all parts of the simulated system and can interact with all Simics API calls. CLI and Python scripts can exchange data and variables with each other, and it is common to find snippets of Python embedded inside of Simics scripts.

An example Simics script is shown in Figure 2.4. It opens a Simics checkpoint and then runs a command on the target. The parameters to the command are sent in as Simics CLI variables to this script, but they are also provided with

```
## Parameters to run:
if not defined opmode        { $opmode = "software_byte" }
if not defined generations   { $generations = 100    }
if not defined packet_length { $packet_length = 1000 }
if not defined packet_count  { $packet_count = 1000 }
if not defined thread_count  { $thread_count = 4    }
if not defined output_level  { $output_level = 0    }

## Ensure stall mode to enable cache analysis
sim->cpu_mode = stall

## Load existing checkpoint
$prev_checkpoint_file  = (lookup-file "%script%") + "/after-ca001-
booted-and-setup.ckpt"

if not (file-exists $prev_checkpoint_file) {
  interrupt-script "Please run ca001 script first to establish the
checkpoint!"
} else {
  read-configuration (lookup-file $prev_checkpoint_file)
}

$system = viper
$con    = $system.console.con

# Script branch that will run the program and wait for it to complete
# by watching the target serial console
$prog_name = "/mnt/rule30_threaded.elf"
$cmd  = ("%s %s %d %d %d %d %d \n" % [$prog_name, $opmode,
$packet_count, $generations, $packet_length,$output_level,
$thread_count])
script-branch {
  local $system = $system
  local $con    = $con
  local $cmd    = $cmd
  local $prompt = "~]#"
  add-session-comment "Starting run"
  $con.input $cmd
  $con.wait-for-string $prompt
  add-session-comment "Run finished"
  stop
}
```

FIGURE 2.4

Example Simics target automation CLI script.

default values in case nothing is provided. The *script branch* at the end is a construct that lets script code run in parallel to the target system and react to events in the simulation. This makes it very easy to automate and script systems containing many different parts where the global order of scripted events is unknown before the simulation starts. Separate scripts can be attached to the different parts.

Scripting in Simics is implemented by an embedded Python interpreter (using Python 2.7 at the time of writing). This Python interpreter has access to the complete Simics API, thanks to an automatic system that generates Python bindings for the Simics API at build time. The Simics CLI is actually implemented using Python, as are large parts of the Simics infrastructure itself, such as the unit test framework.

CONFIGURATIONS AND THE SIMICS OBJECT MODEL

A running simulation in Simics is called a *configuration*. A configuration is a collection of *configuration objects*, each maintaining its own state. Each configuration object is an instance of a *configuration class*. A class represents some "thing" being simulated—typically a model of a processor core, a memory, an I/O device, an interconnect, or a network. A configuration class can register a number of *attributes* and *interfaces*. The attributes can be used to interact with objects and they are available to other objects and to the user, who can directly interact with the objects through the CLI or via scripts. Classes are defined in *modules*, as mentioned before. Simics *checkpoints* save the entire configuration to disk, allowing it to be reconstructed later. Checkpoints are discussed in more detail later in this chapter.

ATTRIBUTES

An *attribute* has a name and a value. The configuration class determines the types and names of its attributes. Attributes allow the user to configure, inspect, and modify the internal state of a configuration object in a controlled way. Rather than directly changing variables in the implementation of a class, attributes provide a level of indirection that allows the *internal implementation* and *external representation* of the state of an object to be separated. It also means that classes can be implemented in any language, and freely mixed in a Simics implementation.

Attributes can have many different types; for example, an attribute may hold a simple integer value representing the state of a register, it may reference another configuration object, or it may contain a complex list of values such as the state of a TLB. Attributes referencing other objects are of particular importance, because that is the way that configuration objects know of each other. If an object *A* needs to call an interface in object *B*, it will have an attribute giving it a reference to *B*. All connections between objects are done in this way, allowing them to be configured, reconfigured, and inspected at any point in the simulation. Object references in attributes are also properly checkpointed and restored—something that would not be the case if a simple memory pointer was maintained in object *A*.

The attributes should capture the entire internal state of the configuration in a way that allows it to be exported and imported on demand. This is important for two key Simics features known as *checkpointing* and *reverse execution.* Checkpointing allows the entire state of the simulation to be stored to disk and then later restored and restarted as if nothing had happened in the target system. Reverse execution allows the simulation to run backwards in time, and this is extremely useful when debugging. Furthermore, attributes are typically not directly used as part of advancing the simulation. For example, objects should avoid querying or modifying attributes of other objects; that is the job of interfaces.

INTERFACES

An *interface* is a set of methods that do specific things, and is the method by which configuration objects communicate with each other. It is analogous to the object-oriented concept of interfaces as used in, for example, Java. A configuration class can implement any number of interfaces, and even implement the same interface multiple times via the concept of named *port interfaces*. When implementing an interface, the class defines the behavior of the interface's methods for all objects of that class. Interfaces are the main mechanism by which objects in the simulation communicate. While technically attributes and interfaces can both be used to query and modify the state of a configuration object, they serve different purposes. Attributes are for the simulation system and the user; interfaces are for other objects and for expressing the target system's behavior.

Interfaces are normally involved in advancing the simulation. Interfaces do not capture any state and are not used by, for instance, the checkpointing mechanism. Interfaces often represent various hardware communications mechanisms, such as simple 1-bit signals for interrupts, Ethernet packet transmission, or more complex interfaces such as NAND flash. One of the most commonly used interfaces is the `io_memory` interface, which is implemented by objects that have a memory-mapped I/O (MMIO) interface. Memory transactions in Simics use the `io_memory` interface.

Unlike languages like SystemC, the Simics interface concept is unidirectional: it describes a set of methods that other objects can call on a particular object. There is no requirement to set up a connection where both objects involved know of each other. This has the nice side effect that it is possible to call many interfaces from the Simics command line, which is useful for interactive testing and performing actions on the target system. In practice, many interfaces are implemented as a pair of interfaces where both sides know about each other. Examples of this are the relationship between a serial device and a serial link, where the link needs to call the device to deliver data in, and the device needs to call the link to send data out.

HAPS

Simics modules can register and fire off global events known as *haps*. Haps can be related to the simulated machine, such as processor exceptions or control register writes, and the simulator itself, such as stopping the simulation. To react to a hap, a module registers a callback function for the hap. This callback function will then be called every time the hap occurs. Any module can react to any hap, and hap names are global within a Simics simulation session. Any module can register haps. The name *hap* was chosen to not overload *event* with yet another meaning, and *happenstance* was deemed too long. It is also easy to associate a hap with any happening that happens to happen in the system.

Haps should not be used to communicate between device models in a simulation. For that purpose, interfaces are the recommended mechanism, because they

do not involve a trip through the Simics kernel and have much richer semantics. Another disadvantage of haps for communicating between objects is that they hide the communications channel between the objects. With an interface, the sending object will have an explicit pointer to the receiving object. With haps, this is entirely implicit and impossible to determine from the configuration. Haps are useful to communicate important events to scripts and tooling extensions, because the sending object does not need to know who the recipients are.

PORTS

It would not be sufficient if objects could only implement an interface once. Because interfaces correspond to hardware activities, it is often necessary to implement the same interface multiple times in a single object. For example, a general-purpose I/O (GPIO) controller usually controls multiple I/O pins from the same device. Thus, Simics has the mechanism of named *ports*. A named port is an instance of an interface with a particular name, and to communicate with this interface, you need to reference both the object and the name of the port.

DOCUMENTATION AND METADATA

Every module and class in Simics carries its own documentation and other metadata. This makes it possible for Simics to offer help and descriptive text for literally everything inside a simulation session, including components and objects, all the way down to attributes and individual device registers. This self-description keeps documentation tied to the items it relates to, and keeps most of the documentation inside the source code where it is easy to keep it in sync with the code. A large part of the Simics manuals is actually automatically generated by running Simics and extracting the documentation and metadata from modules and classes. Metadata on modules is used to automatically load modules, as classes in them are required.

CALL CHAIN AND LOCALITY

Simics uses an event-driven simulation model (as discussed in more detail later in the chapter). An important consequence of this is that Simics objects do not run in their own threads, but rather they are only activated by calls from other objects, or the simulation event processor. Each function call into an object over an interface or port is executed immediately and in its entirety before returning to the caller. This provides great locality and the best simulation performance.

CHANGING THE CONFIGURATION

A Simics configuration can be modified at any point during a simulation. New modules can be loaded and new hardware models and Simics extensions can be added. All connections between components and objects can be changed from

scripts and the Simics CLI at will. This dynamic nature of a system is necessary to support system-level development work. Reconfiguration can also be used to implement fault injection and test the target software's behavior in the presence of faults in the hardware.

COMPONENTS

To aid configuration and management of a simulation setup, Simics has the concept of *components*. Components describe the aggregations of models that make up the units of a system, such as disks, SoCs, platform controller hubs, memory DIMMs, PCIe cards, Ethernet switches, and similar familiar units. They carry significant metadata and provide a natural map of a system. Figure 2.5 shows an example of a component hierarchy from a simple target called `viper`. The `viper` target has a model of an Intel® Core™ i7 processor and an Intel® X58 chipset.

