WIND RIVER

Creating Virtual Platforms with Wind River Simics

Jakob Engblom, Technical Marketing Manager, Wind River Simics

Executive Summary

Wind River Simics is a full system simulator used by systems and software developers to simulate the target hardware of everything from a single processor to large, complex, and connected electronic systems. Referred to as a virtual platform, it simulates the functional behavior of the target hardware. This enables the target software (same board support package, firmware, real-time operating system, middleware, and application) to run on the virtual platform the same way it does on the physical hardware. The simulation is fast enough that it can be used interactively by developers.

Within this fast and accurate virtual environment, engineering, integration, and test teams can use approaches and techniques that are simply not possible on physical hardware. For example, developers can freeze, save, email, and restore the whole system; they can view and modify every device, register, or memory location; and they can run the whole system in reverse to find the source of a bug. Virtual platforms provide software debugging and analysis features that are not possible to implement on physical hardware.

Virtual platforms have the potential to transform how embedded software products are developed, by making hardware availability issues nonissues and reducing development time and schedule risk. Virtual platforms have been found to reduce capital expenditure (CAPEX) costs by 45%, time-to-market by 66%, and debug time by 35% or more.

A virtual platform for a particular system or application allows you to quickly develop the system model. This is the task of system modeling. This paper discusses how system modeling is supported in Wind River Simics. At the core, Simics is an extremely fast transaction-level model (TLM) simulator. Simics features an efficient simulation infrastructure that has been honed by active use for more than 10 years. It includes very fast processor simulators, optimized target memory handling, and a proven API for device modeling. All Simics models have transaction-level interfaces, so they communicate via transaction not signals.

Table of Contents

Modeling a Target System

Modeling a system in Wind River Simics is an iterative process, where the aim is to start with a minimal but still testable system and build out from there. During the modeling process, you gather more and more information on the requirements of the model and use it to drive modeling to the right areas of the overall platform. It is typical to model only the functionality needed to fulfill certain use cases, not all the hardware devices in a system and all its functionality. The goal is to run a certain software stack on the model. Over time, the platform model will become more complete as more functions and devices are added, making more use cases available. Using a combination of unit tests and system tests with the target software, the correctness of the model is progressively proven.

Note that even when the eventual goal is a model with complete device and functionality coverage for a piece of hardware (e.g., when bringing a new system-on-chip to market), the model can be built iteratively and incrementally. Key customers can be seeded early with the functions they need, and the ecosystem players such as operating system vendors can start porting operating systems long before the model (or the underlying design) is complete.

Simics models tend to follow the structure of the physical hardware closely, while abstracting functionality where possible. The overriding goal is to ensure that the software developed on the simulated system will run on the physical hardware and vice versa. Thus, all software-visible functions have to be accurately represented in the simulation, and the easiest way to ensure this is to design the simulation model using the same component and communications structure as the physical hardware.

Figure 1 shows an outline of the process that takes place when modeling a new system in Simics. The thick arrows are the most common feedback loops in the system, where most of the iteration time is spent. The dashed arrows are paths that occur less often (but are just as important).

The following is a classic iterative software-development methodology, where the software being developed—the hardware model—is tested early and often to explore the precise requirements. It goes by many names, such as spiral model, agile, or test-driven development:

1. Collect information about the target system to get an idea of what needs to be modeled. Typically, this includes design specifications, programmer's reference manuals, and other relevant documents. For new hardware, it might involve interviewing designers and looking at early design documents.
2. Map the outline of the target system, creating an initial list of devices and processors that make up the system and how they are connected and grouped. Based on an analysis of foreseen system usage and software load, make a preliminary decision on the necessary level of modeling of each device. Can it be ignored or stubbed out, or does it need to be fully implemented?

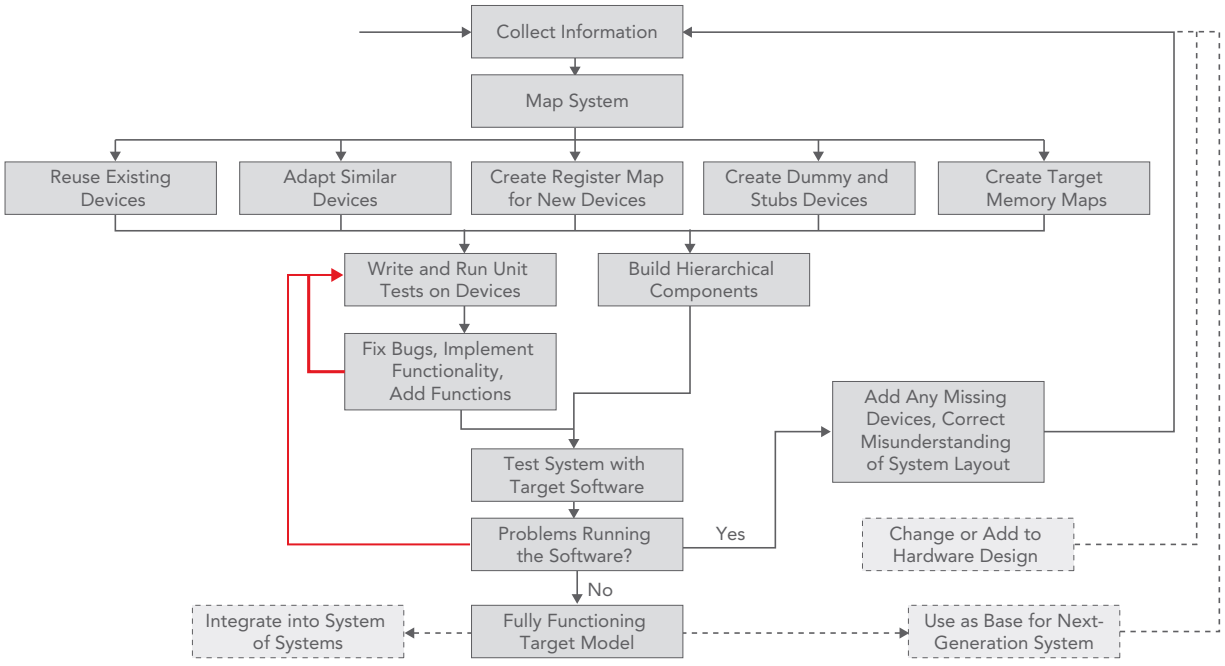

Figure 1: Modeling process

3. Where possible, reuse existing device models and processor models from the Wind River model library or your own existing simulation model library.
4. If a new device is similar to an existing device, adapt the existing device rather than starting from scratch.
5. Create simple dummies for devices that are not really needed.
6. Create initial models for the devices that need to be modeled, which usually involves creating the register maps and some values for important ID registers. This can be done very quickly using Wind River's device modeling language (DML) tool. Registers that are not yet implemented are marked and will generate warnings if accessed by the software.
7. Create the system memory map, indicating where devices are mapped in memory and any configurable options in the memory layout.
8. Create unit tests for any new or adapted devices. Tests will typically be designed from the reference manuals for the device. Test the units until they pass, before moving on to system-level tests with target software. The level of initial testing might vary between projects, depending on the complexity and completeness of modeling.
9. Create Simics hierarchical components, grouping the device models into logical subsystems. Also, create Simics start scripts that can set up a system model with software to run.
10. Test the system model with software. If there is real target software available, use it for testing. For new devices where no software yet exists, create independent test cases that exercise the specification of the software-hardware interface from the software side.
11. If testing reveals bugs in the functionality of a device, in all cases, correct the mistakes and test again. Bugs can include missing registers in devices, bad memory map setups, and other configuration issues.
12. If testing reveals access to unimplemented registers or dependencies on unimplemented functionality in devices, implement them and test again. Typically new unit tests are added before implementing the new functionality to allow local testing before going for a new round of system testing.
13. If testing reveals that new devices need to be added to the system, loop back to start and collect information on the new devices and go through the flow again (adding to the existing system model, not rebuilding it from scratch).
14. Iterate until the model runs the available software and passes all defined test cases.

