Publisher
Richard Bowles

Managing Editor
Stuart Douglas

Content Architect
Daniel Aarno
Jakob Engblom

Program Manager
Stuart Douglas

Technical Editor
David Clark

Technical Illustrators
MPS Limited

Technical and Strategic Reviewers
Daniel Aarno
Jakob Engblom

Intel Technology Journal

ISBN 978-1-934053-62-1, ISSN 1535-864X

Intel Technology Journal
Volume 17, Issue 2

Publisher: Richard Bowles
Managing Editor: Stuart Douglas

Library of Congress Cataloging in Publication Data:

Printed in China
10 9 8 7 6 5 4 3 2 1

First printing: October 2013

INTEL® TECHNOLOGY JOURNAL
SIMICS UNLEASHED – APPLICATIONS OF VIRTUAL PLATFORMS

Articles

Foreword Simics*—The Early Years

Peter S. Magnusson,
Engineering Director, Google, Inc.

When I interviewed for an internship at the Swedish Institute of Computer Science (SICS) in 1991, the project was to write a parallel-computer simulator for the Data Diffusion Machine (DDM) research effort being led by Seif Haridi and Erik Hagersten. I was hired by Andrzej Ciepielewski and Torbjörn Granlund and the project was supposed to take six weeks. It took a little longer than that.

Back in the 1980s, it was common for computer architecture research to be entirely based on simulations running computationally intensive workloads—traditional high performance computing. The best practice in the field was summarized by the release of the Stanford Parallel Applications for Shared Memory (SPLASH) benchmark suite—which coincidentally also occurred in 1991.

However, at the time (late 1980s, early 1990s), a number of research groups recognized that much of parallel-computer usage was not compute-intensive as much as it was data-intensive—for example, transactional workloads, better represented by benchmarks like TPC-C. But these workloads were generally commercial software, large parts of which were only available in binary. They also heavily relied on the underlying operating system.

So a project was conceived to develop a simulator to both support the computer architecture work around the DDM project and also support porting an operating system to the prototype. An existing, groundbreaking simulation environment developed by Robert Bedichek at the University of Washington was extended to support a multiprocessor system and to mimic real devices.

(As a curious aside to the reader, Robert's work on simulation began at his time at Intel in the late 1980s, so Simics now being an Intel product closes the loop.)

The six weeks grew. Some six calendar years, twenty man years, and several hundred thousand lines of code later, in 1997, the simulation group in the Computer and Network Architectures (CNA) group at SICS finally succeeded in the original goal: booting a commercial operating system (Solaris* 2.6) on a simulated Sun Microsystems server (sun4m architecture). This was the first known occasion of an academic group running an unmodified commercial operating system in a fully simulated environment. The "full system simulator" was born.

The simulation group at SICS eventually grew to five people, all of whom became founding employees of Virtutech in 1998: Magnus Christensson, Fredrik Larsson, Peter Magnusson, Andreas Moestedt, and Bengt Werner. Our first customers were Sun Microsystems, Ericsson, and HP. To the original SPARC* V8 architecture, we added SPARC V9, x86, x86-64, Power, ARM, Itanium®, and so on. We invented a number of new technologies and tools along the way, making Simics by far the most capable tool in its field.

With the launch of Simics 3.0 and the Hindsight* technology in 2005, all the core elements that I remember scoping out on a whiteboard around 1993 were in place, and several I hadn't imagined. So in some sense, it became a software project that literally took over 100 times longer than originally planned.

In the process I became convinced (and still am) that this is by far the best way forward to improve software development environments, since, once inside a deterministic simulator, you can do some very interesting things.

SIMICS* OVERVIEW

Contributor

Daniel Aarno
Software and Services Group,
Intel Corporation Line break
Jakob Engblom,
Wind River

This article provides an overview of Wind River Simics*, a full-system simulation framework jointly developed by Intel and Wind River. Simics technology has been used to help develop complex software and hardware systems for more than two decades. This technical overview describes what Simics is, its main design goals and principles, and how it works. The article also describes the overall simulation landscape, and how Simics fits into the big picture.

Introduction

"A full-system simulator (FSS) like Simics is a model of a digital system that is complete enough to run the real target's software stack and fast enough to be useful for software developers."

A full-system simulator (FSS) like Simics[7] is a model of a digital system that is complete enough to run the real target's software stack and fast enough to be useful for software developers. The speed and full-system simulation capabilities of Simics differentiates it from most simulation tools provided by the electronic design automation (EDA) industry[8], which are typically extremely accurate from a hardware perspective, but too slow to be practical for operating system (OS), application, or systems software.

In an FSS, there are models of processors, memories, peripheral devices, networks, and so on, making up a model of the *target machine*. The key goal of the simulation is that as far as the software running on the target is concerned, it could just as well be running on physical hardware. Often this means that the simulation solution includes more than just the computer components. The simulation also integrates various simulators for the external environment that the computer system is operating in.

"The main users of Simics are software and systems developers, and their main problem is how to develop complex systems involving both hardware and software."

The main users of Simics are software and systems developers, and their main problem is how to develop complex systems involving both hardware and software. Virtual Machine Monitors (VMMs) like VMWare* or Virtualbox* also run complete software stacks—but for a runtime use case, not for the complete product lifecycle. In addition a VMM only simulates a generic, simplified hardware platform, whereas Simics can ensure binary compatibility with an actual real-world machine such as a specific Intel chipset (PCH) and processor variant.

Target Systems

The target systems simulated with Simics range from single-processor aerospace boards to large shared-memory multiprocessor servers and rack-based telecommunications, data communications, and server systems containing thousands of processors across hundreds of boards. The systems are often heterogeneous, containing processors with different word-length, endianness, and clock frequency. For example, there can be 64-bit Intel Architecture (R)

processors running control software, alongside 8-bit microcontrollers managing a rack backplane, talking to data processing boards containing dozens of 32-bit VLIW DSPs. The target systems are typically built from standard commercial chips along with some custom FPGAs or ASICs.

Often, target systems are networked. There can be networks of distinct systems and networks internal to a system (such as VME, I^2C, PCIe, and Ethernet-based rack backplanes). Multiple networks and multiple levels of networks are common.

Simulation runs can cover many hours or days of target time and involve multiple loads of software and reboots of all or part of the system. Even a simple task such as booting Linux and loading a small test program on an eight-processor SoC can take over 30 billion instructions. Profiling and instrumentation runs can take tens of billions of instructions.

"Simulation runs can cover many hours or days of target time and involve multiple loads of software and reboots of all or part of the system."

Simics can be used to model future processors and chipsets well in advance of hardware availability. Such "early hardware" deployment of Simics allows BIOS, OS, and application software development to be performed long before even prototype silicon is available.

"Simics can be used to model future processors and chipsets well in advance of hardware availability."

Simics is often used with models of hardware that are also available in silicon. Some models started life as early hardware models, and others have been created after the hardware was commercially available in order to directly support the main software and system development effort.

It is not uncommon for Simics to be used to model old hardware. Many embedded systems have lifespans covering decades, and development boards and tools tend to become exceedingly scarce over time. In such circumstances, Simics can provide an easily accessible, convenient, and available tool to keep up the maintenance of software for the systems.

Simics Use Cases

Full-system simulation can be applied during the complete product lifecycle as shown in Figure 1. It helps to define systems, by providing an executable model of the hardware interface and hardware setup. FSS supports hardware and software architecture work, and it validates that the hardware can be efficiently

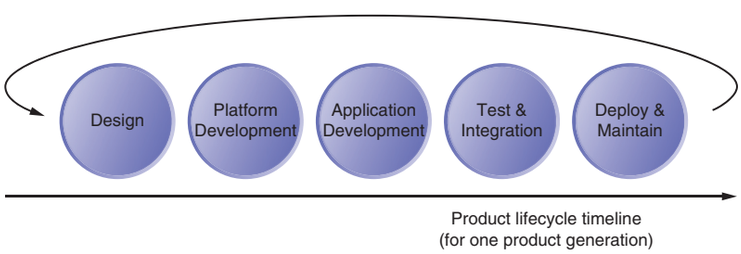

Figure 1: Product lifecycle
(Source: Wind River, 2013)

"The software development schedule can be decoupled from the availability of hardware…"

used from the software stack. FSS is used to develop system software, including debug and test. The software development schedule can be decoupled from the availability of hardware when using FSS and it improves software development productivity by providing a better environment than hardware.

Figure 2 illustrates the concept of "shift-left", where software, drivers and BIOS development, integration, and test efforts are performed much earlier in the development process. This not only reduces products' time to market, it also reduces the cost of fixing defects when discovered earlier in the product lifecycle and increases product quality and customer satisfaction.