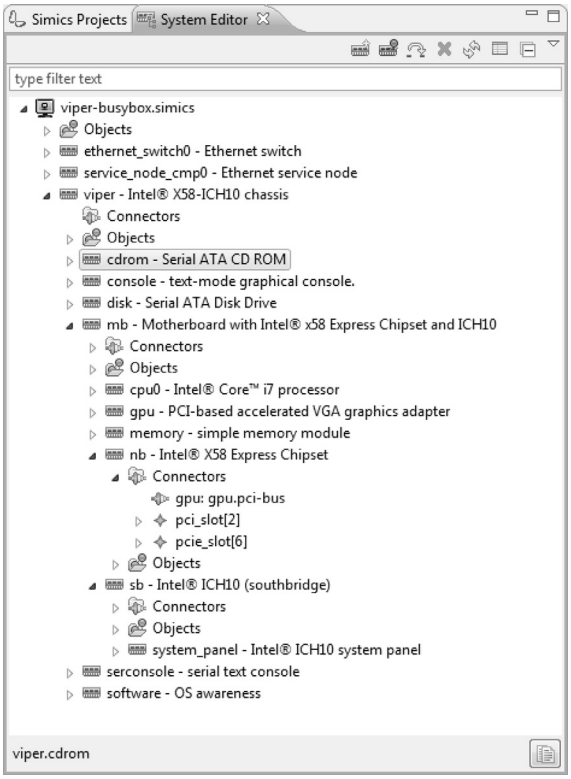

FIGURE 2.5

Example of a component hierarchy from a simple system as shown in the Simics system editor.

Components usually have parameters like the number of processor cores to use in a processor, the size of memories, the clock frequency of a core, or the media access control (MAC) addresses of an Ethernet controller. Components provide the standard way to create Simics simulation setups, and a normal Simics setup script simply creates a number of components and connects them together.

Components encapsulate the details of connections between parts of the system, creating abstractions like DDR memory slots, PCIe slots, and Ethernet ports. Components can be used to change the simulation configuration both during initial setup and at runtime.

Components can contain other components, and normally each board in a simulated system is represented by a single top-level component that in turn creates all the distinct units in the system. This is seen clearly in Figure 2.5, as the `viper` board contains several distinct units that map to the hardware decomposition of the real system. Components are covered in more detail in Chapter 4.

TIME IN SIMICS

The representation and management of time inside the simulation is one of the most fundamental and difficult aspects to consider when creating a simulation environment. This section describes the timing model used by Simics. The first important concept is the difference between *virtual time* and *real time*.

Real time, also referred to as wall-clock time, is the time perceived by humans in the physical world. On the other hand, the virtual time is the time as it is perceived by an observer inside of the simulation. The ratio between the real time and the virtual time, known as the *simulation slowdown*, typically varies during the simulation and can range from zero (simulation is infinitely fast) to infinity (simulation is stopped). A slowdown of one means that the virtual time and the real time progress at the same rate. Simics usually tries to push virtual time forward as quickly as possible to minimize the slowdown.

Simics normally does not attempt to synchronize the virtual time to the real time, because this would destroy determinism and repeatability. Virtual time is part of the state stored in a checkpoint, such that each time the checkpoint is loaded into Simics, the same saved virtual time is used. This makes checkpointing completely invisible to the target software.

Inside of Simics a multiple-clock time model is used. Virtual time is represented locally at one or more *clock objects*. Technically, a clock object is a configuration object that implements the cycle interface, but in most cases it is simply a model of a processor core. Each clock gets to advance time for a certain number of cycles, its time quantum, before the simulation forces a switch to the next clock object. A clock that is advancing time is said to *execute* cycles.

Every processor core in the simulation has its own local representation of time based on the number of cycles it has executed, its frequency, and its offset.

Note that the number of cycles executed may differ from the number of instructions executed, because the execution of an instruction does not need to take exactly one cycle.

Processor models in Simics have a constant rate of instructions per cycle (IPC), with the default value of one IPC. The IPC can be configured by the user and can be changed during runtime to more closely simulate the approximate real-world speed of a processor (Canon et al., 1980). In addition, there are other mechanisms that make a single instruction take more than one cycle. For example, a processor may stall, meaning that it gives up normal execution during some time period. This effectively halts execution of instructions while the cycle queue advances normally with respect to the processor's frequency. Another example is a disabled processor, which works in much the same way. Both stalling and disabled processors will continue to handle events on their cycle queues.

Time is represented in Simics as a 128-bit integer, and the smallest time unit that can be represented in Simics is a picosecond. Using a 128-bit integer places no practical upper bound on the time that can be simulated, which is on the order of a quintillion years. Older versions of Simics used a 64-bit integer, which caused problems for some users, because less than a year in virtual time could be simulated before time maxed out.

ABSTRACTION LEVELS

Simics is mainly used as a fast, functionally accurate simulator for software developers, and it is the authors' opinion that this should be the basic design point for the virtual platform. Where more details are required, individual models or groups of models can be replaced with more detailed variants to study specific behavior. Not only is this efficient in terms of the resources required to create the model, because more detailed models take longer to develop, but it is typically the case that not all parts of the platform need a detailed model. If the foundation for the virtual platform is a more detailed model it is more difficult to abstract "up" and gain simulation performance, because that requires substituting all detailed (slow) models. Figure 2.6 illustrates commonly used abstraction levels and their performance characteristics. The figure also shows the typical users for each abstraction level.

A major division occurs somewhere around the approximately timed (AT) abstraction level. At or below (more detailed) the AT abstraction level models are typically *clocked*. A clocked model drives the simulation by advancing a clock that is propagated out to the different models that are constantly "active" doing work. At or above (less detailed) the AT level it is more common to use *transaction-based modeling*, where the models are driven by higher-level transactions. Such a transaction can correspond to transferring a few bytes of data as a result of a memory load or store, or it can be an even higher level, such as delivery of an Ethernet frame.

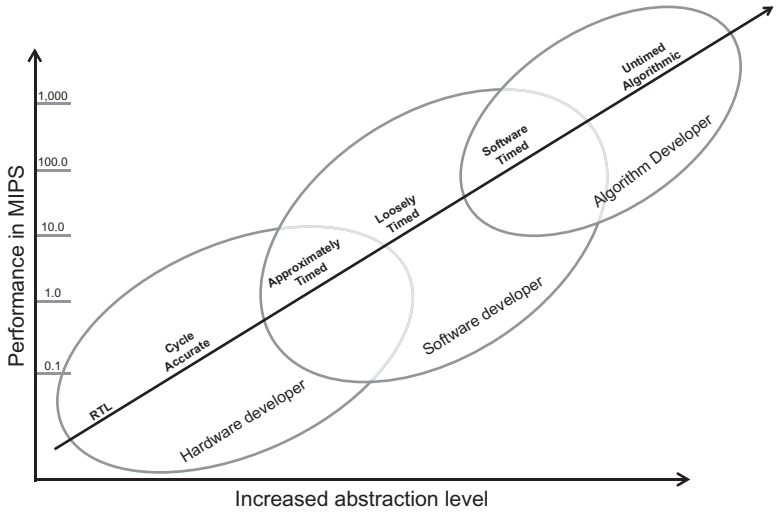

FIGURE 2.6

Virtual platform abstraction levels.

The simulation performance can easily differ by one or two orders of magnitude between two abstraction levels. Similarly, the time required to develop the models is also significantly reduced for the higher abstraction level, although not to such a great extent as the simulation performance is increased.

Simics models are typically developed using the *software timed* (ST) abstraction level to provide maximum performance. The term *software timed* is used to highlight that only enough timing information is added to allow unmodified software to run on the simulated environment. Simics ST is similar to the SystemC† TLM 2.0 loosely timed (LT) model (IEEE, 2011) in that a memory access is a blocking call. However, the Simics model is a special case of the LT model with a zero time delay, which sometimes is also known as a programmer's view (PV) model. The most common timing models are illustrated in Figure 2.7. For the rest of the book it should be assumed that models relate to ST models, unless otherwise stated.

The ST abstraction level means that each interaction with a memory-mapped device, such as a processor reading from or writing to the registers of a device, is handled at once: the device is presented with a request, computes the reply, and returns it in a single function call. The device does not add any immediate delay to the operation, and the result is computed before returning to the processor so that the processor can use the result immediately. In the same way, when a device calls an interface method in another device, it can rely on the result being returned immediately, without any time having passed in the simulator.

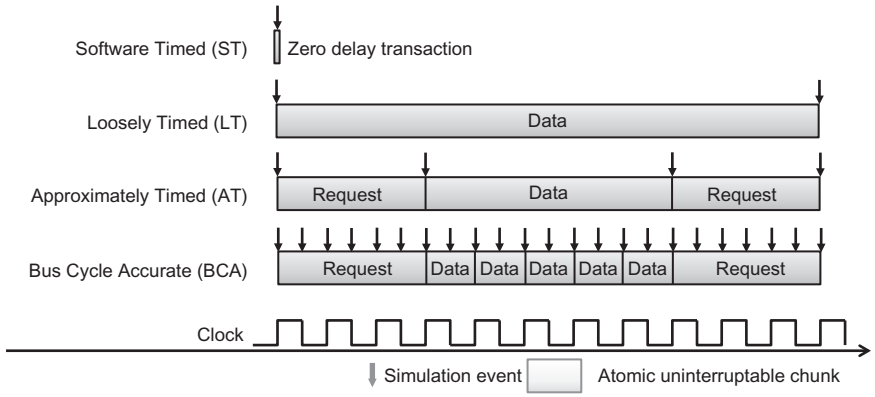

FIGURE 2.7

Different transactions models used to model memory accesses.

EVENT-BASED SIMULATION

The TLM simulation in Simics is *event based*, which means that the simulation is driven by executing a sequence of events. An event is a specific change at a specific point in virtual time. The execution of an event results in the simulation changing from one state to another, and simulation progress is achieved by a sequence of such state transitions. Event-based simulation is typically contrasted with threaded simulation. In an event-based simulation, all actions are taken as a result of events, and once an event is handled, control is returned to the simulation kernel. In a threaded simulation, each model contains (at least) one thread sitting in a main loop, waiting for things to happen. In SystemC, threaded models are coded as `SC_THREAD` and use `wait()` to release control to the simulation kernel. In general, event-based simulation has better code locality and higher performance than thread-based simulation.