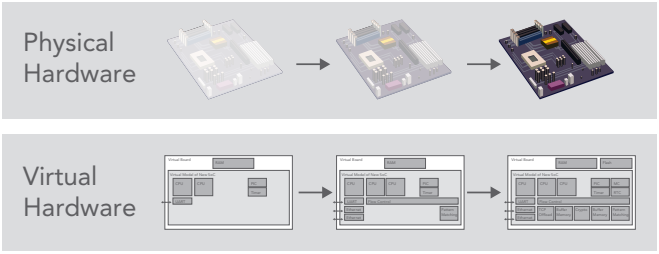

Figure 2: Evolution of a virtual platform along physical hardware

15. Once there is a complete model, it can be integrated to form part of a bigger system or used as the starting point for modeling the next generation of the hardware. The model is an asset that will be maintained and probably used for many years to come.

The iterative approach can be applied to the development and debugging of the hardware design itself, using the Simics model of the hardware as a design artifact. The Simics model can be modified to try ideas and check how things would work in various scenarios.

This modeling process can be applied to new hardware not yet released, existing hardware, or even obsolete hardware where physical availability is waning; it all depends on the nature of the final system being targeted. Simics has been used to model hardware in all stages of the product life cycle. It is worth noting that once a platform comes into use, it tends to be used for many years; so it pays to build a solid platform from the start.

Evolving Hardware Designs

When a virtual platform design starts before the hardware is complete, the hardware specification tends to change and grow over time. As shown in Figure 2, as the physical hardware design evolves and matures toward completion, the virtual platform adapts and aggregates more content. Thus, the virtual platform can be used as an executable specification of the hardware that can run software and be used to validate and evaluate the design.

During the hardware development process, the virtual platform is used to start software development early and to provide feedback from the software team to the hardware developers. This is an important value offered by a virtual platform that should not be overlooked.

Software Setup

Often it is possible to begin using a virtual system immediately after starting development. Even a basic system that does not yet contain all components can be used to get development started. A minimally useful configuration contains a processor, some memory, a timer (to let an operating system obtain periodic interrupts), and a serial

port for input and output. With this hardware, an OS kernel can be brought up using simulation back doors. A boot loader is developed much later in the process because it needs to configure the hardware, which requires a more complete model.

If the target software can be reconfigured to use a subset of the devices needed in the real system, it is best to start with the available devices and processors. The target system is created with a memory map containing the devices that are available. The target software is configured to use only the devices that are present. Missing devices are then added as they become available, and iterative development applies to each device in turn.

Another method is to use an existing system that is similar to the new system and then replace and add components as they become available. This requires the software to be somewhat flexible about the target hardware on which it runs. It is the natural strategy when developing models of new generations of systems that have already been modeled with Simics. It allows the hardware and software to be upgraded in parallel.

If the software exists and cannot be configured to use a similar system or a subset of system devices, it will be necessary to create initial device models of the system before attempting to run the software. Here, iterative development will be applied to all devices in parallel and the real target memory map will be used from the start.

Each individual device model can start out quite simple, implementing only the basic operation modes, and later, more complex optimized operating modes can be added. Over time, more devices and more details for each model will be added to the virtual system, evolving toward the final model. It is always possible to go back and improve the model thanks to the clear modularity of the Simics system and its robustness for partial models.

Chip-Level Modeling

A typical system is built from one or more boards, and on each board there are a few chips. The chips can be standard simple chips or systems-on-chip (SoCs), memory modules, or in-house application-specific integrated circuits (ASICs) and field programmable gate arrays (FPGAs). When building a full system model, each chip typically results in a modeling project by itself. In each modeling project, the chip is broken down into components, the memory map determined, and modeling done following the process outlined previously. With the hierarchical components system, described later in the "Anatomy of a System Model" section, each subsystem can be neatly encapsulated and created, tested, and integrated in isolation.

From a software perspective, it makes no difference whether a device is part of a standard SoC or a custom FPGA or ASIC. The modeling is done the same way, using the same tools.

For in-house designs, there might be an opportunity to reuse existing algorithm models or simulation models to speed the modeling project, as discussed later in the "Model Implementation Languages" section.

System-Level Strategy

In large and complex target systems containing multiple boards (or subsystems), it is necessary to decide the implementation order of the boards. To get to a working model that can run some meaningful software as quickly as possible, the best strategy is to follow the boot order of the system. This means that the first board or subsystem to model is the one that takes charge during power-on of the system, and that other system components are modeled later.

This is especially relevant when modeling new hardware because the software groups will want to start working with the system boot first. It is the most hardware-dependent so access to virtual hardware is the most pressing.

For example, in rack-based embedded systems there is usually some controller board or master board that boots first and then provides software for data plane boards and slave boards in the rack. This controller board has to be modeled first, as the other boards are not of much use without it. The same pattern occurs in other places, such as digital signal processor (DSP) boards, where a controller processor sets up software for the DSPs before booting them. Here, too, modeling the component that boots first as the first step is the sensible choice.

Model Creation

Wind River is often asked about the skill set needed to create virtual platform models. Is it most appropriate to task a hardware or software developer with model creation? In general, the best candidates are software developers familiar with device drivers and firmware. These developers are familiar with the programmer's reference manuals, and they understand how the software interacts with the hardware. If hardware engineers are tasked with the work, it is important they are strong software programmers because creating models is fundamentally about writing software. Furthermore, it is important to be aware that Simics models should not reflect the implementation details of the hardware, only its functionality. This is often a new paradigm for hardware developers.

TLM Modeling of the Hardware-Software Interface

The Simics approach to system modeling is to focus on the hardware-software interface and model whatever is needed to make the software side perceive the virtual hardware as just another piece of hardware, using transaction-level modeling to make the model fast. As shown in Figure 3, the hardware model consists of the same units as the physical hardware.

The functionality of the virtual platform is implemented using transaction-level modeling (TLM). In TLM, each interaction with a device (e.g., a write to or read from the interface registers of the devices by a processor) is a single simulation step: The device is presented with a request, computes the reply, and returns it in a single-function call. This is far more efficient and easier to program than modeling the details of how bits and bytes are moved across interconnects, cycle by cycle. A TLM model is much easier to write than a register transfer level (RTL) or cycle-level model and will run many orders of magnitude faster.