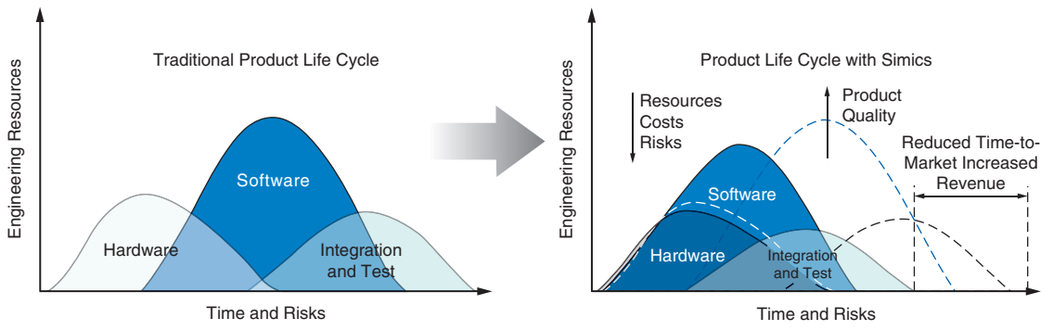

Figure 2: Shift-left of the product lifecycle (Source: Wind River, 2011)

The article "Using Virtual Platforms for BIOS Development and Validation" by Steve Carbonari describes the development of BIOS code on Simics models in advance of hardware availability, as well as how Simics is being used after silicon becomes available.

The article "Post-Silicon Impact: Simics Helps the Next Generation of Network Transformation and Migration to Software Defined Networks (SDNs)" by Tian Tian describes a high-level view of how Simics has been used for early hardware access for Intel communications chips, developing software stacks and drivers.

The article "Early Hardware Register Validation with Simics" by Alexey Veselyi and John Ayers describes a lower-level use case, where Simics is used to validate the register design of hardware very early in the design process.

"Simics can add and remove boards, bring new processors online, reconfigure network topologies, introduce faults in networks and hardware devices, and plug and unplug hot-pluggable hardware."

A particular use of Simics is to change the simulation target during a simulation run in order to test software behavior. Simics can add and remove boards, bring new processors online, reconfigure network topologies, introduce faults in networks and hardware devices, and plug and unplug hot-pluggable hardware. The software will perceive these events like it would on physical hardware, allowing users to test and develop all aspects of the software stack, including automatic configuration, load balancing, fault detection, and fault recovery.

Simics can be used in tasks outside the immediate realm of development and engineering. For example, Simics has been used to demonstrate new products to

prospective customers and to procurement agencies involved in large programs. Simics is also a training tool, both to train and educate users in general concepts (using Simics instead of hardware to make the training more efficient), and to train users of particular systems (typically developed using Simics to begin with).

As a system matures and the next generation begins development, Simics can be used to smoothly move from the current generation to the next generation. By setting up a model containing a mix of old and new hardware components (such as different generations of boards in a rack-based system), software can gradually be updated to match the next hardware generation. As part of this process, new boards can be tested in a system containing existing legacy boards. This is represented by the arrow back to the start in Figure 1.

"…Simics can be used to smoothly move from the current generation to the next generation."

Important Features of Simics

The feature set of Simics has been developed and adjusted for more than twenty years in order to meet the needs of system developers (the first code in what was to become Simics was written in 1991). In this section, we describe the most important Simics features and why they were designed into the product.

Run Unmodified Real Software

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 managed to run most types of software, including hypervisors with guest operating systems, small MMU-less embedded operating systems and bare-metal code, desktop and server operating systems like Windows* and Linux*, and real-time operating systems (RTOS) like VxWorks*.

"A key design goal of Simics has always been to run the real software stack,…"

Running real unmodified software stacks has many benefits. Since 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 or variant builds for simulation and quick tests.

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 be portions of the system where only machine code is available, such as proprietary libraries, drivers, or operating systems, and in such cases running the real binary code is the only way to get a complete software system running.

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.

The article "Using Virtual Platforms for BIOS Development and Validation" mentioned earlier describes how Simics is used to develop, test, and debug

low-level BIOS code, which is probably the most difficult type of software to run on a simulator.

The article "Using Simics in Education" by Robert Guenzel describes how Wind River makes use of the ability to run unmodified software to train users in topics like device driver development and network management.

"Simics is modular; each device model, processor, or other Simics model or feature is shipped in its own self-contained dynamically loaded object file…"

Modularity

Simics is modular; each device model, processor, or other Simics model or feature is shipped in its own self-contained dynamically loaded object file (as shown at the bottom of Figure 3). This fine-grained structure makes it possible to supply the exact set of models and features needed for any specific user. The object file and its associated command files are referred to as a Simics *module*.

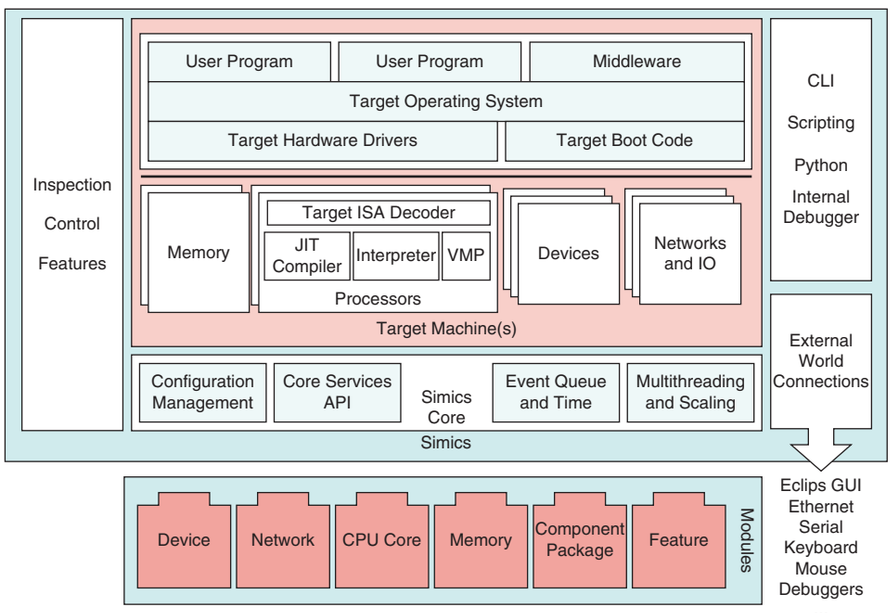

Figure 3: Simics architecture
(Source: Wind River, 2011)

"Simics models can be distributed as binary-only modules,…"

Simics models can be distributed as binary-only modules, with no need to supply source code to the users. Binary distribution simplifies the installation for end users, as they do not have to compile any code or set up build environments. It also offers a level of protection for intellectual property when different companies exchange models. By obfuscating the names of hardware registers and limiting the amount of metadata included in the modules it is possible to safely distribute models of very sensitive future hardware designs to external users. It makes it possible to limit the information disclosure by the model to precisely that of the documentation provided, even if the model itself needs to contain undocumented and secret registers to make BIOS and low-level firmware code work correctly.

Simics modularity enables short rebuild times for large systems, as only the modules that are actually changed have to be recompiled. The rest of the simulation is unaffected, and each Simics module can be updated and upgraded independently.

A Simics model exposes an arbitrary set of *interfaces* to other models in other modules, and objects can call any model interface in any module. Interfaces are used both to model hardware communications paths 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 the debug module. Unlike SystemC*, an object can implement an interface multiple times using named ports and the bindings are not made at compile time. Some interfaces are unidirectional, but bidirectional interfaces (like network send and receive) are common and simply implemented as two complementary interfaces, one in each direction.

Simics uses the C-level ABI and host operating system dynamic loading facilities. The C++ ABI varies between compiler versions and compiler vendors, and is thus not usable in the interface between modules, even though C++ can be used internally in modules. The Simics framework provides bindings to write Simics modules using DML (see below), Python, C, C++, and SystemC, but users can actually use any language they like as long as they can link to C code. For example, a complete JVM has been integrated into Simics, running modules written in Java.[1]

Scalability

As discussed above, Simics target systems can potentially be very large. To efficiently simulate such large systems, Simics makes use of several techniques which are described in more detail in the section "Simics Performance Techniques." Scalability has been an important attribute of Simics since the very first commercial deployments, originally relying on distributed simulation[7], and evolving into a multithreaded (and distributed) implementation.[8]

The article "Simics on Shared Computing Clusters: The Practical Experience of Integration and Scalability" by Grigory Rechistov describes a use case where Simics was scaled up and scaled out to run a simulation of more than one hundred Intel® Xeon® server boards, containing 1792 target processors.

"…Simics was scaled up and scaled out to run a simulation of more than one hundred Intel® Xeon® server boards, containing 1792 target processors."

Multiple User Interfaces

From the very beginning[7], 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 command-line and GUI interfaces are needed. Today, the primary user interface for new users to Simics is the Eclipse-based GUI, but the command line is still there for more advanced tasks. Figure 4 shows a screenshot of the Simics 4.8 Eclipse GUI, running two simultaneous, but separate, simulation sessions (clockwise from top-left: simulated serial text-terminal, Simics Eclipse GUI, simulated graphical console).

"From the very beginning[7], Simics was designed as an interactive tool that could also be used in automated batch runs."