In Simics, events are scheduled on event queues, which are typically attached to processor models (in the general case, to any clock object). Because each processor has its own representation of local time, it also has its own event queue.

In a single-processor system the event processing is straightforward. The processor maintains two queues: one step queue and one cycle queue. The step queue's main responsibility is to execute instructions and the cycle queue is responsible for executing other events based on the virtual time. Remember that each processor has its own representation of local virtual time based on its cycle count, frequency, and offset. Any object may post an event on the queue of a clock object. When an event is posted on a specific virtual time, the time is first converted to a cycle count on the processor and then posted on the processor's cycle queue.

As mentioned, only clock objects, such as processors, define time inside the simulation.

Device models are passive and only perform actions when activated due to an event on an event queue. The most common case is that a processor executes a load or store instruction. When the load or store terminates in a memory-mapped device, that device is activated and may perform side effects as a result of the memory operation. It is important to note that no virtual time will pass while the device is called. The other way a device is activated is when a timed event that the device posted earlier—for example, as a result of a store to a register—is triggered as part of advancing virtual time. Such events result in a call to a function inside the device model that has been registered with the event. Events are the mechanism by which device models manage tasks that should appear to take virtual time.

Simics device models handle time and delayed actions by posting events on the time queue of a processor to get a callback at a certain point in virtual time. Figure 2.8 shows a simple example of a timer being programmed from a processor. We can see how the timer device model gets activated each time the processor does a memory access to it. While the device model handles the access, the processor is stopped in the middle of the instruction, waiting for the device model to return. At the second memory access, the timer sets up an event callback corresponding to the timer expiry time. This is done by calling the processor model. Some more instructions are executed by the processor, and when the time for the event comes, the processor stops between two instructions and calls the event handler in the timer device. This event handler calls the interrupt controller device model, which calls the processor to trigger an interrupt. The processor notes that the interrupt needs to be handled, and returns. Once the normal processor loop is

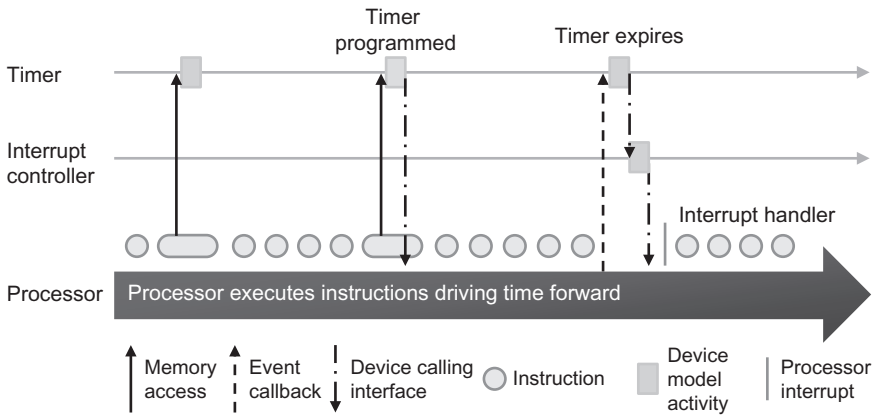

FIGURE 2.8

Device interaction.

back in control, it handles the interrupt by redirecting the instruction flow to the interrupt vector associated with the interrupt triggered by the interrupt controller.

This modeling method is used for anything with a software-noticeable latency. For another example, consider the device driver for an I/O device. The driver expects to be able to run code to prepare for the completion interrupt after writing a register to start an operation in the device. All too often, the driver would crash if the completion interrupt arrived immediately upon starting the operation. The Simics model of such a device would typically compute the device-local effects of the operation immediately upon starting the operation, and then post a timed event at the point in the future when the operation is complete. Only at this time would the processor get the interrupt, and the device driver would see the expected behavior.

MULTIPROCESSOR SCHEDULING

With more than one processor core or clock in the simulation, event scheduling becomes more complicated. To achieve high simulation performance it is important to allow the processors to be *temporally decoupled*. Rather than switching between the units at each step of the simulation, each unit is allowed to run for a certain amount of time, its time quantum, before switching to the next unit. Temporal decoupling has been in use for at least 40 years in computer simulation (Fuchi et al., 1969), and it is a crucial part of all fast virtual platform simulators (Aynsley, 2009; Cornet et al., 2008). Efficient support for temporal decoupling is one of the reasons why Simics processor models have local time instead of a single global time.

The clocks in a configuration take turns to advance time by a small amount. This is done in round-robin order. The amount of time advanced each turn is the *time quantum*, which is sometimes called the CPU *switch time*. For example, assume a system with three clocks and a time quantum *Q*. Their times would then proceed like the illustration in Figure 2.9.

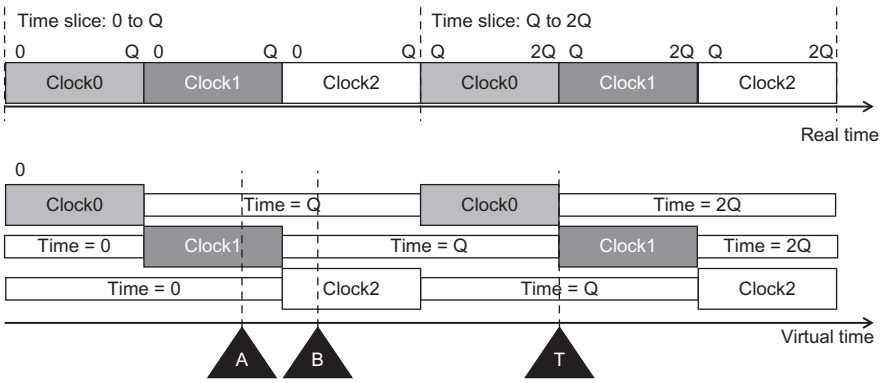

FIGURE 2.9

Round-robin scheduling of temporally decoupled clock objects with time quantum *Q*.

At the point marked T in Figure 2.9, $Clock_0$ has reached time $2Q$ while both $Clock_1$ and $Clock_2$ are still at time Q. As mentioned before, there is no single global time in the simulation; time always belongs to a clock, and different clocks can have different views of the time.

Because of the time quantum, the clocks are loosely synchronized. For instance, the time of $Clock_1$ at point A is greater than that of $Clock_2$ at point B, although A precedes B in the simulation. This may cause time paradoxes if objects using different clocks interact or share a mutable state. The most common communication channel between clocks is memory shared between processors. If processor $Clock_1$ writes a value to a memory location M and its local time is t, that value might well be read by a later processor $Clock_2$ at a local time that is less than t. This behavior might look odd to an outside observer, but it rarely causes any issues, because the code on $Clock_2$ does not know when the value was written. All it sees is a value in memory. However, there have been cases where code designed to synchronize the clocks of processors have failed due to temporal decoupling, thus requiring special handling.

Another potential issue with temporally decoupling the clocks is that events can be posted in the (local) past if processors post events on each other's queues. Consider the case where $Clock_1$, at some point between local time 0 and time A, posts an event on $Clock_0$'s queue at time A; because $Clock_0$ has already passed time A and is already at time Q, the event will appear to be scheduled in the past.

When clocks have different frequencies, the length of the time quantum in clock cycles will be different between the clocks, while being the same in virtual seconds. The number of cycles in a quantum will sometimes vary between individual activations of a clock to avoid drift in the global time synchronization. The scheduler is perfectly deterministic and will always produce exactly the same schedule for the same set of clocks, as that is necessary for a deterministic simulation.

It is important to note that all events on a time queue are triggered precisely. The Simics semantics allow events to be triggered at any point during a time slice. For example, this means that a timer local to a processor will trigger at the right cycle, regardless of how many processors there are in the simulation. Some other simulation systems use temporal decoupling where events are only allowed to trigger at quantum boundaries. This is not the case in Simics, which makes it possible to model hardware event timing more precisely and allows running with a longer time quantum than is practical in most other simulators.

Managing time and scheduling becomes even more complex when the simulation is running in multithreaded mode. This is covered in more detail when discussing networked simulation in Chapter 5.

CYCLE-ACCURATE SIMULATION

When developing hardware, it is common to use models that are called *cycle accurate* (CA) or *clock-cycle accurate* (CCA). Such models are needed to accurately predict the eventual performance and behavior of the hardware and to

validate that the hardware works correctly with respect to bus protocols, cache coherency protocols, and the like.

CCA models are commonly used as a *design tool*, where hardware design teams first build a rough bandwidth and latency model, and then a more detailed model of the internal microarchitecture and pipeline of the hardware. Such models are then used with various forms of input to validate performance goals and find problems in the design. This methodology has been used for processor design since the late 1950s (Brooks, 2010), and it is the dominant paradigm today. From a modeling perspective, such models essentially *prescribe* the timing that the eventual hardware implementation needs to have to satisfy the requirements.

CCA models can also be *generated* from the actual implementation of hardware using tools like the Carbonizer from Carbon Design Systems. Such models essentially serve as a software model of the precise hardware behavior, allowing the simulation of the cycle-by-cycle behavior of hardware as it is actually implemented. This can be very handy for low-level validation of hardware.

Manually programming a CCA model of an existing piece of hardware has been proven impractical. It is very time consuming and almost guaranteed not to match the hardware (Engblom, 2002; Phillips, 2010). Instead, CCA models should focus on the design and validation aspects.