Modeling the behavior at the hardware-software interface and leaving the implementation of the hardware behavior to the modeler has several advantages:

- The hardware-software interface is often the best documented and most stable layer in the system. Since hardware design and software design are usually done by different teams, documentation is necessary. It is an exposed interface in the physical machine and therefore considered external by the hardware group.
- It is the natural integration point between hardware and software engineering teams and between companies producing hardware and their customers building software.
- Device models can often be constructed with only the details provided by a programmer's reference manual because it specifies what the hardware does from the software perspective. Detailed documentation on the actual implementation is typically not needed. Documentation at the bus signaling level is not needed because it is too low-level for a high-performance virtual platform.
- Modeling times are minimized because only the functionality actually used by the software needs to be implemented. Low-level hardware implementation details, such as those defined by RTL, are not required, making modeling much simpler and faster. A TLM model of the software-relevant functionality can often be written in 1/50 to 1/100 of the time needed to create a full implementation.
- Simulation performance is optimal because the simulation functionality is implemented with the minimal amount of state change needed and with minimal timing details.

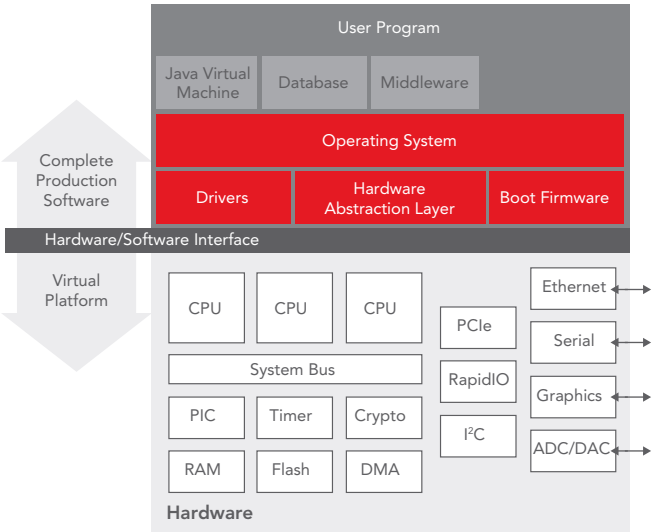

Figure 3: Modeling at the hardware-software interface

A good Simics model implements the *what* and not the *how* of device functionality. The goal is to match the specification of the functionality of a device and not the precise implementation details of the hardware. A good example of abstraction is offered by network devices. In the physical world, an Ethernet device has to clock out the bits of a packet one at a time onto the physical medium using a 5/4 encoding process. In Simics, this can be abstracted to delivering the entire packet as a unit to the network link simulation, greatly simplifying the implementation. As far as the software is concerned, this makes no difference.

A nice side effect of Simics-style modeling is that it is easy to reuse device models across multiple implementations of the device in question. As an example, the standard PC architecture contains a cascaded i8259 interrupt controller. Over time, the hardware implementation of the i8259 has changed from being a separate chip to becoming a small part of modern south bridges such as the Intel 6300ESB. Despite this huge change in implementation, the same Simics model can be used because the functionality exposed to the software is the same.

Time Management

All memory accesses in Simics are performed as synchronous transactions that pass through the entire hierarchy of memory maps, call a device function, and return immediately. Access to memory is generally assumed to take no time, even though it is possible to introduce models of caches and memory delays to perform analysis of how software interacts with the caches.

Often immediate completion of an operation when a device register access occurs is sufficient for modeling devices. When hardware units need to raise completion interrupts or change status registers after a significant time, events are used. The device model posts an event for some point in the future and then completes the current operation. When the time for the event comes, the device model gets a callback and it can set status bits, trigger interrupts, and complete work that should not be observed by the software until that time. Simics devices do not use threads to model time, only events.

It is easy to fall into the trap of modeling detailed aspects of the hardware that are invisible to the software. The execution overhead of modeling in too much detail can significantly slow the simulation. A simple example is a counter that counts down on each clock cycle and interrupts when it gets

to zero. This should not be modeled by calling the timer model each clock cycle. Note that the counter is only visible to the software when it is explicitly read. A better implementation is for the model to register a call-back, with the simulation kernel at its expiration time. If the software reads the register before this point, the model has to work out what would be in the register at that point by looking at the current simulation time. A model as such runs much faster than the detailed counter and is indistinguishable from it as far as the software is concerned.

Direct memory access (DMA) controllers are another example of abstraction. In Simics, DMA is typically modeled by moving the entire block of memory at once and delaying notification to the processor (or other requesting device) until the time when the full transfer would have completed on the physical hardware. The bus contention between the processor and the DMA controller is not modeled because this is not visible to the software. For a system architect with bandwidth concerns, a more detailed model can be created that logs the amount of data pushed, allowing bandwidth usage to be computed.

Common names in industry and academic discourse for the Simics TLM style are PV (programmer's view) or LT (loosely timed) TLM. Compared to the OSCI SystemC TLM 2.0 coding styles and abstraction levels, the Simics default level of abstraction corresponds to a streamlined form of LT, where device models never consume time inside a single transaction from the processor and never do wait. Timing annotations on memory accesses are typically assumed to be zero in Simics.

Interfaces Between Models

Simics is a simulation framework where all individual simulation objects (processors, devices, and everything else in the simulation) are treated as separate encapsulated units. They do not interact with the internals of other objects. Simics objects are created from classes, where a class might be a certain type of hardware, such as a serial port, or a feature such as a debugger connection. There can be multiple instances of every class: for example, several serial ports in a target. All classes are loaded from Simics modules at run-time, where modules are host system DLL or SO files. Modules can be loaded at any point during a run and objects created at any point. This is very similar to how dynamic loading and binding of objects works in environments such as a Java Virtual Machine or the common language run-time (CLR) in Microsoft .NET Framework. Code is only loaded when needed, and there is no static configuration of a target system.

To allow objects to communicate, Simics uses interfaces. Each interface is a collection of functions that an object exposes to other objects. A transaction on an interface is a function call from one object to another, and the function is expected to return immediately, just like all other transactions in Simics.

A Simics simulation object exposes an arbitrary set of interfaces, and objects can call any interface in any object. Interfaces are often looked up and configured as the simulation is starting up, but they can be looked up and the target of a connection changed at any point. It is even possible to call interfaces directly from the Simics command line, to quickly try things out or interact with objects from scripts.

Interfaces are used both to model hardware communications and to implement other simulator functionality and information flows, such as getting the current cycle count of a processor or finding the address of a variable from a debug module. Some interfaces are unidirectional, but bidirectional interfaces are common (e.g., a network and a network device sending packets to and from the device). Bidirectional interfaces are simply implemented as two complementary interfaces, one in each direction.

Simics comes with a large set of predefined interfaces, for interrupts, serial lines, Ethernet, I^2C, PCI, PCIe, RapidIO, MIL-STD-1553, memory operations, frequency setting, resets, and many other communications channels. There are interfaces for accessing debug information and interacting with processor models as well as for OS awareness.