"…the Simics architecture separates the function of the target hardware system from the connections to the outside world."

"…Simics opens up a network connection from the virtual network inside of Simics to the host machine or other machines on the network."

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 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 3, the Simics architecture separates the function of the target hardware system from the connections to the outside world. The target consoles shown in Figure 4 are not part of the device models of the serial ports and graphics processor unit, but rather 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 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.

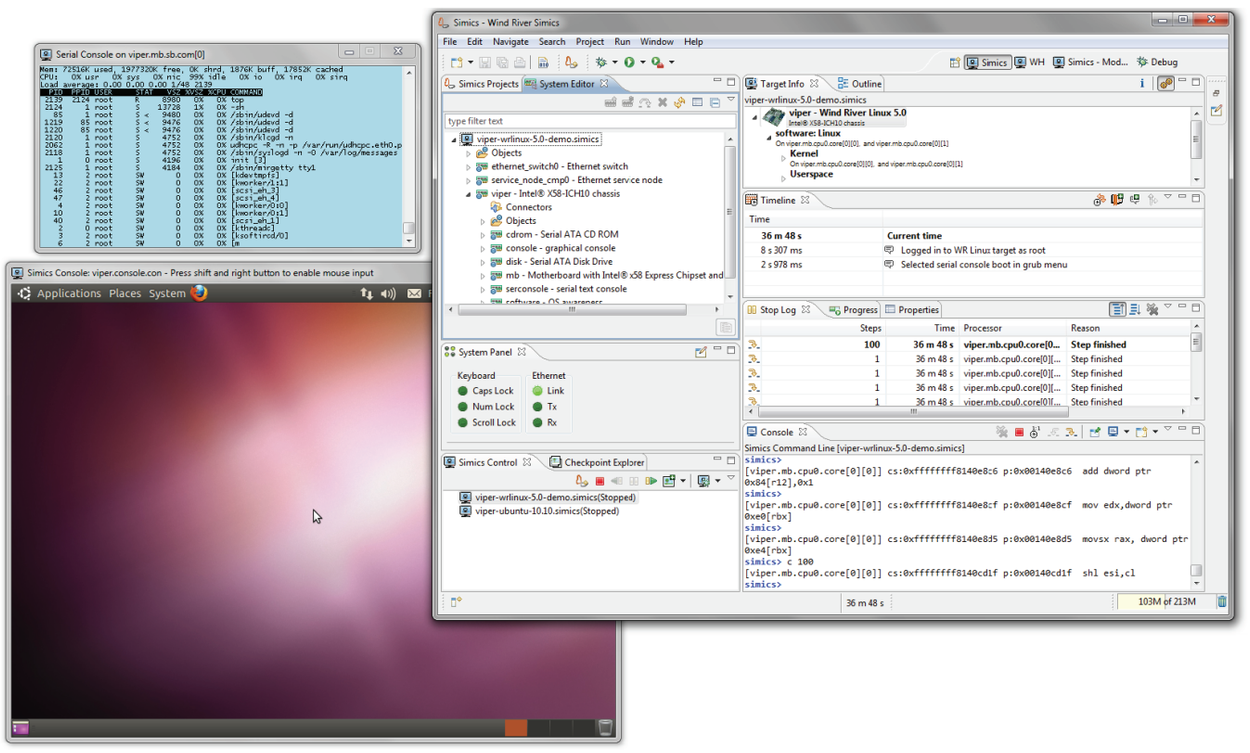

Figure 4: Simics Eclipse GUI and target consoles (Source: Wind River, 2013)

As illustrated in Figure 5, network connections from Simics to the outside world are accomplished indirectly. The target system is connected to a virtual network, and that virtual network can in turn add a connection to the real world. There can be other virtual target systems on the virtual network, and it is quite common to add features like traffic generators and inspection modules to a virtual network to inspect and affect the target system behavior.

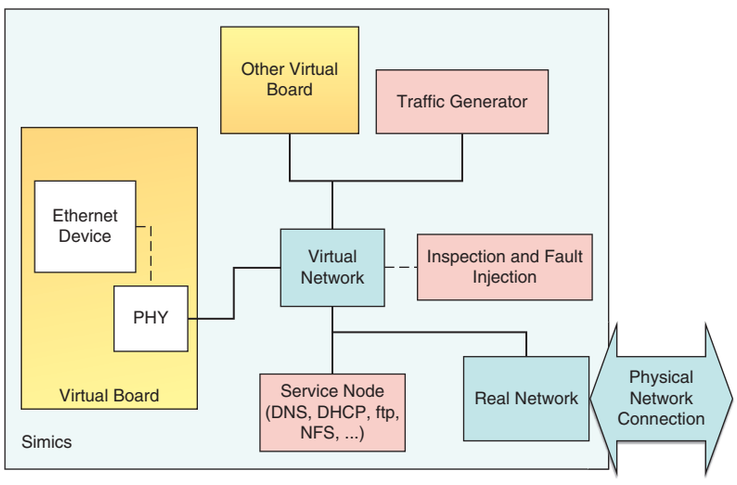

Figure 5: Simics network simulation (Source: Wind River, 2013)

Visibility and Control

The Simics GUI, CLI, and the Simics API provide deep and 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, memories, and interconnects. Figure 6 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.

"It is easy to look inside any part of the system and check the state of hardware devices, processors, memories, and interconnects."

As well as passively observing the state of the target system, Simics users can change it. This is used for fault injection or to quickly set up a system to make software run without necessarily having all boot code in place.

Scripting

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 CLI 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

"…there is a full Python environment embedded in Simics."

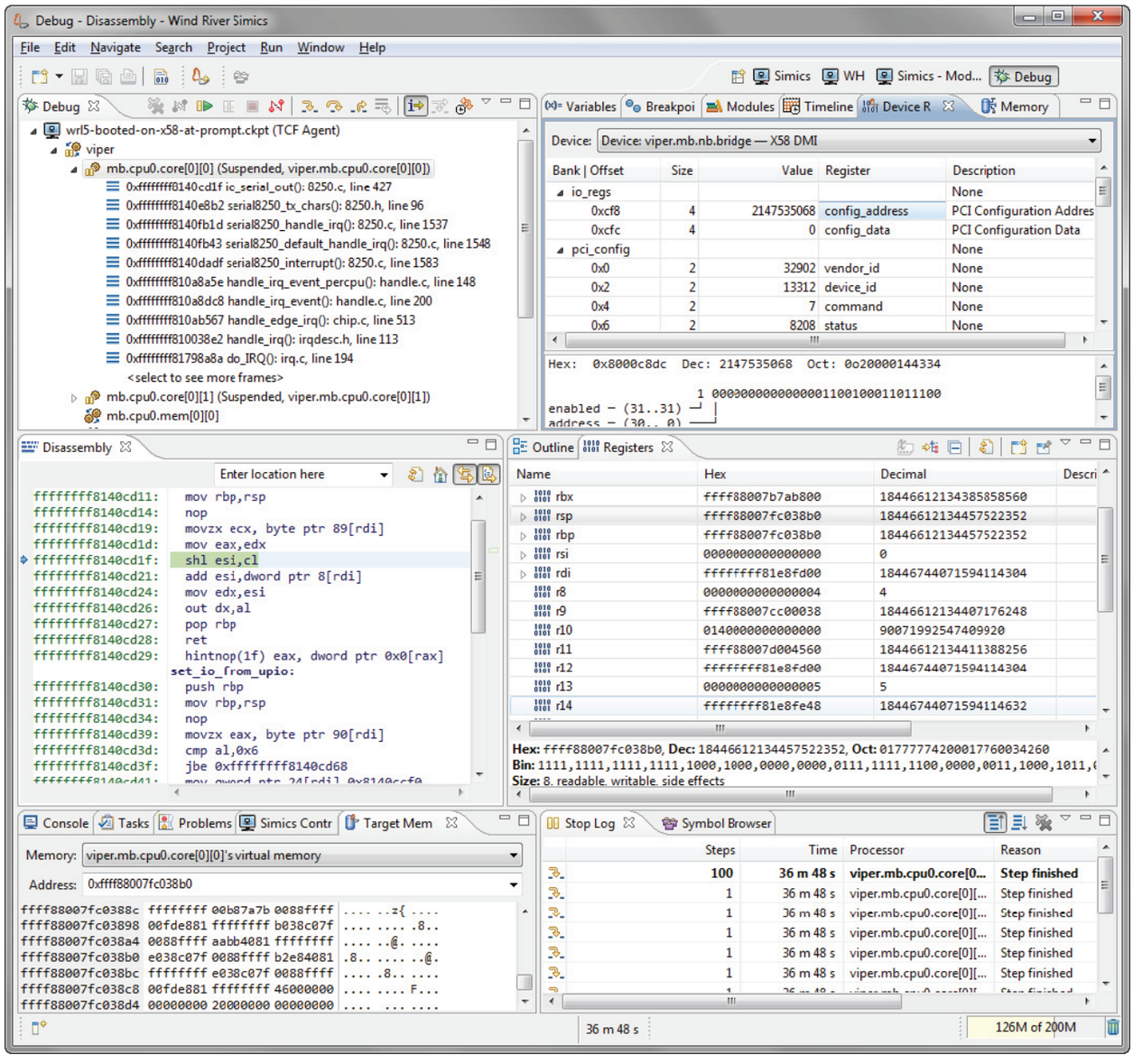

Figure 6: Simics target system inspection (Source: Wind River, 2013)

"Users can create their own custom CLI commands…"

variables with each other, and it is common to find snippets of Python embedded inside of Simics scripts. Users can create their own custom CLI commands in order to automate or simplify common tasks peculiar to their system or environment.