Simics setups can be extended to contain CCA models by building *transactors* that map between Simics ST-level transactions and cycle-level activities. Typically, only a few device models are replaced each time, keeping most of the platform at the native Simics abstraction level. Such hybrid setups are used to support workflows, such as testing detailed models with transaction flows from a full platform, and real software stacks. See Chapter 9 for more details on such hybrid simulations.

MEMORY MAPS

The *memory map* is a fundamental service provided by the Simics framework and is a core component to enabling very fast simulation. Memory maps take care of routing memory accesses from their source to their destination, based on the address of the access. Memory maps replace the explicit modeling of buses in Simics. Memory maps provide the software view of the bus system, but not the hardware view. It is a typical transaction-level abstraction from the hardware behavior. Simics memory maps are dynamic and can be changed during the simulation, both by the user and more commonly from various devices and components that manipulate memory mappings to implement dynamic target behaviors.

Figure 2.10 shows an example of the memory map for a simple system. It is not a complete system, but instead it is just intended to show a few typical cases of memory mappings. The processor has a memory map that shows how RAM and a few devices are mapped, as well as a PCIe memory space in which PCIe devices are mapped.

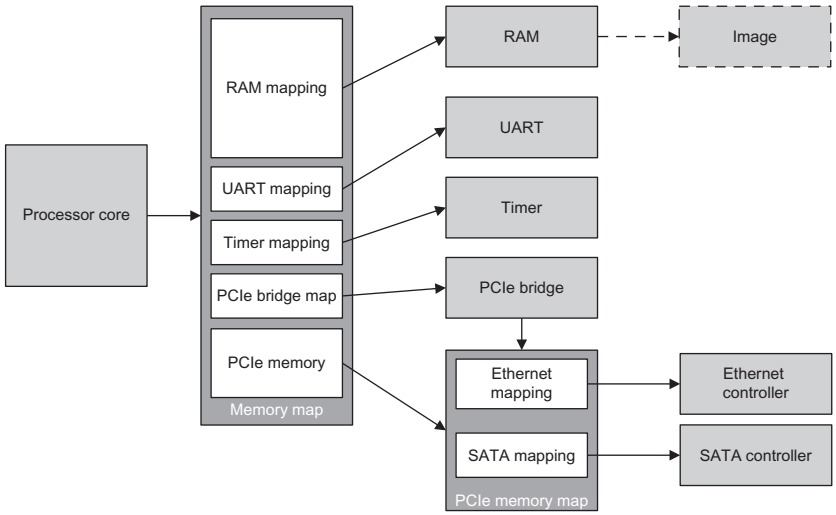

FIGURE 2.10

Memory maps.

HIERARCHICAL MEMORY MAPS

The basic building block for creating memory maps are objects of the memory-space class. When a memory transaction is initiated from an origin, typically a processor model, the address a of the access is passed unmodified to the first memory space. For a processor model, the first memory space typically represents the physical address space of the processor. If nothing is mapped at address a, an access error is signaled to the origin. However, if the access falls within the range of a second memory space that is mapped in the first memory space at offset b, the transaction is passed on to the second memory space with the local address $c = a - b$. Memory spaces are traversed in this fashion until an unmapped address is reached or the transaction terminates in a device model. This process is shown in Figure 2.11.

Figure 2.11 also shows how the final access to a device always ends up using a local offset into the mapped register bank. The device does not need to know where it is mapped in memory to function, and the memory map can be reconfigured without the device being concerned. Figure 2.11 also shows a device being mapped in multiple places. This is trivial to achieve in the Simics memory map system, and the same device can also be mapped multiple times from the same memory space.

RAM, ROM, AND FLASH

When it comes to models of devices like RAM or ROM—that is, memory without side effects—Simics will use efficient caching of the target address to avoid

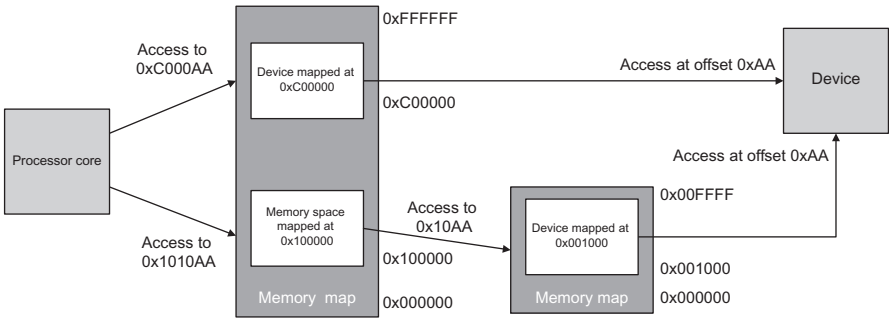

FIGURE 2.11

Hierarchy and multiple mappings.

traversing the memory space hierarchy for every instruction executed, and will instead directly access the underlying memory. Such direct memory access is common in all virtual platforms, and the Simics implementation is based on the information from the memory maps.

In Figure 2.10, we see how the RAM is mapped in the processor's memory map. The RAM object ties the memory map to the image object, which cannot be directly mapped, and implements the support for fast memory access from the processor. The actual data is stored in memory image objects, which are explained in more detail later.

When a Simics model includes a memory controller it is only used to manipulate, initialize, or control the underlying activity of the Simics memory system. The controller is not directly involved in accessing memory.

PCI AND OTHER MEMORY-MAPPED INTERFACES

For the simulation of PCIe and similar interfaces, Simics uses subordinate memory maps cascaded from the primary memory map, configured by the PCIe controller in response to the mappings set up by software following the PCIe probing. The software setup process results in the configuration of a memory map in PCIe space, and devices are mapped into this space. Once configuration is completed, the memory space for PCIe is handled just like any other memory map, with efficient access to resources. The PCIe controller is only used for the configuration.

Figure 2.10 contains a PCIe memory map inside the system's main memory map, containing two devices. Those device mappings are created by the PCIe bridge device when the software probes and configures the PCIe system. The PCIe bridge itself is mapped into the memory map directly accessible to the processor, and it has an arrow down indicating that it configures the PCIe memory map.

Having device models manipulate memory mappings is a very useful implementation trick. For example, it has been used to simulate devices being turned off by replacing the mapping of an actual device with passive RAM. It has been used to implement devices with multiple operation modes, where rather than having each register in the device check the mode on each access, a different register bank is mapped for each mode. When the mode is changed, the device updates the memory map to route transactions to the correct bank.

MULTIPROCESSOR MEMORY MAPS

In a system with multiple processor cores sharing memory, each core typically has its own local memory map. The local memory map provides mappings for core-local devices like interrupt controllers. All accesses that miss the local devices are forwarded on to the shared system-level memory map, as shown in Figure 2.12.

The use of hierarchical memory maps does not affect performance, because the processor models cache accesses to the performance-critical memory where instructions and data are stored. The direct-access mechanisms described earlier function regardless of the depth of nested memory maps.

MEMORY IMAGES

The Simics memory image simulation system provides several features that are unique[1] and enables Simics to tackle simulation of systems far bigger than any

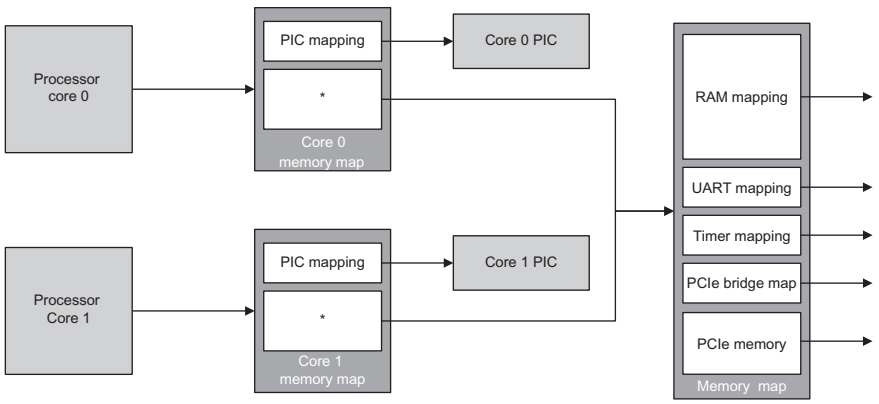

FIGURE 2.12

Memory maps with multiple processors.

[1]Unique at the time of writing in mid-2014, to the best knowledge of the authors, compared to other development-oriented virtual platform products and frameworks.

other simulation system. The Simics memory image system is used to simulate all large storage in a system, be it RAM, flash, or disk.

The image system only represents the memory that is actually being used, which means that the host machine only needs to have enough RAM to represent the working set of the target machines. In many cases this is significantly less than the total amount of simulated RAM. Memory images allow Simics to simulate target memory that is larger than the physical memory of the host machine (Alameldeen et al., 2003a; Rechistov, 2013). This is in contrast to a typical virtual machine monitor system or other virtual platforms, where all target memory is explicitly allocated as part of the simulation setup.

REAL-WORLD STORY: VERY BIG MEMORY

Back in the early days of Simics, a customer was building very large 64-bit servers using Simics. Having gone to 64 bits, it was interesting to find out if the operating system actually handled a fully populated 64-bit memory correctly. For example, was there some code that used signed instead of unsigned so that a memory of 2^{64} bytes would be considered -2^{63} instead? Even if there had been a server capable of loading that much RAM, the cost of buying it would have been bigger than the U.S. defense budget at the time. But with Simics, they just set the memory size to the 2^{64}, and off the system went booting. As long as the simulation did not touch too much memory, Simics easily simulated this on a machine with less than a gigabyte of RAM on the host.

Simics can also reduce the RAM needed by representing identical memory pages only once. Unused pages are never explicitly represented, and it is possible to set the default value of unused memory (e.g., for flash, unused memory has the value `0xFF`). If two different memory pages have exactly the same content, they can also be merged and represented only once. This is the same idea as the Linux kernel's Kernel SamePage Merging (KSM), or VMware's Transparent Page Sharing (TPS). Such pages are surprisingly common, in particular when simulating many instances of similar boards in networks.