Processor Models

Simulating at the Simics level of abstraction, the majority of the simulation execution time will typically be spent in the processor models running code. Processor models are highly optimized and tightly integrated with the simulator memory system to reach peak speeds of many billions of simulated instructions per second.

To run all the software of a system, the processor must implement all operating modes: user-level, kernel-level, hypervisor-level, secure execution, and any other modes found in a processor. The memory-management unit is also needed, along with model-specific registers and other low-level interfaces visible to the software in some operating modes. Fundamentally, anything readable or writeable from software has to be modeled.

Simics devices and processors do not usually attempt to model the precise cycle timing of code execution as dictated by the processor pipeline, cache hierarchies, and memory bus contention. Wind River processor models allow the speed of a processor model to be set by the processor clock frequency and the cycles-per-instruction (CPI) property of the processor. Typically, setting CPI to 1, that is, executing one instruction every cycle, is a good default. Setting CPI to other values such as 2 or 3/2 can help more closely approximate the average instruction execution rate of a processor for the software in a certain system. Experience shows that this works well for almost all workloads in most scenarios.

Processor models are created and supplied by Wind River, or by third parties using the processor API available since Simics version 4.0. In principle, there is no necessary speed penalty or feature loss for third-party processors using the processor API because the Wind River models use the same API for connecting to the simulator core. However, most third-party simulators tend to lack the features of Simics processor models, in particular, check-pointing, reverse execution, and the ability to run in a multi-threaded simulation.

Wind River provides a large library of fast and functionally complete and correct processor models for most common embedded and desktop architectures, including Power Architecture, MIPS, ARM, SPARC, and x86, in both 32-bit and 64-bit variants.

More Detailed Models

In addition to the default fast-simulation mode, Simics supports simulating a system at a greater level of timing detail:

- Simics can be run in a slower simulation mode where all memory accesses can be annotated with timing delays to account for memory access delays and the effects of cache systems.
- Detailed in-house models of processors can be used in a Simics system context, using the processor API.

Simics can use fast functional models to position a workload and then change to more detailed models for in-depth studies. In this way, much larger workloads can be efficiently handled than what is possible with only detailed models. When studying parts of a system in greater detail, other parts may be left at a functional level. This provides a faster implementation route to a model complete enough to run a real software load and also makes the execution itself faster.

Anatomy of a System Model

To understand how to model a system in Simics, it helps to break down the contents of a hardware system into six categories of models:

- **Processor cores:** The CPUs actually running processor instructions, for example, PPC 476, e500, Core 2 Duo, MIPS 5Kc, 80386, 68040, and ARM11
- **Interconnects:** Networks and buses connecting devices, machines, boards, and cabinets together, for example, on-chip memory buses, off-chip memory buses, serial, Ethernet, I²C, PCI, SCSI, USB, or MIL-STD-1553
- **Memory:** RAM, ROM, EEPROM, FLASH, and other types of memory devices that store large amounts of code or data
- **Devices:** All the peripheral units that move data between machines or do work that is not instruction processing; timers, interrupt controllers, ADC, DAC, network interfaces, I²C controllers, serial ports, LED drivers, displays, media accelerators, pattern matchers, table lookup engines, and memory controllers
- **Memory maps:** An abstract service in the simulator that implements how devices and memories are mapped into the memory space of a processor
- **Hierarchical components:** A logical structural layer of components to build chips, boards, and systems but not perform any simulation work at run-time

Figure 4 shows an example categorization of the hardware units found in a Freescale MPC8572E SoC design, along with some parts of a virtual board incorporating it. Note how the number of devices is much higher than the number of processors, interconnects, and memories. Memory maps and components are left out to make the picture simpler.

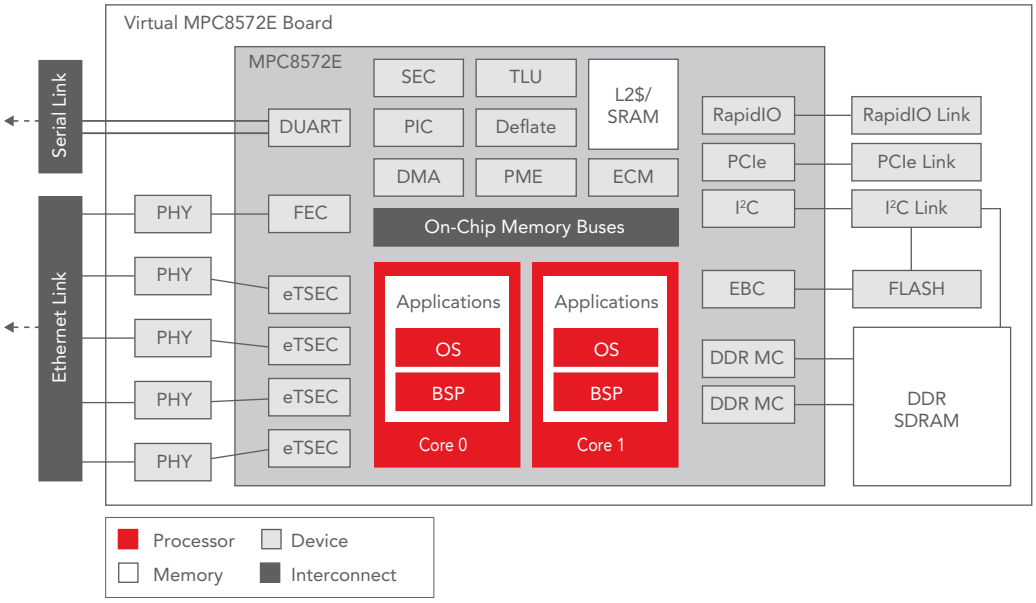

Figure 4: Example system breakdown into types of hardware units

In general, most of the work in creating a new system model is spent modeling devices, and they are the most numerous and least standardized of the hardware components. Most systems in Simics contain a large set of devices reused from existing model libraries, but there are always new devices in a custom system that have not yet been modeled.

In contrast to devices, processors tend to have fairly few variants. They are expensive to model because they have large instruction sets and are much more complex than most devices. Creating a truly high-performance processor is also hard work. Thus, most Simics users use the processor models provided by Wind River. It is also possible to integrate your own processor models or those available from third parties into Simics.

Like processors, memories are a special case. They need to be tightly integrated with the processor models to provide maximum performance and are very generic in nature. Simics provides a standard memory simulation model that supports demand-based paging, 64-bit addressing, zero-page detection, common-page detection, incremental check-pointing, reverse execution, and other features.

Interconnects are also comparatively few in variants and are standardized. Thus, they are normally easy to reuse across target systems. For Simics, they are usually provided by Wind River because a fully featured implementation is fairly complex. Interconnect models have to support distributed and multithreaded simulation as well as connections between virtual and physical networks and support for traffic record and replay for reverse execution.

Memory Maps

The memory system connecting a processor to memory and memory-mapped devices is a very special type of interconnect. In a physical system, it is implemented as a hierarchy of buses, memory controllers, bridges between buses, external bus controllers, and so on. In Simics, to enable high-speed simulation, the focus is on the memory map that is implemented by the bus hierarchy. Simics features a special simulation model called the memory map, whose only job is to route memory and device accesses to the right devices.