A typical Simics scripting example is shown in Code 1. It is a script that 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 are also provided with 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.

```
## 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
}
```

Code 1. Example Simics Target Automation CLI Script
Source: Wind River, 2013

"Compared to configuring hardware lab setups for even small networks, Simics can save hours and days of setup time."

Using Simics scripts, it is easy to automate and replicate the setup of even the most complex target systems. Multiple machines, boards, and networks can all be set up, configured, and reliably reproduced. Compared to configuring hardware lab setups for even small networks, Simics can save hours and days of setup time.

The article "Using Simics in Education" mentioned earlier describes how network topologies are automatically generated in order to support networking training, providing a typical example of the power of Simics scripting to automate system setups.

Another use-case enabled by automation is testing of code as it is being built or checked into version control. With Simics, it is quite easy to launch an actual target machine (for any target architecture), load the software, and test it. Physical hardware would be much harder to invoke on-demand and automatically in this fashion.

"Using Virtual Platforms for BIOS Development and Validation" mentioned earlier describes how BIOS code is tested on check-in, both before and after the availability of silicon.

OS Awareness and Debugging

Simics includes a very powerful full-system debugger, based on Eclipse CDT and some Wind River extensions. The debugger functionality is equally accessible from the Simics command line, providing the ability to automate debug tasks and to control the debugger from the CLI while looking at the state of the system in the GUI.

"The Simics debugger obviously supports reverse debugging,…"

The Simics debugger obviously supports reverse debugging, 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.

"OS awareness provides the user with a full software perspective of the system,…"

As Simics models the actual hardware and runs the OS code just like the physical hardware would, it does not directly know anything about the OS. Indeed, it is not necessary to run an OS on a Simics model, "bare-metal" code is commonly used for low-level tasks. Thus, for Simics to be able to provide advanced features based on the OS running on the target a feature known as OS awareness is necessary. OS awareness provides the user with a full software perspective of the system, in addition to the hardware perspective. OS awareness allows Simics investigate to the state of the target system and resolve the current set of executing threads and processes. The OS awareness module for a particular OS knows the layout and structure of things like process descriptor tables and run queues, and can provide the debugger with information about the currently running processes and threads. OS awareness lets the Simics debugger, scripts, and extensions act when programs and processes are started, terminated, or switched in and out.

The debugger leverages OS awareness 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 debug and to software stack contexts (like a kernel or user application) for debugging only a certain part of the software system.

"The debugger leverages OS awareness to allow debugging of individual applications and threads, as well as stepping up and down through the software stack layers."

The Simics debugger is a full-system debugger, meaning that it connects to the entire target system and not just a single processor or board. In Figure 7, we see two target machines inside a single debug session (server_p and server_a),

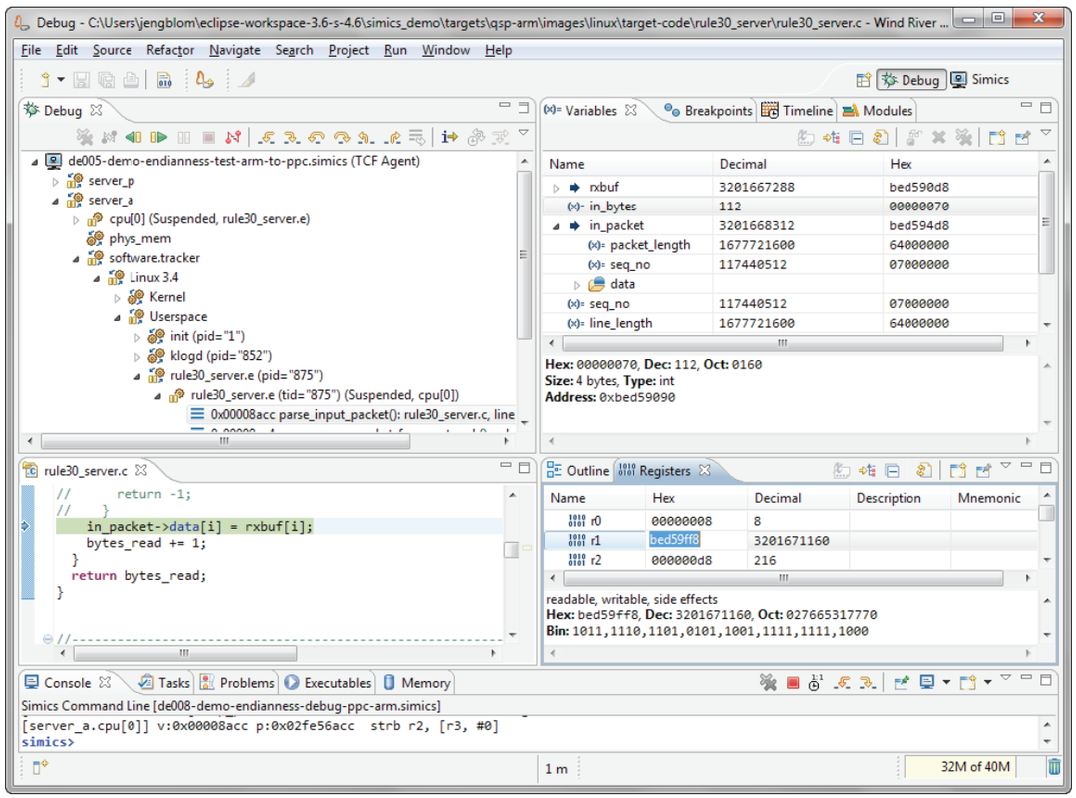

Figure 7: Simics debugger
(Source: Wind River, 2013)

as well as OS awareness digging down to a certain thread inside the program called "rule30_server.e" and doing source-level debug on this particular program in the context of the overall system.

Checkpointing

Simics has been designed from the ground up to support checkpointing of the simulation state. This gives Simics the ability to save the complete state of a simulation to disk and later bring the saved state back and continue the simulation without any logical interruption from the perspective of the hardware model and the target software.

"Simics has been designed from the ground up to support checkpointing of the simulation state."

"Checkpoints contain the state of both the hardware and the software…"

Checkpoints contain the state of both the hardware and the software (which is implicit in the hardware state as it is described by the contents of memory, disks, CPU registers, and device registers). Based on our experience, Simics checkpoints are portable across time and space, and let users do things like the following:

- Restore the simulation state from a previous run for the same user on the same machine as the checkpoints were taken. This helps an individual user work more efficiently.
- Restore on a different host machine. This means that checkpoints can be shared between users, enabling all kinds of collaboration.
- Restore into an updated version of the same simulation model. This makes it possible to use checkpoints taken with older versions of a model, making them portable across time.
- Restore into a completely different simulation model that uses the same architectural state. For example, a detailed clock-cycle driven model initialized from a fast Simics run.
- Replay a particular sequence of inputs captured in one simulation session into a second simulation session.

Checkpointing can be used to support workflow optimization, such as a "nightly boot" setup where target system configurations are booted as part of a nightly build, and checkpoints saved. During the workday, software developers simply pick up checkpoints of the relevant target states, with no need to boot the target machines themselves.

"…package bugs and communicate them between testing and engineering, between companies, and across the world."

Another important use of checkpointing is to package bugs and communicate them between testing and engineering, between companies, and across the world. Simics checkpoints make the reproduction of the bug and the environment needed to reproduce the bug trivial.[2]

Simics checkpoints can contain an embedded history of asynchronous inputs. This makes it possible to communicate a slice of time, and not just an instantaneous state of the target machine. Alternatively, a checkpoint of a single point in time can be used along with a script that drives the simulation in a deterministic way to achieve the same effect.

Repeatability and Reversibility

Simics has been designed from the bottom up to be a *repeatable* and *deterministic* simulator, with the exact same simulation semantics regardless of the host machine. As long as asynchronous input to the simulator is being recorded, any simulation run can be repeated precisely on any host at any time. Note that determinism does not mean that the simulation always runs the same target software in the same way. If the timing of any input, or any part of the initial state changes, the simulation will execute differently.

Determinism does not prevent a user from exploring variations in target software behavior. Rather, the user remains in control and can repeat any simulation run where the variations triggered some interesting behavior.