If more RAM is needed to represent the target state than is available, the Simics image system does its own swapping to disk, based on information available to the simulator about what is the best data to swap out. It also means that Simics can simulate system memory larger than the virtual memory of the host. This means that the available disk space is the limit to how large images can be simulated.

Simics images also track the pages that have been changed between operations, such as starting the simulation and taking a checkpoint.

Simics images used to simulate disks are normally read-only; this prevents the simulation from accidentally changing fundamental disk images and makes it safe to share the same disk image between multiple target machines. For example, this means that you can boot several copies of the same Linux system into a simulation without them clobbering each other's state. All changes are considered local to each target machine and will be discarded when the simulation exits. If desired, the changed pages can be saved as "diff" files and later reapplied to achieve the new disk state by combining the original unchanged image file with the diff.

Simics images are not in themselves disks, RAM, or flash—they are used as the storage behind those various types of memory or storage systems. Figure 2.10 shows an example of how an image is used to provide the storage for a RAM object, where the RAM object is mapped into the target memory map and the image is behind it.

CHECKPOINTING

Checkpointing is a very important and useful feature of Simics. Making sure that checkpointing works and works well has been the design goal throughout the history of Simics, and has had a deep influence on a number of design decisions (Magnusson et al., 2002; Engblom et al., 2010a). The Simics object model is crucially designed to allow checkpointing to function. By avoiding threads in models, there is no state hidden on the execution stack. Rather, each time the simulator is stopped, all state is stored inside of the objects of the simulator (and typically accessible via attributes).

It does happen that Simics objects contain state that is not checkpointed; however, such state is defined to be ephemeral and has to be reconstructed each time a checkpoint is opened. Typically, the non-checkpointed state consists of various caches of information from the rest of the simulation, which can be reconstructed from the state of other objects. Such state is cached locally in an object purely as an optimization for performance.

A Simics checkpoint is really a directory in the host file system, which contains several files. An example is shown in Figure 2.13.

- The `config` file describes the current state of all processors, devices, and other checkpointed simulation modules. It is a text file following standard Simics object state syntax.
- The `info` file provides the metadata for the checkpoint, including the versions of Simics used to create the models, and user-provided comments and annotations. This is also a text file following Simics standard object state syntax.
- A set of Simics `craff` (compress random-access file format) files for all memory images in the simulation that have changed because the simulation started or the last checkpoint was saved.

REAL-WORLD STORY: VERY BIG CHECKPOINT

The Simics checkpointing system has proven very robust in the face of demanding users. Sometimes to an extent that even surprised the developers. One example was a customer who built their own custom Simics models and ended up with a very large system model containing a thousand or so instances of a device with a few thousand registers in it. The result was a `config` file that was about 5 gigabytes in size, enough to overload most text editors. But Simics was able to parse and load this checkpoint file without a hitch. It is nice when your product surprises you in this way!

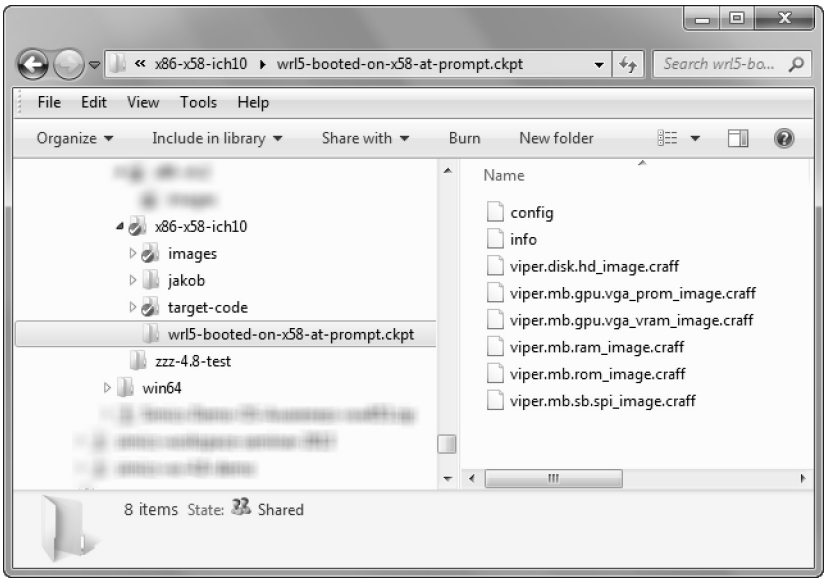

FIGURE 2.13

Example Simics checkpoint.

PORTABILITY AND IMPLEMENTATION INDEPENDENCE

Simics checkpoints are portable across host types and implementations. In practice, this means that a checkpoint of a target system taken on a 32-bit Windows machine using Simics 4.4 can be opened on a 64-bit Linux machine using Simics 4.8 (as long as the same set of simulation models are available). This is enabled by the explicit export and import of state, without assuming any common implementation between the saving of the checkpoint and the later loading of it.

Such portability is not possible with approaches to checkpointing that involve dumping the entire binary state of the target memory to disk and then restoring it into a new process. Such checkpointing only works when the basic simulation code is unchanged and you restore the checkpoint onto the same machine or one with a virtually identical operating system installation. Even the change of a single system library can break such checkpoints if some data structure stored in the checkpoint is affected (Kraemer et al., 2009).

In case the implementation of a model does change enough between model versions to make old checkpoints invalid, Simics allows the model designer to create checkpoint update code that transforms the old checkpoint state of an individual model to a new format that fits the new version of the model.

DIFFERENTIAL CHECKPOINTS

Simics checkpoints make use of the differential memory image system described before to save space and reduce the time needed to save a checkpoint. This means that to open a checkpoint, you need all previous checkpoints that it depends on and any disk or memory images used to set up the initial simulation state.

The advantages of using differential checkpoints are that saving a checkpoint can normally be a very quick process and that checkpoints can remain fairly small. Checkpointing systems that rely on saving the complete target memory state often have to save many gigabytes of memory to disk, even if only a tiny portion of it has changed since the last save. With Simics, only the change is saved, and if nothing has changed, nothing needs to be saved at all. Thus, the size of a checkpoint depends on how much the target system state has changed since the last checkpoint, and not on how large the target system is overall.

SESSION STATE AND SIMULATION STATE

Simics checkpoints are defined to contain the simulation state—that is, the state of the simulated system. By design, they do not save session state such as command-line variables, breakpoints, symbol file associations in the debugger, or running scripts. Excluding the session state is necessary to facilitate sharing checkpoints between users and using checkpoints to archive state for the future.

For example, it would be very confusing to a user who opens a checkpoint and starts an overnight simulation if Simics suddenly stopped on a breakpoint the user had never set and did not know about, just because it was included in the checkpoint by mistake. By cleanly separating the target state (checkpointed) from what the user is doing with the target (not checkpointed), checkpoints remain robust communications tools.

If a user wants to reproduce a particular Simics session setup, it is usually done by running a script after opening the checkpoint to restore the session state. Such a script would start up script branches, add symbol file mappings to the debugger, and set up the values of command-line variables. The Simics Eclipse debugger will save the debugger state in Eclipse, including symbol file associations and breakpoints.

DETERMINISM AND REPEATABILITY

Simics is normally deterministic; this means that from a given starting configuration, the simulation always proceeds in exactly the same way. This is a very useful property, because it allows analysis of the same simulation over and over again, and makes it easier to debug the simulator and its models, as well as the software running on the simulated system. Determinism is also a requirement for reverse simulation.

A simulation is only deterministic if all of its parts are. In particular, all classes that are used in the simulation must be written deterministically. For example, if a module uses random numbers, it must maintain the random number seed as part of its configuration so it can use the same random sequence for each run.

Models that communicate with the outside world require particular care. This includes communication over a real network connection and keyboard or mouse input. In these cases, input has to be recorded using the *recorder module*. This permits subsequent reexecution of the same simulation to be performed with already recorded input instead of reading new data. Simics can save the recorded asynchronous inputs along with a checkpoint so that another user can replay the exact same execution scenario. Input can also be scripted to provide deterministic behavior during testing, for example.

Deterministic does not mean *predetermined*. In a predetermined execution, you would know what is going to happen before it happens, and the exercise would be fairly pointless. Instead, the system's execution on Simics will vary for the same reasons it varies in the real world: the exact code loaded, the timing of asynchronous events, and the accumulation of state over time. Simics executions exhibit the same *variability* as a real system would, with the difference that *repeating* a certain execution is trivial thanks to determinism and recording.

In a variable system, each time an action is performed, the effect might be different. A good example is running a multithreaded program on a multicore computer. Each time it runs, the precise pattern of thread starts, thread switches, locks, thread communications, and other concurrent properties of the program will be different. Sometimes this affects the program results, sometimes not. Such variation will be present on a deterministic virtual platform too, as long as the program starts from a different state each time. For example, running the same program many times in succession within the same simulation will lead to variability because the state of the simulator will be different between runs. Similarly, if nondeterministic user input, such as mouse movements or keyboard input, is provided to the simulator, variability will manifest itself. However, if the simulation is restarted from time zero or from a checkpoint and runs under deterministic script control, each such run will repeat the same execution.

REVERSE EXECUTION

Simics can give the illusion of running the simulation backwards; this is referred to as *reverse execution*. Simics uses an implementation model for reverse that is known as *reconstruction* (Engblom, 2012a). When enabled, the entire simulation state is automatically saved in memory, in what is known as a *micro-checkpoint*, at some interval. The micro-checkpoints permit Simics to restore the configuration to a previously saved state and proceed to any subsequent point.