Figure 5 shows a small subset of a real memory map. The processor sends accesses to the memory map, which passes them on to device models, memory images, and other memory maps. Simics has highly optimized the way processors access memory. Accessing RAM, FLASH, and ROM is very fast via back-door accesses and does not involve any explicit device models or even the memory map in most cases. The memory handling is part of the Simics core and is exposed to processor models using the processor API.

A memory controller model in Simics exists in parallel with the memory itself. The memory controller model will implement functionality such as changing how memory is mapped by changing the memory map setup in the simulator, but it is not involved directly in regular memory accesses. For PCIe and similar interfaces where several levels of addressing are being used, Simics uses hierarchies of memory maps. This makes it easy to translate real-system mappings into the Simics system configuration. PCIe models in Simics support software probing and configuration. The software setup is then reflected in the contents of the PCIe memory map, and device accesses are as fast as if they were connected to the main memory map.

Setting up the memory map is a crucial part of Simics system modeling.

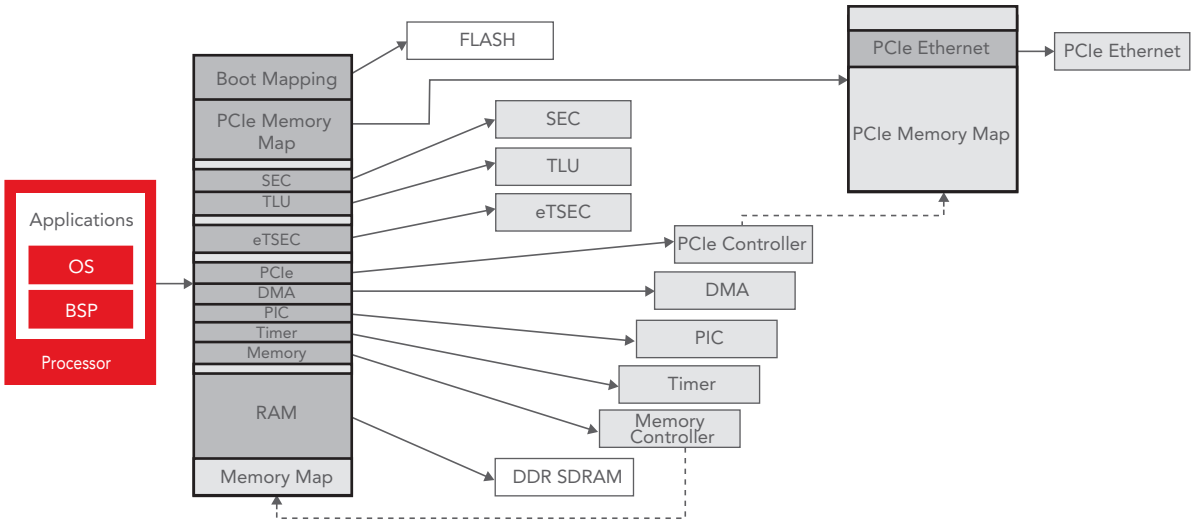

Figure 5: Example of a memory map

Hierarchical Components

In theory, a Simics virtual platform can be built from a collection of individual devices, processors, memories, interconnects, and memory maps, using Simics scripts to create and connect individual objects. However, users require something that is easier to use. Modelers need structures that support encapsulation of logically isolated components and the reuse of models. The model structure needs to reflect the structure of the physical hardware to facilitate maintenance and user recognition. Subsystems should be easy to instantiate in any number of copies without worrying about name clashes or connections accidentally connecting unrelated objects.

To support this, Simics features a hierarchical component system. A Simics "component" is an abstraction added on top of hardware device models, corresponding to the physical or logical grouping of devices into chips, boards, racks, networks, and systems of systems. A component corresponds to a physical unit such as a board, SoC chip, memory module, hard disk, or other unit that is typically found on a system block diagram. They can also correspond to useful logical groupings of devices that recur in multiple systems.

Figure 6 shows a somewhat abstracted example of a hierarchical component system that consists of two boards connected by an Ethernet network. Each board has a different type of SoC, along with memory, network PHY connectors, and an FPGA or a small computer system interface (SCSI) disk controller. The left-hand side shows the physical structure to model, and the right-hand side shows the corresponding Simics hierarchical components.

The fundamental simulation units of processors, devices, memories, and interconnects discussed previously are grouped into components corresponding mostly to the physical packaging. However, since the CPU complex in SoC 1 is used in several related chips, it is given its own component, to make it easier to reuse.

Simics hierarchical components encapsulate how devices are connected to each other inside the subsystem that they represent, including memory maps, interrupt routings, and other connections. Where the subsystem interfaces to the outside world, the connection points are brought out as explicit connectors between components. In the example, note how Simics memory maps are present at all levels of the component hierarchy because each component contains devices that need to be mapped into memory.

Furthermore, components expose parameters such as memory sizes, processor speeds, number of processors, and so forth, to the user. They offer a way to check constraints on value and to compute low-level implementation parameters from high-level system parameters. For example, configuring a DDR3 memory module for a certain size requires the component to compute the bank counts, rank sizes, and other parameters that are communicated via I^2C to the memory controller. The external configuration interface is just the size of the memory and an optional specification of the DDR (double data rate) module setup.

Typed connectors make sure that only components that can usefully talk to each other can be connected. Constraint checking is part of the connection process, to ensure that connections make sense.

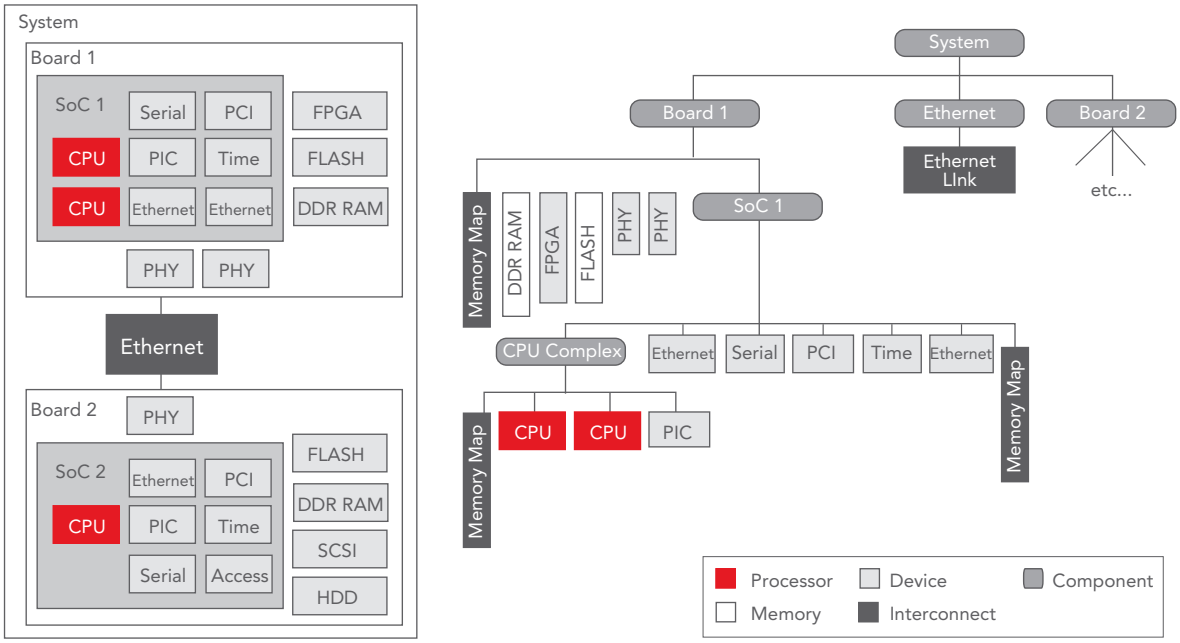

Figure 6: Example system component structure

Figure 7 shows how Simics components can be integrated into higher and higher levels of systems. The system in Figure 6 is now one rack in a system with two racks and a wide-area network. The top-level "system" in a Simics full system simulation will vary with the user and nature and scale of the target system.