Based on repeatability, Simics also implements reverse execution and reverse debugging, where the user can go back into the history of the system execution. Reverse execution was incorporated in Simics 3.0 and launched in March of 2005, which makes it the first usable reverse execution implementation. This is a powerful tool for software debugging, especially for intermittent and timing-dependent bugs, which are difficult to reproduce on hardware using classic iterative debugging. Note that Simics reverse execution applies to a complete target system, including multiple processors, boards, and operating-system instances. Network traffic, hardware accesses, and everything else going on in the system is reversed, not just a single user-level process as is targeted by most other reverse execution approaches such as gdb.[3]

"…Simics reverse execution applies to a complete target system, including multiple processors, boards, and operating-system instances."

A key enabler for determinism, checkpointing, and reverse execution is that Simics simulation models do not normally use resources outside the simulator; notice that the target machine is internal to Simics in Figure 3 and how the network is isolated from the model in Figure 5. Hardware models live in a completely virtual world and do not open files on the host or drive user interaction directly. All external interaction is handled by the Simics kernel and special infrastructure modules for text consoles, graphical consoles, and network links. In the case that the outside world needs to be connected to a model, Simics provides a recorder mechanism that can reliably replay asynchronous input under reverse execution.

The article "Landslide: A Simics Extension for Dynamic Testing of Kernel Concurrency Errors," by Ben Blum, David A. Eckhardt, and Garth Gibson, provides an example of a creative use of reverse execution. It is employed to implement a backtracking search through the execution space of a concurrent software stack.

Dynamic Configuration and Reconfiguration

A Simics simulation can be reconfigured and extended at any point during a simulation. New modules can be loaded and new hardware models and Simics extensions added. All connections can be changed, and simulation models deleted or disconnected from the running system.

Such changes are done from scripts, the Simics command-line, and the System Editor in Eclipse at will. This dynamic nature of a system is necessary to support system-level development work and to support the dynamic nature of the target systems discussed in the introduction.

Extensibility and Programmability

Simics is an extensible and programmable system. Any Simics user is able to build not just new device models and system configurations, but also arbitrary Simics extensions. With the Extension Builder product, users have access to the complete Simics API and can basically implement any functionality they want to. New functionality often starts out as scripts, but over time it migrates into custom Simics modules in order to make the setup more robust and to achieve higher performance.

"Any Simics user is able to build not just new device models and system configurations, but also arbitrary Simics extensions."

The article "Software Power and Performance Correlation on Simics" by Parth Malani and Mangesh Tamhankar describes an implementation of a power estimation tool inside of Simics, quite radically extending the kinds of data that can be collected from Simics.

The article "Sim-O/C: An Observable and Controllable Testing Framework for Elusive Faults," by Tingting Yu, Witawas Srisa-an, and Gregg Rothermel, and the article "Landslide: A Simics Extension for Dynamic Testing of Kernel Concurrency Errors," mentioned earlier both provide examples of how custom Simics modules can be used to build powerful software verification tools on top of Simics.

External World Connectivity

A simulator used for system development and software development often needs to include more than just the computer system and its software. In some way, the outside world needs to be introduced into the system.

The most basic connections are the serial and graphics consoles provided with Simics to allow a user to interact with the simulated computer system. It is also quite common to connect Simics simulated machines to the real world via Ethernet networks and serial ports, using various *real-network* systems to bridge between the physical world and the virtual system (as illustrated in Figure 5). In this way, Simics target systems have been used in hardware-in-the-loop simulations.

Today, it is very common to use simulation of the physical world during the design of products like vehicles, space crafts, and engines. Such environment simulators can be integrated with Simics, creating holistic models that encompass all aspects of the target system. Essentially, hardware-in-the-loop is replaced by simulation-in-the-loop, making it possible for any developer to have a complete cyber-physical system on their desk for software testing and debugging.

"…making it possible for any developer to have a complete cyber-physical system on their desk for software testing and debugging."

Figure 8 shows how such a setup is achieved. On the Simics computer side, there need to be models of the actual devices the computer uses to sense and control its environment. Then, the environment model is either run inside of Simics, or (more commonly) in a separate process communicating with a proxy module in Simics over a network socket or other inter process communication mechanism.

Simics Performance Techniques

Simics is designed from the ground up to be a fast simulator in the tradition of software-oriented simulators going back to the 1960s.[4] The key design goal is that it is better to run a whole software stack with a low level of timing fidelity, rather than run a very small piece of software with a high level of timing fidelity. For most software, detailed hardware timing simply does not matter much, and Simics takes advantage of this to create a very fast simulator.

"For most software, detailed hardware timing simply does not matter much, and Simics takes advantage of this to create a very fast simulator."

Transaction-Level Modeling

Simics is built on the ideas that are now generally known as transaction-level modeling. 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.

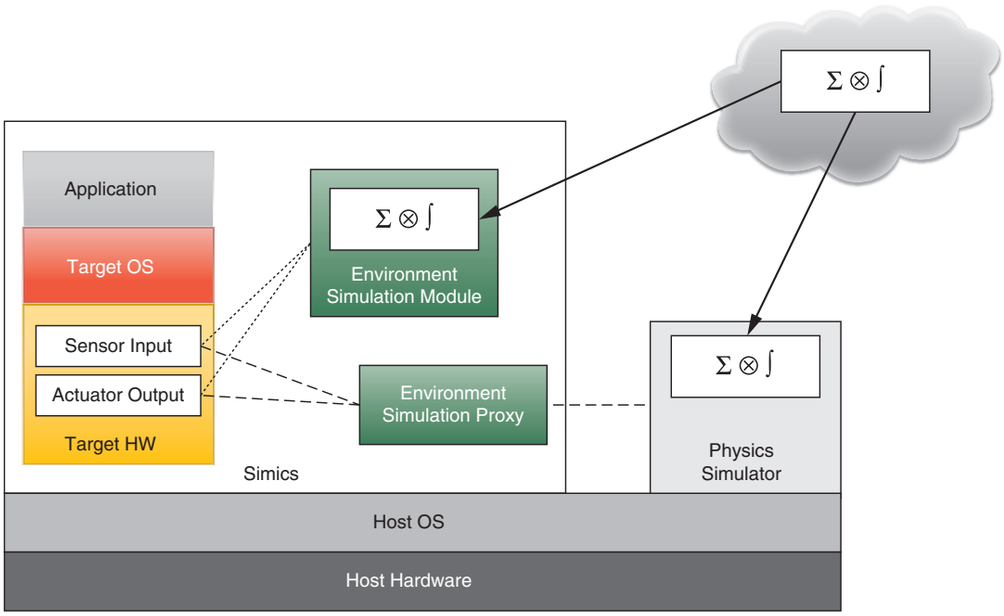

Figure 8: Simics and physics simulators
(Source: Wind River, 2013)

The Simics memory model is similar to the SystemC TLM 2.0 loosely timed (LT) model[5] 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; this is sometimes referred to as software-timed (ST) or programmer's view (PV). The different common timing models are illustrated in Figure 9.

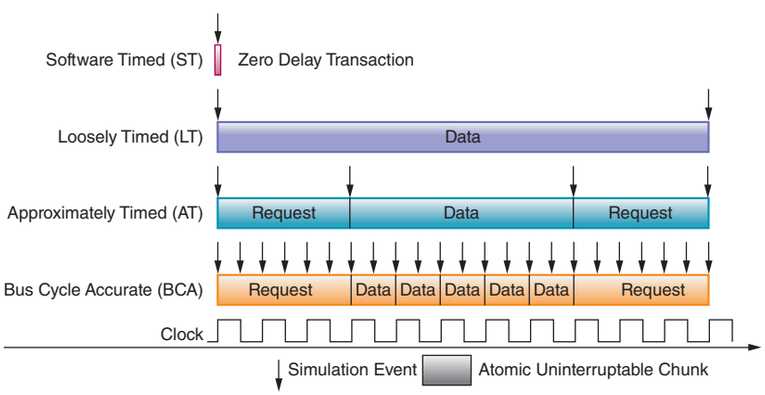

Figure 9: Simics TLM abstraction
(Source: Intel Corporation, 2013)

Often, immediate completion of an operation when a device register access occurs is sufficient for modeling a device. 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. This principle is applied to all types of communication in Simics, including networks, serial lines, and buses connecting hardware units. It opens up for a host of optimizations in the core simulation system.

Still, it is possible to connect more detailed models into Simics. Typically, most of the system runs at the standard Simics level of abstraction in order to maximize speed, with a few units replaced with detailed models driven by the software running on the fast Simics models.

"…detailed SystemC models can be integrated into Simics, creating a hybrid setup with a good balance between performance and simulation detail."

The article "Simics–SystemC* Integration" by Asad Khan and Chris Wolf describes how detailed SystemC models can be integrated into Simics, creating a hybrid setup with a good balance between performance and simulation detail. It is also common to connect Simics with hardware emulators.