Figure 2.14 shows the basic technique used to reach a certain previous point in time in Simics. First, the system state is pulled back to a micro-checkpoint, which comes *before* the desired target time. The system then executes forward

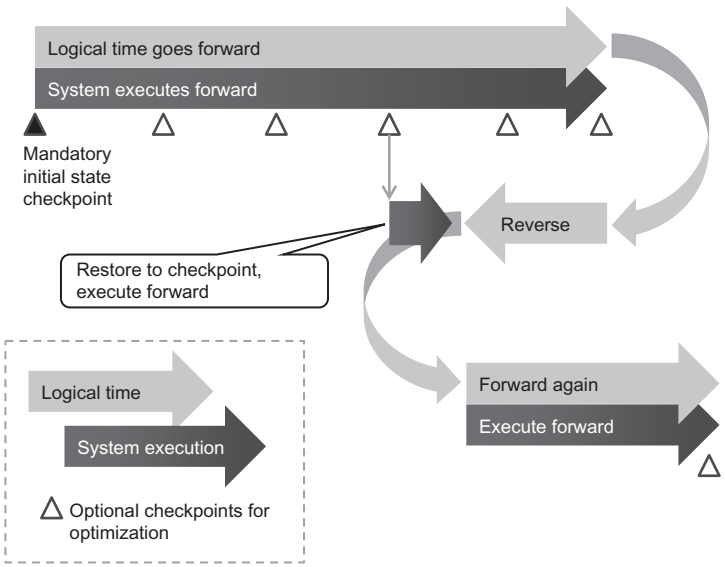

FIGURE 2.14

Reverse execution by reconstruction.

until the desired point in time has been reached. The crucial fact is that the execution of the system model is always in the forward direction, even if the user-perceived time is indeed going backwards.

For reverse execution to work, all models used must be deterministic and checkpoint-safe. This is considered a requirement for normal model development. When using reverse execution for going backward in time, it is not defined how many times interface methods and callbacks are called, or in what sequence. Some special cases, such as console output and breakpoints, are temporally ordered by Simics.

Reverse execution is used to implement reverse debugging, as discussed in Chapter 3.

RECORDERS

As mentioned before, a key part in enabling deterministic behavior, repeatable runs, and reverse execution is the recording of asynchronous inputs to Simics. This is done using the *recorder* class in Simics, a piece of simulation infrastructure provided by the Simics framework. Recorders are used both to replay inputs during a reverse execution run and to provide the ability to save and replay input scenarios associated with checkpoints.

A user module that communicates with the outside world has to make sure to save all inputs using a recorder; see Chapter 9 for more about writing user modules that interact with the world.

SIMICS PERFORMANCE TECHNOLOGY

Simics is designed from the ground up to be a fast simulator. The first use of Simics was to boot a multiprocessor 64-bit Solaris OS on a simulated Sun Workstation (Magnusson, 2013; Magnusson et al., 1998), which required pretty good performance. Since then, many different performance technologies have been added to Simics. Some of them are described in detail in later chapters, when they are applied.

- Transaction-level modeling (TLM) improves simulation performance by reducing the amount of detail in the model.
- The Simics memory map abstraction speeds access to devices and target memory.
- *Just-in-time (JIT) compilation* where the Instruction Set Simulator (ISS) translates target instructions to host instructions for faster execution. The JIT can run in a thread parallel to the main simulator thread, reducing the impact of the JIT compilation phase on the overall simulation.
- For Intel® Architecture targets on Intel® Architecture hosts, *Simics VMP* uses Intel® Virtualization Technology (Intel® VT) for IA-32, Intel® 64, and Intel® Architecture (Intel® VT-x) virtualization instructions to directly execute target code on the host processor. This makes it possible to get close to native speed.
- *Direct memory access* and instruction and data TLB caching enables the ISS to directly access the host memory used to simulate target memory, without checking the target MMU mappings and simulator location of target memory pages for each access.
- *Temporal decoupling* improves simulation locality.
- *Multithreaded simulation* lets Simics use multiple host processors to simulate multiple target machines and multiple target processors.
- *Hypersimulation*, also known as idle optimization, allows the simulator to skip ahead in virtual time when a processor is idle. Idleness is either executing an architecturally defined idle instruction such as HALT, MWAIT, or WFE; executing a branch to self (common idiom); or spinning in a loop that will not terminate until something happens in the target machine. Such loops are automatically detected by analysis of the executing target code or manually defined for cases where Simics cannot automatically detect them.
- The Simics image system reduces host memory pressure, allowing larger systems to be efficiently simulated.
- Distributed simulation extends the multithreaded simulation paradigm across multiple host machines to simulate really large target systems.

SIMULATION SPEED MEASURES

When discussing simulation speed, it is important to understand that there are several ways to measure simulation performance. The two main ways to measure the target system's progress are simulation *time* and simulated *instructions*.

For a software-oriented, fast, transaction-based simulator like Simics, Qemu, ARM Fastsim, SystemC TLM, or IBM CECSim, the preferred measure is the number of target instructions executed per host second. This is the simulated instructions-per-second (IPS) number. Hopefully, a prefix such as *mega* or *giga* can be applied to the IPS, giving up to MIPS or even GIPS to work with.

For hardware-oriented simulators, such as RTL simulators, SystemC used for cycle-accurate simulation, or hardware emulators, *time* is the better measure. Time is measured in target cycles per host second, which is usually expressed as hertz, or Hz (1 Hz equals one target cycle per host second).

IPS and Hz are not directly comparable, because it typically requires several cycles to process one instruction. The activities in a fast transactional simulator do not translate directly to the activities of a cycle-level simulator. Still, at a rough level it usually works to compare IPS and Hz to get an idea for how quickly software can progress on the simulator.

Another popular measurement is *slowdown*—compared to a physical machine running the same workload, how much slower (or faster) is the simulator? This is a very practical measurement because it relates performance to the work a user wants to get done, but it is also very hard to define in a precise way in the simulator without a clear external reference. Simics provides slowdown measurements by looking at the clock frequencies set for its target processors and comparing this to the progress of time in Simics. This means that the same simulation speed can correspond to widely different slowdowns, if the processor clock frequency is different. As a measure of the inherent speed of a particular simulator, a slowdown number is meaningless. IPS and Hz do indicate the speed of a simulator in a way that can be compared between simulators, as they are based in absolute numbers. The inverse of slowdown is sometimes also used and is referred to as the *real-time factor* (RTF).

Domain-specific measurements can also be used when simulating particular workloads to indicate the simulation progress in terms interesting to the user. For example, network application users might be interested in network *packets processed per second* or *network data volume transmitted per second*. Multimedia applications can be measured by their *frame rates*. Such measurements require some custom tool that is able to observe the simulation output and interpret it in domain-specific terms. While similar to slowdown, they allow the comparison of different ways of doing things regardless of any particular speed reference.

MULTIPROCESSOR SIMULATION SPEED MEASUREMENT

When simulating multiple processors, simulated *time* per host second (RTF) works just fine. Target time will progress at some speed, and simulating a larger system typically just makes this move slower. However, simulated *instructions* per host second (IPS) becomes a bit trickier because it can be considered both for each processor individually or for the simulation as a whole.

Starting with an ISS that progresses at N IPS for a single target processor and running P such processors in the same simulation using round-robin scheduling means each ISS will run for $1/P$ of each real-world second. Thus, the progress of each individual ISS will be N/P IPS. However, the progress of the simulation as a whole will be $P \cdot (N/P)$, which is N. Thus, it is fair to call this an N IPS simulator regardless of the number of processors simulated. For clarity, the overall IPS might be called *aggregate IPS*, indicating the sum of progress on all processors in one host second.

A domain-specific measurement like frames per second or packets per second might or might not reflect the effect of the round-robin simulation; it depends on the behavior and load balance of the software being run on the simulator. Slowdown tends to go up as more processors are simulated using a single host processor.

When using multiple host cores to simulate multiple target processors or processor cores, there are essentially multiple ISSs running in parallel. Thus, if there are H host cores being used to run N IPS ISSs, the aggregate IPS is going to be $H \cdot N$. To measure the fundamental technology, the N IPS of the basic ISS still seems the most appropriate. Note that the $N \cdot H$ number assumes linear scaling as the number of host cores increases, which is only reasonable if the target software parallelizes perfectly and there is insignificant communication needed between the simulated cores.

SPEED VARIABILITY

Given the complexity of a full-system simulator and the many different performance techniques used, it should not come as a surprise that simulation speed tends to vary greatly during a run and between different runs. The actual performance of a simulator like Simics depends on the nature of the code being executed, the type of the processor being simulated, target code and simulator code locality, achievable parallelism in the simulation, and many other factors. This often changes during a simulation run, as the target code goes through different phases. To give an idea for the variability, Figure 2.15 shows a performance profile of Simics running a boot of a dual-core embedded target system running a real-time operating system. The horizontal axis shows the *virtual* time, and the vertical axis shows the measured MIPS as well as the slowdown. Note that the vertical scale is logarithmic to capture the very large magnitude changes over the run.