Hierarchical components provide the Simics user interface with a hierarchical namespace, where the user can move around and interact with the devices local in each namespace. The hierarchical namespace makes it easy to write scripts that interact with a small part of a system regardless of the setup of the rest of the system. For example, in Figure 7, scripts and debugging can be focused on a single CPU core or just Board 1, ignoring what is going on in the rest of the system, even if the rest of the system is reconfigured during the run.

Extensions

In addition to the hardware models implementing the target system, Simics setups contain features such as the command-line interpreter, debugger connections, instrumentation and tracing modules, and fault injection. Collectively, such functions are known as extensions because they extend the functionality of Simics. Extensions are implemented using Simics objects, classes, and modules, just like hardware models. Extensions can be loaded and taken into use at any point during a simulation run. They typically use a wider set of Simics APIs than basic device models, allowing more control and inspection into the simulation.

Extensions can be models of interactive devices such as screens, serial consoles, keypads, and blinking LEDs. In Simics, the extensions are written separate from the device model. This encapsulation allows the device model to be reused across different target systems. Simics standard interfaces for network, serial, and other connections are used to communicate between user interface objects and target hardware models.

The simulation can be extended to include features of the physical process that the target system controls. There can be integrations that allow Simics to co-simulate with other simulators, such as cycle accurate simulators, RTL simulators, or even physical hardware.

Anatomy of a Device Model

As stated previously, most of the work involved in modeling a hardware system is in creating the device models unique to that system. To allow the embedded software to run completely unmodified, the system model has to simulate the properties and behavior of the hardware, such as programming registers and device functionality. The devices in the system typically show up as banks of control registers in certain locations in the processor memory map. The software's device drivers read and write to these addresses. These are the memory-mapped programming registers shown in Figure 8.

A register map can be very simple, such as the eight 1 byte registers used to control an NS16550 serial port. It can be spread out in several parts across the memory map of the processor, such as a PCI device that maps both a configuration space and a memory space. The register map can be very large, such as that of a Freescale MPC 8260 CPM that has thousands of registers. Simics includes features that make specifying and implementing large and complex register mappings easy and fast.

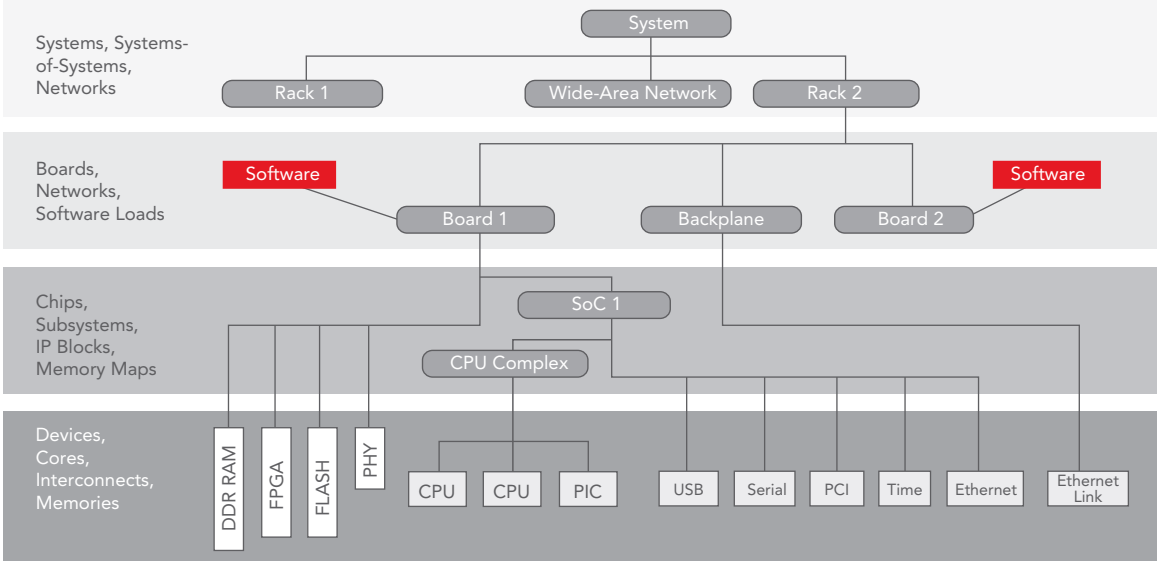

Figure 7: Example system hierarchy

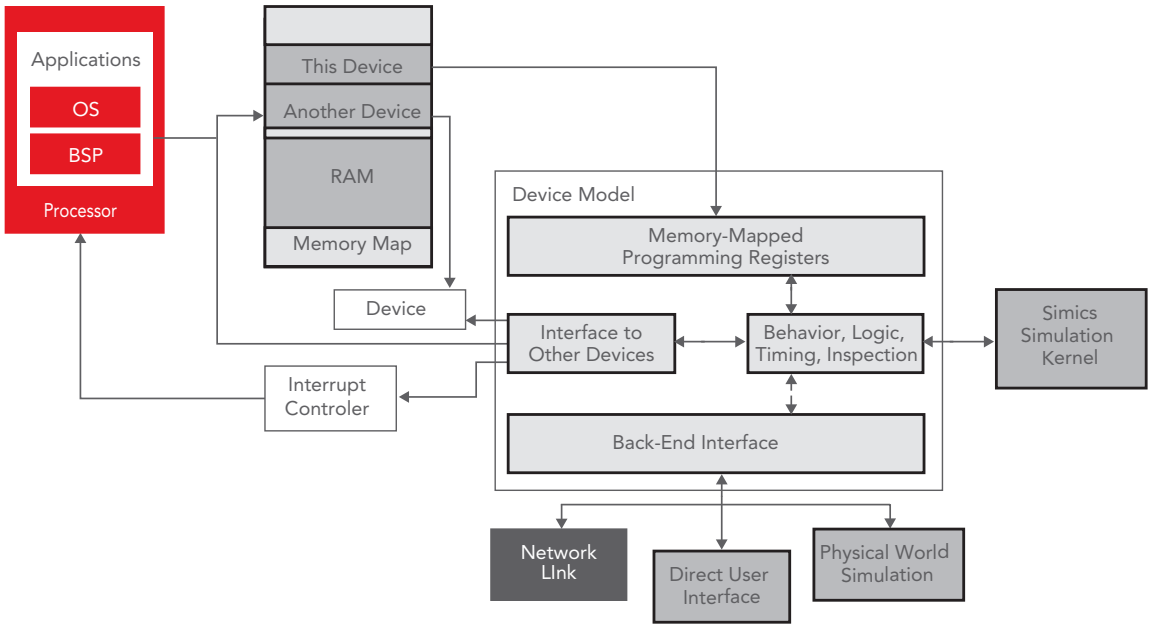

Figure 8: Interfaces of a device model

Along with the register description, the behavior of the device model has to be defined. At a minimum, there needs to be enough device functionality written so that the software will operate correctly. Simics device models are passive objects that only act when something happens. Their behavior is coded in a reactive, event-driven way. Most of the work is performed in callbacks. The interaction with the simulation kernel takes place from the behavioral part of a model, as shown in Figure 8.