Temporal Decoupling

Temporal decoupling is a standard technique for improving the performance of a simulation containing multiple concurrent units. Rather than switching between the units at each step of the simulation, such as a clock cycle, each unit is allowed to run for a certain amount of virtual time, its *time quantum*, before switching to the next unit. Temporal decoupling has been in use for at least forty years in the area of computer simulations[5] and is a crucial part of all fast simulators today[6]. Experience shows that using a fairly large time quantum is critical to achieving really high simulation speeds. In our experience, performance typically increases by a factor of 100 from a time quantum of 10 cycles to a time quantum of 100,000 cycles. As far as the software is concerned, time quanta below 100,000 cycles tend to be unnoticeable.

"This allows Simics to reach speeds in excess of 1000 MIPS…"

Fast Processor Simulation

Simics CPU models employ JIT simulation techniques where the target CPU code is translated into code for the host CPU. This allows Simics to reach speeds in excess of 1000 MIPS when simulating compute bound code on nonnative instruction sets such as a Power Architecture target on an Intel® architecture host.

For Intel architecture targets, Simics also takes advantage of the Intel® Virtualization Technology (Intel® VT) found in Intel processors to run the target code directly on the host. This makes it possible to achieve performance close to native speeds.

Another important processor simulation technology is *hyper-simulation*, where the processor model skips through idle time in a single simulation step, rather

than going through it cycle by cycle. For example, if an Intel architecture processor executes the HALT instruction, it will not do anything until the next interrupt. Since Simics knows when the interrupt will happen, time is advanced immediately. This is enabled by the architectural isolation of targets from their environment, as illustrated in Figure 3. Simics hyper-simulates defined idle instructions, but can also automatically detect loops that do nothing except wait for an event and immediately skip forward to the loop exit condition. This means that hyper-simulation applies to many operating system idle loops, even when such loops do not make use of power save or idle instructions.

Multithreaded Simulation

"Simics makes use of multiple host processor cores to simulate the target system."

Simics makes use of multiple host processor cores to simulate the target system. When running in *multithreaded* mode, Simics still implements precisely the same target semantics and behavior as when running single threaded. This means that the simulation behavior is independent of the host, and that simulation repeatability is maintained as checkpoints and setups are communicated between Simics users.

When the processor power (or memory) of a single host is insufficient to run a large system, Simics can also use *distributed* simulation. In such a setup, multiple Simics processes running on different hosts are connected into a single coherent and time-synchronized simulation system. Multithreading lets Simics take advantage of scale-up of individual hosts, and distribution takes advantage of scale-out as more simulation hosts are added. See the article by Rechistov for an example of scaling up and scaling out Simics to run a very large target system.

System Modeling

Getting a model in place for a relevant target system is a prerequisite to using Simics. Without some virtual hardware to run on, software will not be particularly interesting.

Using Existing Models

"…Intel and Wind River have a substantial library of existing models that can be used."

The simplest way to get a model is to use one that already exists. This is a fairly common case in practice, since Intel and Wind River have a substantial library of existing models that can be used. For example, Intel has created models of quite a few modern Intel hardware platforms, and such platform models can be used with very little work. Over time, platform models (typically of reference boards) tend to be customized to become models of the actual boards used in a particular system.

For other users, a standard system might be sufficient. Simics ships with Quick Start Platforms (QSP), which provide a simple idealized multicore system that runs Wind River Linux and VxWorks by using customized BSPs. The QSP provides serial ports, Ethernet ports, timers, and disks. The QSP will not run the same bootrom and OS image as a real board, but it will in general run user-level application binaries compiled for the real system. In this way, they provide a system that gets a user started quickly and that is entirely sufficient for using Simics features to debug, test, and analyze software applications.

"…Simics provides the tools necessary to quickly and efficiently develop new models that can be easily integrated into existing targets."

Device Modeling

If a model does not exist, Simics provides the tools necessary to quickly and efficiently develop new models that can be easily integrated into existing targets. The core of building models of new hardware in Simics is the modeling of the device found in the new hardware. As mentioned above, Simics provides a C API and ABI meaning that models written in almost any language can be integrated into Simics. However, Simics also provides its own domain specific language, the Device Modeling Language (DML), which is specifically developed to allow rapid development of robust device models for Simics. Besides DML, the other most commonly used languages for creating device models are C, C++ (including SystemC), and Python.

"DML is based on the view that modeling is programming,…"

DML essentially wraps snippets of C code inside a language designed to make it easy to express device register maps and other common hardware modeling constructs. DML is based on the view that *modeling is programming*, and tries to make the code required to describe a device model as short and concise as possible.

Key features supported by DML include expressing register maps, bit fields in registers, bit-endianness, byte-endianness, multiple register banks, simulator kernel calls such as event posting, and the connections between cooperating device models. *Templates*, not to be confused with C++ templates, are used to provide reuse for repeated patterns of code. DML cuts down on repetitive coding and makes device model code easier to write and maintain. DML can also be used to wrap existing C and C++ models of algorithms and core hardware functionality into a register map and Simics model, enabling their use inside of a Simics virtual platform. DML separates declarations and definitions, allowing reuse of artifacts from the hardware design by automatically converting register descriptions to DML declarations.

The article "Device Driver Synthesis" by Mona Vij et al. describes a creative use of the device models created for Simics. They use DML models of hardware created as part of the hardware design process as an input to a tool that creates device drivers.

The article "Using Simics in Education" mentioned earlier contains an example of using DML to create a device for the purpose of training in device driver development.

No matter how device models are programmed, the recommended methodology in Simics is to build functional tests that test a device in isolation before integrating it into a system. For this purpose, a test framework written in Python is provided, along with an Eclipse GUI to make running and inspecting tests a natural part of the development flow.

Figure 11 shows some of the Eclipse views provided by Simics to aid modeling. We can see the Test Runner in the upper right-hand corner, and

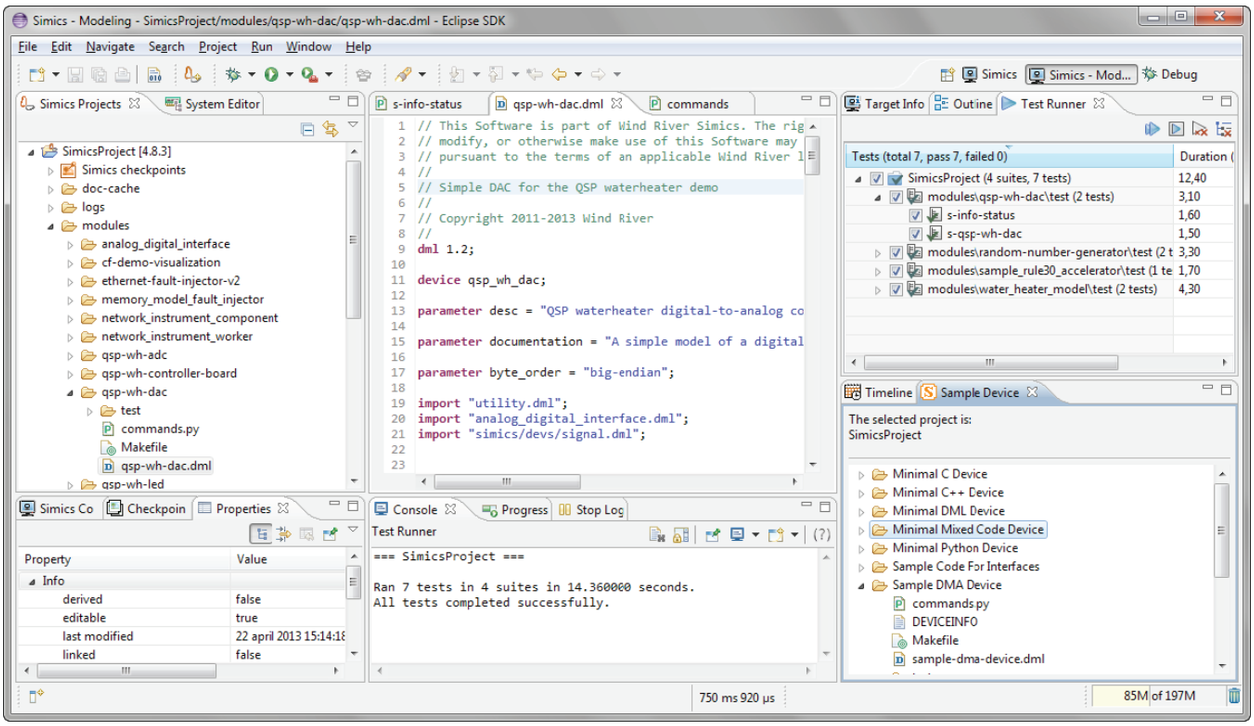

Figure 11: Simics modeling in eclipse
(Source: Wind River, 2013)

the Sample Device browser below it. The Eclipse "New Sample Device" wizard creates new devices, and other modules, based on the examples provided with Simics. The Sample Device view lets you look at the example code without necessarily creating a new device model in your own workspace. This is quite convenient when borrowing functionality from an example.

Component System

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, SoC, platform controller hubs, memory DIMMs, PCI Express* cards, Ethernet switches, and similar familiar units. They carry significant metadata and provide a natural map of a system.