It can be seen in Figure 2.15 that early in the boot, the slowdown is close to 10. This is due to the software performing many device accesses, which is an expensive operation in any Simics-style simulator. After about one second, the boot speeds up for a short while when the target waits for some tasks to complete. Then there is another bout of intense processing, and finally at around three virtual seconds, the target is booted to prompt and starts idling at many billion instructions per host seconds (the top number was around 50 GIPS), executing

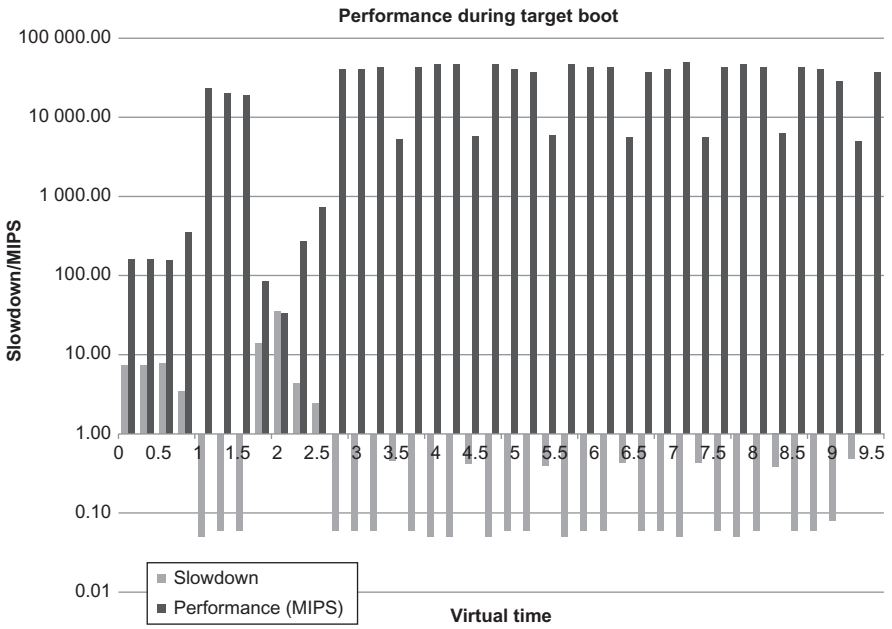

FIGURE 2.15

Relationship between virtual and real time.

with a slowdown below one, which indicates that it executes faster than a real machine would. It can also be seen that once every virtual second, the target executes a bit slower for one measurement interval. This is caused by periodic operating system processing making the target less idle, reducing the effect of hypersimulation.

TEMPORAL DECOUPLING

As mentioned before, the length of the time quantum used in temporal decoupling has a huge effect on the performance of the simulation. A time quantum of a few cycles will slow the simulation down, even if everything is operating as a transaction-level model with no timing details added, because it reduces the locality of the code and invokes the overhead of switching between processors more often. Having a too short time quantum also disables performance techniques such as JIT compilation or direct host execution (VMP).

Typically, Simics systems ship with a temporal decoupling time quantum of 10,000 to 100,000 clock cycles. Experience has shown that this range offers the best balance between reasonable software behavior in a multicore system and simulation performance. See below for more on this subject.

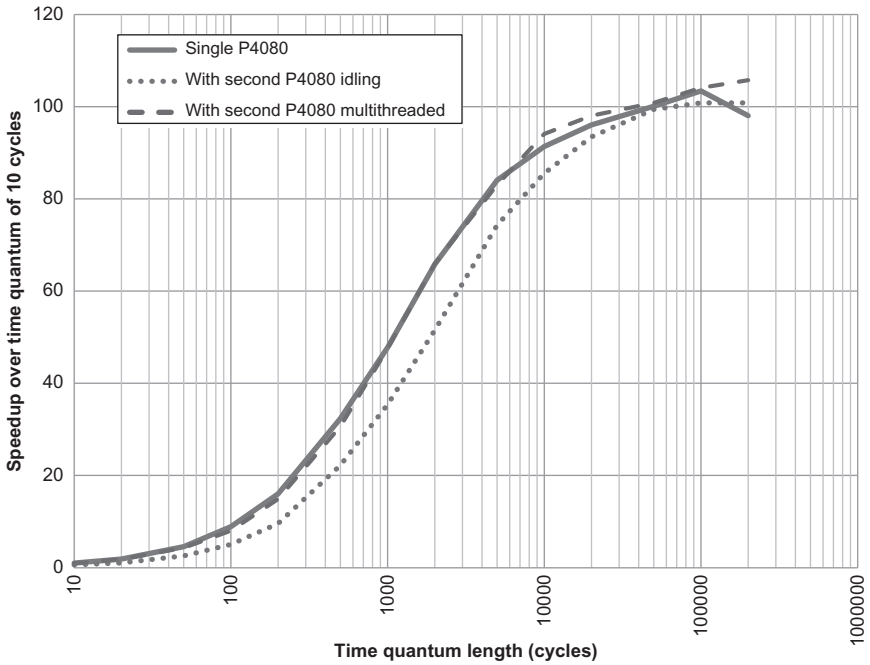

FIGURE 2.16

Performance scaling with different time quanta.

Figure 2.16 shows how the performance improves as the time quantum is changed from 10 to 500,000 cycles on an eight-core Power Architecture target. The target system processors are simulated on the Simics JIT-compiled ISS and run a computationally intense benchmark on all eight cores. Performance improves radically before peaking at around 100,000 cycles.

We also see the effect of Simics hypersimulation; when adding a second identical machine to the setup, performance does not suffer very much as long as the machine is idling. When using multithreading to simulate the second machine, there is no impact on the performance on the first machine.

Figure 2.17 shows the runtime of a checksum computation running on an Intel® Architecture target system, as the time quantum (called CPU switch time in Simics) changes from 100 to 100 million cycles. The Intel Architecture system is different from the Power Architecture system because it uses the Simics VMP technology to directly execute code on the host. This makes the simulator more sensitive to being interrupted in the middle of executing instructions, and thus it makes sense to use a longer time quantum. Peak simulation performance is thus reached at a longer time quantum than for a JIT-based simulator, at around 1 million cycles.

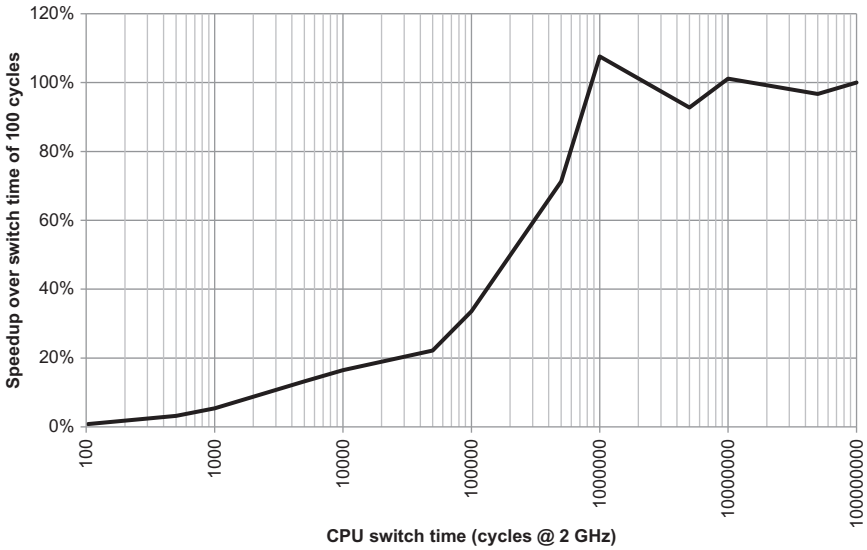

FIGURE 2.17

Temporal decoupling and performance with VMP.

The target system used in the example shown in Figure 2.17 is an Intel® X58 chipset with a quad-core CPU implementing the Intel® microarchitecture code name Nehalem, with two threads per core. The data is generated by running four instances of the `md5sum` program to calculate the checksum of data received from a pipe, thus no disk I/O is involved in the process. The checksum computation is a compute-bound process.

In addition to affecting Simics performance, the time quantum length used in a run might have an effect on how the software executes. For a target processor operating in the GHz range, the Simics default time quantum is on the order of microseconds, which is usually short enough not to be noticeable by the target operating system and software load.

However, temporal decoupling can affect the order in which events happen, and this might have software-visible effects. For example, consider the scenario in Figure 2.9. With the given time quantum Q, A will be simulated before B, even though the virtual time stamp of the event B is lower than that of A. However, if the time quantum is reduced to half of Q, B would happen before A, as illustrated in Figure 2.18. If A and B involved side effects such as sending data to an output device, those side effects would occur in a different order in the two simulation runs.

Another example is if the computation performed at A and B involves some shared state that is read and modified. In such a case, the value computed in the two runs will most likely be different, because A and B happen in different orders. It is very similar to the nondeterministic effects seen in parallel software, except

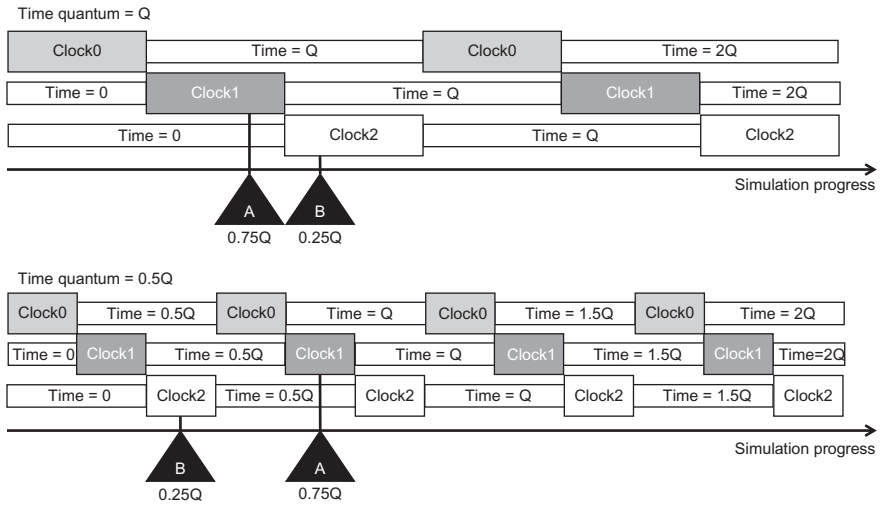

FIGURE 2.18

Event order and temporal decoupling.

that in Simics it is repeatable—as long as the time quantum is kept the same, it will behave the same way. It is also important to note that when this code is running on physical hardware, typically both event orders are possible, due to weak memory models, thread scheduling decisions by an operating system, and other system noise. The Simics simulation corresponds to one possible real-world scenario. Varying the time quantum length in Simics has proven to be a fairly effective way to cause varying event orders, and thus finding and repeating timing-dependent bugs that can be very hard to find on physical hardware. Chapter 3 describes one such case.