In most cases, there will also be a back end to the device, where interactions with the environment are handled. This is implemented by having the device model call interfaces in other objects in the simulation. For example, a screen device needs to display what is being drawn on the host machine, and a network device needs to send and receive packets. A sensor needs to pick up values from the simulated world to report readings. The norm is to keep direct user interaction encapsulated in its own object or set of objects so that device models are independent of the details of the user interface implementation. This indirection makes it easy to record and replay user actions and makes sure that the user interface is consistent regardless of which model is implementing the hardware behavior.

Devices often interact with each other in the system. There can be direct memory access to read and write memory, interrupts raised toward a processor, or direct data transfers to tightly coupled devices. For example, a multiprocessor system controller will need to route interrupts from devices to processors and pass interrupts between processors in the system. Network processing accelerators have direct queues or channels connecting to the network interfaces of an SoC where network frames go directly without touching system memory. All such operations are modeled using interfaces in Simics and implemented as discrete events, not continuous processes.

Model Implementation Languages

DML, explained in the following section, is the recommended programming environment for new device models in Simics; however, any language that can call the Simics API can be used. Simics comes with support for C, C++, Python, and SystemC. Python can be used to prototype models that do not require high performance at a higher level of functionality. C and C++ are used for code that needs to implement very complex algorithms and for integration of existing models into Simics.

Using hardware-implementation-level descriptions such as SystemVerilog, Verilog, and VHDL is possible by integrating Simics with RTL-level simulators; but it will result in very slow simulations. Simics models communicate at the transaction-level modeling (TLM) abstraction; therefore, they cannot be automatically derived from cycle-level descriptions of the hardware.

Device Modeling Language

An important enabler for the Simics methodology and speed of model development is the DML tool. DML is a helper system designed to speed TLM device modeling for Simics. It provides a concise syntax and language constructs specifically targeting the device modeling problem at the Simics TLM level of abstraction.

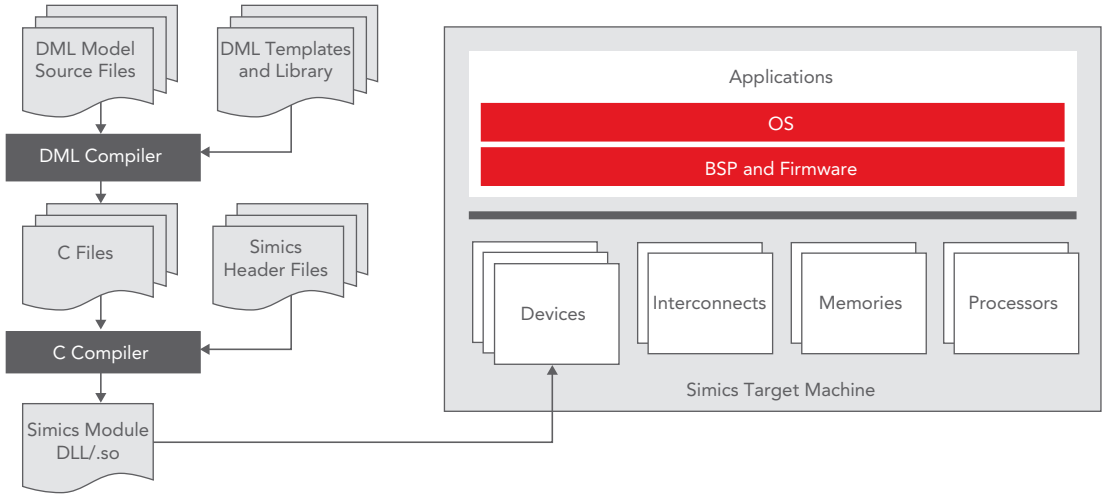

Figure 9: DML compilation

The DML compiler generates C code that includes calls to the Simics API, with full support for Simics features such as check-pointing, logging, and reverse execution. This process is transparent to DML users. The compilation flow is shown in Figure 9.

DML models are much more concise and more domain-specific than corresponding C language models. Therefore they take less time to write, read, understand, and maintain. DML has explicit support for the iterative development of device models by providing default implementations for many device register behaviors, including marking registers as "dummy" or "unimplemented." DML puts the documentation of a model inline in the source code, making sure that documentation is written along with the model implementation.

The most visible part of DML is the specification of the programming register interface of a device. This interface is described as a series of register banks. Each register bank is later mapped to memory locations using the Simics memory map system described previously. Thus, DML devices are completely position-independent and can be instantiated any number of times in the target system with no change needed to the model itself.

From the start of the bank, registers are specified with their size and offset. A register can be from 1 to 8 bytes and may be divided into fields consisting of 1 to 64 bits. Simics allows you to create multiple copies of a device mapped at different locations.

DML explicitly specifies the endianness of a register bank and insulates the programmer from the relationship between the host and target endianness. The same model will run on both little-endian and big-endian hosts without modification, with the same behavior and the same bytes put in the same places in simulated memory. Bit fields inside a register can be specified to use big-endian or little-endian bit numbering. This allows the specification of fields to follow the programmer's manual for a device regardless of the bit-ordering convention used by the device supplier (in particular, most Power Architecture–related device manuals use big-endian bit order while devices used with x86 and most other processors use little-endian bit order).

Functionality is attached to a bank, register, or field by defining methods that are called when a read or write is performed on the device. To cut down on repetitive coding of registers and other functions within a device, DML provides a template feature that allows bundles of functionality to be reused. Such templates can describe the behavior of a register, field, or other part of a DML model.

DML supports the definition of new Simics interfaces, implements existing interfaces, and uses interfaces to call into other objects. It supports using Simics events for timed actions and the definition of reset behavior for a device.

The functionality of a model is programmed as small sequential snippets of code expressed in C-style syntax. Any programmer familiar with a language such as C, C++, SystemC, Java, or Python can quickly get started writing device models. There are some additional language features to support device modeling tasks such as parsing data structure layouts in memory and handling bit slices. The DML compiler automatically generates the code required to activate each sequential snippet, removing the complexity of sequencing and reacting at the right time from the model source code.

Wind River provides a set of templates and base files to simplify the creation of device models attached to interfaces such as PCI, PCI Express, USB, I^2C, Ethernet networks, and serial lines. There are also several example devices provided with Simics, and usually the device source code for a target is provided to users of that target.