Figure 12 shows a stylized example of a component hierarchy, where a system is built from two boards connected by an Ethernet network. At each level of the hierarchy, device models can be present, as well as components. The component system provides both structure to the models and a hierarchical namespace for the running simulation, making it easy to reuse components and devices without any risk of names clashing.

"The component system provides both structure to the models and a hierarchical namespace for the running simulation,…"

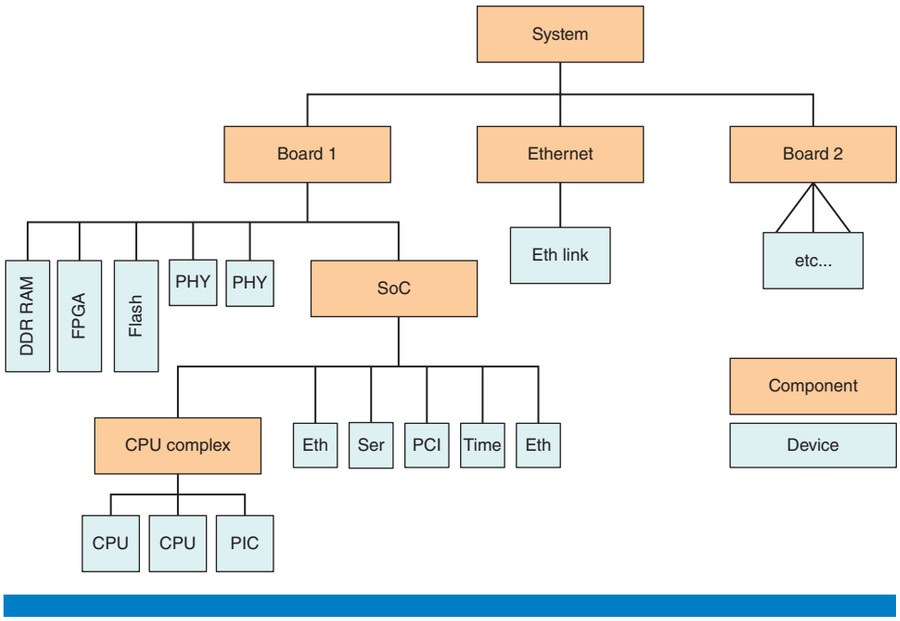

Figure 12: Simics component system
(Source: Wind River, 2013)

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.

Figure 13 shows an example of the system editor view in Eclipse with an instantiated real 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, and it can be seen how it is hierarchically constructed from components reflecting the logical hierarchy of the physical hardware.

"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 MAC addresses of an Ethernet controller."

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 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.

Summary

In this introductory article, we have presented the Simics technology along with some of its use cases and features. The following articles in this issue of the Intel Technology Journal will describe particular ways in which Simics has been used at Intel, Wind River, and in academia.

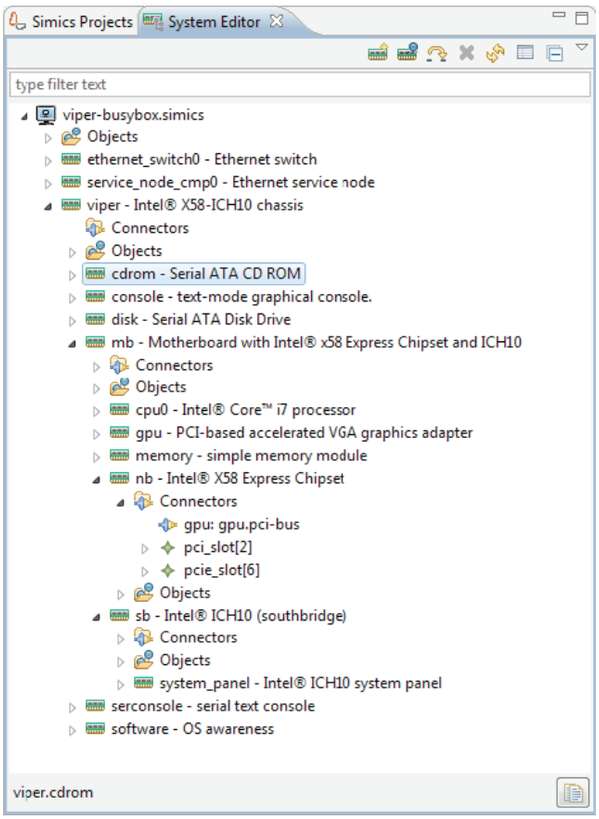

Figure 13: Example of a component hierarchy (Source: Wind River, 2013)

References

[1] Introspection of a java virtual machine under simulation, Tech. Rep. SMLI TR-2006-159, Sun Labs 2006.

[2] Engblom, Jakob: Transporting Bugs with Checkpoints, in *System, Software, SoC and Silicon Debug Conference* (S4D 2010), Southampton, UK, September 15–16, 2010.

[3] Engblom, Jakob: A review of reverse debugging, in *System, Software, SoC and Silicon Debug Conference* (S4D 2012), Vienna, Austria, September 19–20, 2012.

[4] Kazuhiro Fuchi, Hozumi Tanaka, Yuriko Manago, and Toshitsugu Yuba. 1969. A program simulator by partial interpretation. In *Proceedings of the second symposium on Operating systems principles* (SOSP '69). ACM, New York, NY, USA, 97–104. DOI=10.1145/961053.961092 http://doi.acm.org/10.1145/961053.961092

[5] IEEE Standard for Standard SystemC Language Reference Manual, IEEE Std 1666, 2011.

[6] Cornet, J., Maraninchi, F., Maillet-Contoz, L. 2008. "A method for the efficient development of timed and untimed transaction-level models of systems-on-chip." *DATE '08: Proceedings of the conference on Design, automation and test in Europe, pp. 9–14.* doi http://doi.acm.org/10.1145/1403375.1403381

[7] Magnusson, P., Christensson, M., Eskilson, J., Forsgren, D., Hallberg, G., Hogberg, J., Larsson, F., Moestedt, A., Werner, B.: Simics: A full system simulation platform. Computer 35(2), 50–58 (2002).

[8] Jakob Engblom, Daniel Aarno, and Bengt Werner, 2010. "Full-System Simulation from Embedded to High-Performance Systems". Processor and System-on-Chip Simulation, Rainer Leupers and Olivier Temam (ed), Springer New York Dordrecht Heidelberg London, ISBN 978-1-4419-6174-7, DOI 10.1007/978-1-4419-6175-4.

Author Biographies

Daniel Aarno is an engineering manager in Intel's Software and Services Group where he leads a team working on the Simics full system simulation product. Daniel joined Intel in 2010 through the acquisition of Virtutech.

Daniel holds a master's degree in electrical engineering and a licentiate's degree in computer science. Prior to joining the Simics team at Virtutech, Daniel was doing research at the Centre for Autonomous Systems at the Royal Institute of Technology, focusing on human-machine collaboration. During this time he took part of the PACO+ consortium within the 6th framework for research in the EU.

Jakob Engblom is a technical marketing manager for tools at Wind River, with an emphasis on product management for Simics as a commercial product (for end users outside of Intel). He has been working with Simics since 2002, when he joined Virtutech right after getting his PhD in Computer Systems from Uppsala University, Sweden. He also holds a master's degree in computer science from Uppsala University. Over the years, he has worked with Simics in a variety of roles, including outbound marketing, inbound marketing, sales and field engineering, managing the academic program, and product management. He has written and presented more than 100 articles, papers, and talks on a variety of embedded systems topics since 1997. Currently, his main interests are in computer simulation, general simulation, programming, and debugging. He has a personal blog at http://jakob.engbloms.se, and blogs regularly on the Wind River tools blog at http://blogs.windriver.com/tools/.

USING VIRTUAL PLATFORMS FOR BIOS DEVELOPMENT AND VALIDATION

Contributor

Steve Carbonari
Intel Corporation

Creating a simulation environment for the purpose of BIOS debugging and validation requires in-depth simulation models. A separation of initialization versus runtime models is required to optimize performance, improve simulation initialization accuracy, and the debugging environment. The usage model for BIOS debugging and validation can be broken up into two categories: pre-silicon (before initial hardware is available) and post-silicon (after initial hardware is available). Specific debugging features are required to debug BIOS programs due to the large volume of interaction with the system hardware, its impact to the simulation environment, and a desire to replicate the power-on environment interfaces. Specific attention should be given to signaling a software programming error as soon as possible. In addition, specific simulation techniques need to be applied for BIOS memory reference code support. Lastly, using the simulation for validation requires configuration flexibility and fault injection to fully validate all paths within the BIOS being validated. This article describes the high level concepts and additional depth of modeling used to approach debugging and validating BIOS with simulation tools. Although the context of the article is BIOS development and validation, the concepts can be applied to simulation for any firmware project.

"BIOS demands additional functionality and depth from software simulation."