There have been examples of multicore software loads that require a fairly short time quantum to work properly. Such software loads tend to be concerned about time and will check the current time across processor cores using some high-precision local timer. A time quantum of 100,000 or even 10,000 cycles might be enough to be noticeable for such software, and the solution is simply to lower the time quantum length to a point where the software functions correctly.

Exceedingly long time quanta can also have effects on a system. When time quanta reach into virtual seconds, even standard non-real-time operating systems will notice and will generate errors indicating that the operating system believes that processor cores are stuck or crashed.

When multiple cores are competing for shared resources, longer time quanta can have the effect that one core gets an unfair share of time. Assume that a program running on a core locks a resource, operates on it for some time, and then releases it. If the program is such that the resource is mostly locked, the core that first gets the lock will most likely lock out the other cores. Figure 2.19 illustrates

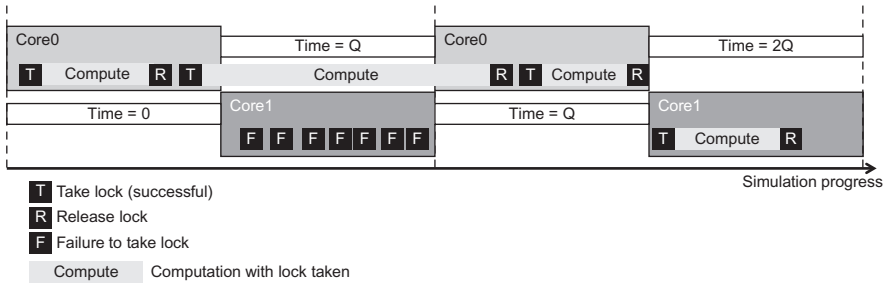

FIGURE 2.19

Temporal decoupling and locking.

this issue. Within its time quantum, *Core0* takes the lock, computes, releases it, and immediately takes it again. Because *Core1* does not run between the release and the take, it will not have a chance to take the lock. On physical hardware, it is quite possible for *Core1* to grab the lock from *Core0* at this point, but with temporal decoupling that will not happen. In the second time quantum, *Core0* releases the lock just before its quantum is up, and then *Core1* is able to come in and take it. But if the compute had taken just a little more time, the scenario from the first time quantum would have been repeated.

The issue illustrated in Figure 2.19 is real, but it is usually not noticeable in real systems. There is a lot of noise in a modern multithreaded, multitasking, multicore system, and it is highly unlikely that software will be "hogging" a lock in the manner presented here—that is bad for performance on a real system too and would have other undesirable effects.

One case where the locking time for shared resources does matter is the simulation of multiprocessor cache systems and shared buses, as well as when studying multicore synchronization at the level of individual instructions. For such work, it is necessary to use a very short time quantum to get reasonable results. Chapter 9 has some more discussions on this issue. Still, the experience from many decades of multiprocessor simulation is that using a time quantum between 1,000 and 100,000 cycles works well for most software stacks.

PERFORMANCE EFFECTS OF CHANGING TARGET TIMING

In general, Simics tries to push the simulation forward as quickly as possible. This means that the amount of real time spent executing a workload is useless as an indicator of the target speed. Even if the target processor clock speed is changed, Simics should complete the same workload in approximately the same amount of real time, even if the virtual time spent is radically different. The only exception is if Simics is running with real-time mode enabled, as discussed in Chapter 9.

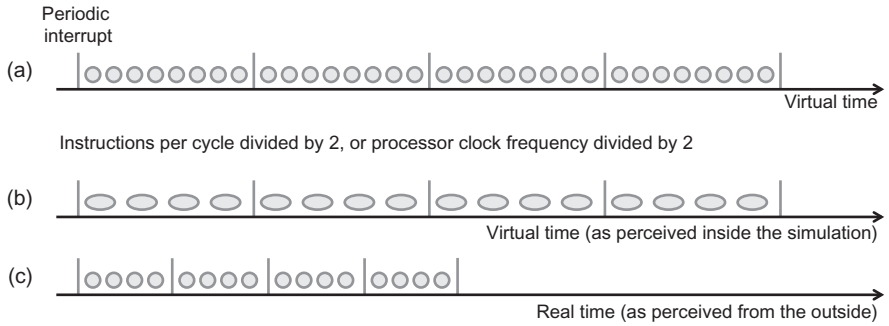

FIGURE 2.20

Performance effects of changing IPC or processor frequency.

Still, changing parameters such as instructions per cycle or the clock frequency of a simulated processor can have relevant effects on the perceived performance of a target system. Normally, Simics runs software as quickly as it can, and the instructions per second executed should not change as the operating frequency or IPC of a simulated processor changes. All that changes is how virtual time relates to the instructions executed. This does have an observable effect in most cases, because OS work and thus overhead is typically scheduled based on intervals of virtual time, such as periodic timers triggering 1,000 times per (virtual) second.

Consider the case shown in Figure 2.20. Figure 2.20(a) shows the normal execution of the system, where an interrupt hits every eight units of work. If we change the IPC to be ½, or change the virtual operating frequency to be half of its original setting, the net result will be that each instruction takes twice as much *virtual* time to execute, and thus that each unit of work takes longer to complete.

Figure 2.20(b) shows the effect in virtual time. The periodic interrupts hit at the same point in virtual time, but instead of eight there are only four units of work executed in each period. If each interrupt invoked the operating system for some periodic work, this increases the proportion of work spent in the operating system. Less virtual time will be left for user code to get work done, essentially slowing down user code processing. However, Figure 2.20(c) shows the effect from an observer looking at the system from the *outside*. Because each unit of work still takes the same amount of *real* time to execute, the periodic interrupts will hit more often in real time, and virtual time will appear to run faster in relationship to real time.

As long as the target system is mostly idle, this can be a very useful effect to exploit to improve the performance apparent to a user. For example, in case a system is spinning waiting for a timer to expire, setting the cycles per instruction to a value like 10 or even 30 will make the simulation get to the end of the loop much faster (assuming the loop cannot be hypersimulated due to some

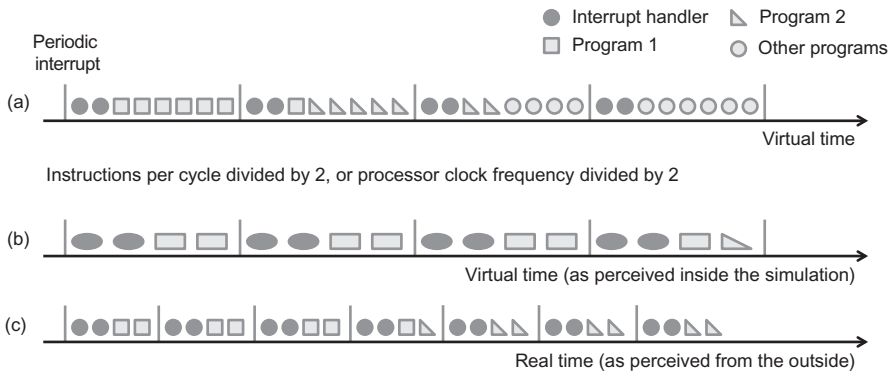

FIGURE 2.21

Performance effects under heavy load.

complexity). It has also been used to make GUIs more responsive, because they typically pick up mouse movements using a periodic interrupt.

However, if a system is heavily loaded, the net effect will be to slow down user processing due to increased OS overhead, as shown in Figure 2.21. Figure 2.21(a) shows the original scenario, with two programs (program 1 and program 2) running, along with an operating system. Each time a periodic interrupt happens, the operating system does some processing. The OS processing overhead is greatly exaggerated for clarity, at about 25% of total CPU time. We then lower the operating frequency by a factor of two. As illustrated in Figure 2.21(b), this means that the OS consumes 50% of the CPU time, and the programs require more virtual time to complete. Program 1 finishes after four periodic interrupts instead of after two. Figure 2.21(c) shows the effect apparent in real time. The behavior in real time is not as bad, but the end of both program 1 and program 2 are clearly delayed compared to the original execution. Thus, changing the IPC or operating frequency was a bad idea in this scenario.

MODELS AND EXTENSIONS

Simics users have the ability to create arbitrary extensions to the simulator, in addition to hardware models. Hardware models are supposed to use a small defined set of APIs to make it easy to maintain determinism, reversibility, checkpointing, and high simulation performance. Extensions have access to the full Simics API and can do anything inside of Simics. Simics itself comes with a large set of extensions, implementing its user interfaces, debug connections, and most of the features of the simulator. In theory, any Simics user could reimplement any part of this, but in practice, user extensions tend to focus on building application-specific inspection and connection functionality. Typical examples include

customized instruction and data trace modules, race condition detectors, and connections between the Simics-simulated computer system and external simulators for the physical world.

Often, extensions begin their life as Python scripts. Python scripting is used to prototype the functionality, but as the desired functionality is clarified and the complexity of the task increases, it typically makes sense to move things into a Simics extension module.

Extensions are described further in Chapter 8, and modeling is described in Chapters 4, 6, and 7.