C, C++, and Python

When writing device models in C, C++, and Python, a programmer has to use the Simics API directly. C code accesses the basic API. C++ models can use a C++ library that exposes the Simics API in an object-oriented form suitable for native C++ code. A translation layer uses the dynamic nature of Python to simplify access to the Simics API and manipulate Simics data types such as lists.

As shown in Figure 10, C and C++ code is often used to provide the back-end functionality in a device model, with DML providing the register map and Simics interface. This makes it possible to reuse existing algorithm models and simulation models in a Simics model, without having to recode anything in DML. The C/C++ code is often shared between Simics and other simulation environments. For very complex device functionality, it might be easier to implement the functionality in C++ than in DML.

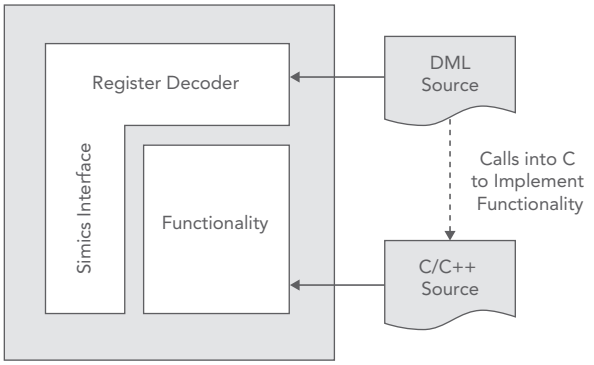

Figure 10: C and C++ for device functionality

SystemC

SystemC models can be used in Simics thanks to the Simics SystemC bridge. This allows the reuse of existing SystemC models to quickly co-simulate with a functional system model in Simics. As illustrated in Figure 11, SystemC is integrated into Simics by including a standard OSCI SystemC 2.2.0 kernel as a subsystem along with the SystemC device models. The SystemC bridge takes care of coordinating and synchronizing the activities of the SystemC kernel and the Simics kernel and moving transactions between the SystemC subsystem and the rest of the Simics system.

As long as the SystemC models have TLM 2.0 interfaces, there is no need to modify the SystemC models to work inside Simics. Using the Simics C++ API makes it fairly straightforward to add Simics integrations to SystemC code, which allows SystemC devices to expose more features to Simics users.

For best performance, the SystemC models integrated should be transaction-level models that are event-driven just like native Simics models. Simics supports the check-pointing of SystemC models if they expose their simulation state for saving and restoring.

IP-XACT

Simics can import and export IP-XACT descriptions of device register maps, turning the register maps into DML skeletons and exporting the registers coded in DML into IP-XACT. This makes it possible to reuse register design information from hardware design tools and avoid manual recoding of existing

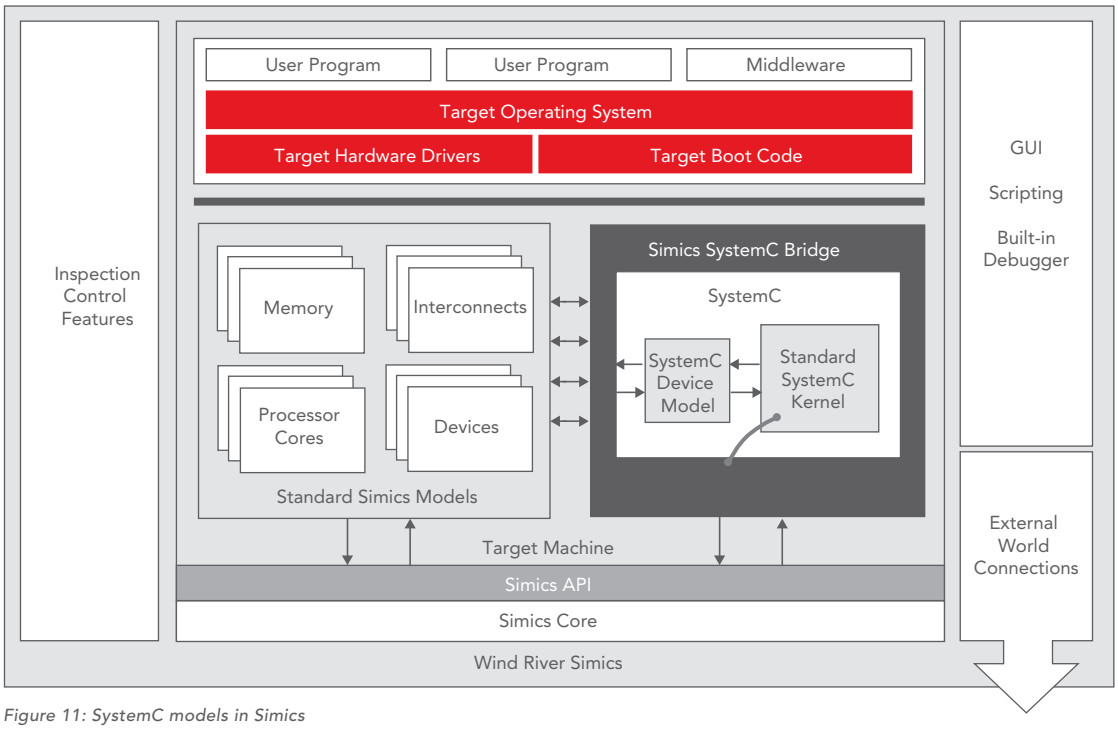

Figure 11: SystemC models in Simics

information. Consistency is ensured by using a single description of registers for all tools involved in a hardware design, reducing the risk of bugs.

Other Models

Any C or C++ code can be used in Simics, with the addition of calls to the Simics API. Over time, many different simulation models built for many different simulation systems have been integrated with Simics. As long as models are transaction-level, it is reasonably straightforward to integrate them with Simics. Several in-house simulation frameworks have been integrated with Simics as well as plain C/C++ simulators for hardware functionality.

This applies to models described in model-driven architecture (MDA) tools such as Matlab and Labview as well. It is possible to use algorithms described in these tools as the core part of a Simics device model, typically with a DML wrapper to describe a register map and handle the integration with the Simics API. As long as you can get a C language function call through to a piece of code, it can be integrated into Simics. Another example is the embedding of a full Java Virtual Machine inside of Simics to run Java code as device models.

Conclusion

Wind River Simics enables improvements in systems and software development by using virtual platforms rather than physical hardware for many development tasks. Simics is a fast transaction-oriented simulator that can model the functional interface of any hardware. Typically, Simics users need to construct models of their custom hardware to get the full benefit of Simics. This paper describes the methods, design principles, and supporting tools that Wind River provides to make modeling fast and efficient.

Building Simics TLM models takes orders of magnitude less time than building the actual hardware implementation of a chip or a device, and the models run many orders of magnitude faster than simulation of RTL. Simics models are developed in an iterative and agile way that ensures they are useful as early as possible in the development process, and that minimizes model development costs. Existing design assets such as register maps and algorithm kernels can be used to build Simics models, and Simics can also include existing models in SystemC and other languages as is.