Introduction

BIOS (Basic Input/Output System) refers to the software that runs after initial power is applied to the computer platform. BIOS' primary function is to initialize all hardware components to enable the computer platform to run higher level software (such as an operating system). Developing, debugging, and validating BIOS using software simulation demands additional functionality and simulation depth. Particular attention needs to be given to signaling errors at the time of register write, platform configurability, and mechanisms for fault injection. This article describes the basics of separating simulation runtime versus initialization, pre-silicon versus post-silicon usage models, development and debugging in the context of BIOS (specifically memory and processor interconnect initialization code), and BIOS validation requirements.

Initialization versus Runtime

To fully understand the demands that BIOS places on a simulation tool, an understanding of the differences between hardware initialization and runtime environments must be understood. When hardware is first powered on, only the minimal amount of components have power. Low-level firmware initializes the minimal amount of hardware components to enable BIOS to execute.

This initial state of platform hardware prior to BIOS execution will be referred to as the *initialization state*. The BIOS executes to initialize the rest of the hardware in preparation for the operating system to run. The state entered after BIOS has executed and prior to the operating system is running will be referred to as the *runtime state*. Refer to Figure 1.

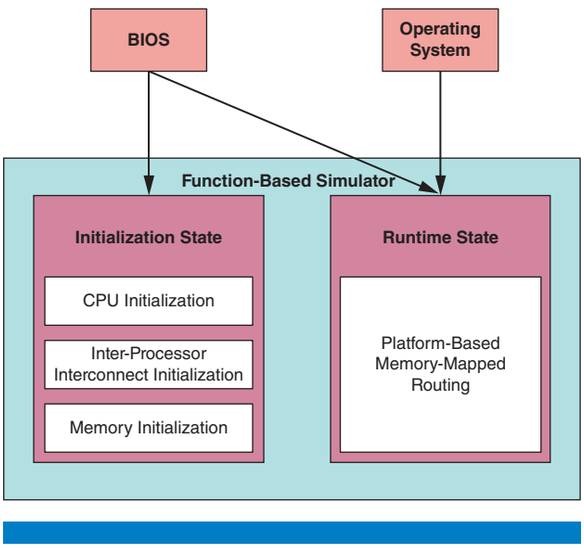

Figure 1: Initialization vs. Runtime States (Source: Intel Corporation, 2013)

In the initialization state only minimal hardware components are initialized. The goal is to enable hardware components required to enable fetching and executing BIOS, such as:

- CPU cores are initialized
- A path to the BIOS flash is established
- Processor interconnects are initialized and available in slow speed mode with minimal routing

In the runtime state the hardware is fully functional for use by an operating system. All hardware components are discovered and configured. In this state several interfaces are transparent to the operating system and hidden by the system memory map:

- Socket interconnects
- Memory channels and interleaving
- Memory type and speed
- Hardware testing interfaces

Many software simulation tools can provide a platform-level simulation interface to support operating system and driver development. However, to support BIOS development the simulation tool must simulate the initialization state to a sufficient depth to support the configuration of a variety of hardware components

"1) In the initialization state only minimal hardware components are initialized.

2) In the runtime state the hardware is fully functional for use by an operating system."

that are not required (transparent) during the runtime state. In addition, to support a seamless boot with optimal performance the transition from initialization state to runtime state must be transparent to software and BIOS.

"The implementation of memory spaces provides a mechanism to transition from initialization to runtime states."

The Wind River Simics* implementation of memory spaces[1] provides a mechanism that supports the transition of components from initialization to runtime. When instructions are executed they access addresses. These addresses are resolved using Simics memory spaces. If the address does not exist in the Simics memory space infrastructure, an error is reported. The dynamic nature of memory spaces allow them to be added and removed during the simulation run. This provides the ability to only add memory spaces after the underlying components have been fully initialized. More detail on how this mechanism is used to support BIOS development will be described in the section "Development and Debugging BIOS."

Key to transitioning between initialization and runtime states is verifying the initialization is completed per the hardware specification. In many cases registers can be written with incorrect values that will not be discovered on real hardware until much later, making debugging difficult. For example, hardware decoders are programmed by BIOS to enable access to memory. The hardware does not check to see if the decoders are programmed correctly. Consequently, if BIOS programs a memory decoder to point to nonexistent memory it will not be discovered until an application accesses the memory region resulting in a hardware machine check and system halt. Therefore, just modeling the components exactly matching hardware behavior is insufficient. The simulator must also add specific checks to verify registers were programmed correctly prior to switching to the runtime state.

Usage Models

The two primary usage models for BIOS development are pre-silicon and post-silicon. Both environments present unique requirements on the simulation tool used.

Pre-Silicon Usage Model

Pre-silicon validation is the activity of validating software prior to silicon availability. In a pre-silicon environment a variety of tools are available for BIOS development and debugging:

- Software simulation—Simulation tools focus on the functional model and interfaces of the hardware platform. They are fast and easy to update to accommodate the latest hardware changes. However, they usually do not model timing characteristics and are less accurate.
- RTL (resistor-transistor logic) emulation—RTL emulator's use programmed field-programmable gate arrays (FPGAs) to emulate the hardware based on the RTL. These are very accurate models. However, they are slow, expensive, and take time to update to the latest hardware changes.
- Pre-power on platforms—Pre-power on systems use interposers on existing platforms to allow for testing of components that have completed

development. These systems are expensive, available in limited quantities, and require BIOS changes to use.

- BIOS support applications—These are applications developed by BIOS teams for specific point testing and data gathering. The primary function is to gather BIOS behavioral and flow information to be evaluated by design engineers for correctness (for example, an application that uses simple text file formats to take register input values and capture register output values from BIOS).

Consequently, due to cost and availability, the primary tool used for BIOS development and debugging in the pre-silicon environment is functional software simulation.

In this environment, hardware interfaces and feature sets can change frequently, requiring maximum flexibility in the simulation tool to quickly adapt to hardware specification changes. In addition, the development of the simulation tool is based on the same specifications as the BIOS development. Those specifications are usually not complete, so staged development is required and close coordination with the BIOS development team regarding

"The development of the simulation tool is based on the same specifications as the BIOS development."

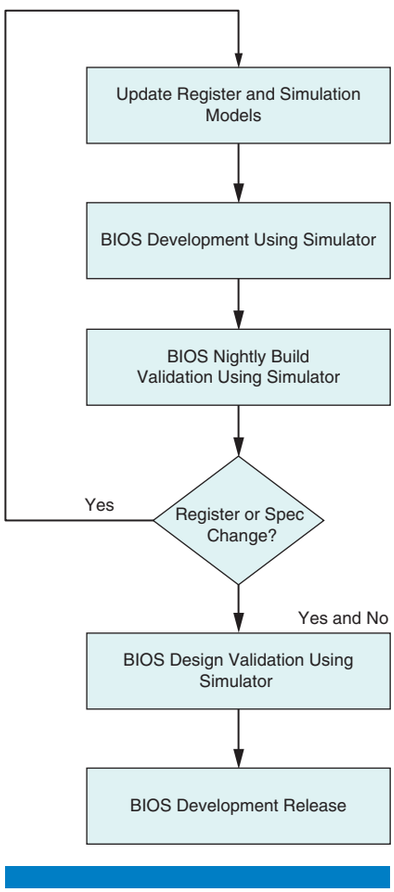

Figure 2: Pre-silicon BIOS development flow (Source: Intel Corporation, 2013)

feature set development and timelines is required (See Figure 2 for pre-silicon BIOS development flow). In this stage:

- The simulator is the primary BIOS development tool
- The functionality of the simulation tool is focused on supporting the hardware power-on configurations
- Validation teams use the simulator to pre-validate BIOS as well as the validation tests that will be run on hardware
- The BIOS team uses the simulator as a check-in criteria test for software check-in

To facilitate smooth BIOS development in a simulated environment, a mechanism is implemented to communicate between the BIOS and the simulator. The mechanism enables the BIOS to recognize it is running in a simulated environment and allow it to tell the simulation it is skipping sections of BIOS that are not developed. The mechanism consists of using an offset in PCI configuration space of a valid bus/device/function but where no register exists in real hardware. When queried on real hardware the register will return 0 per the PCIe specification requirements. However, on the simulator it will return a nonzero value to indicate the BIOS is running on the simulator. It also provides bit fields that can act as a means to communicate between BIOS and simulator-specific feature enabling.

This mechanism is used:

- To increase performance of BIOS boot by skipping delay loops required for booting on real hardware.
- To recognize a variety of environments, such as simulation and emulation.
- To work around features not yet developed in BIOS or the simulation on a temporary basis.
- To enable a single BIOS binary build that can run on hardware and the simulation.
- To provide a convenient mechanism to test a simulator's ability to run an unmodified BIOS at any time.

Post-Silicon Usage Model

In the post-silicon environment hardware is available in limited quantities and configurations. The use of pre-power on systems is stopped and RTL emulation is slowed down. Early during the post-silicon phase the hardware availability and supported hardware configurations are limited. Consequently, the use of simulation continues:

"Early during the post-silicon phase the hardware availability and supported configurations are limited."

- For those configurations not yet available in hardware
- In lieu of hardware availability
- Due to cost of power-on hardware
- In day-to-day BIOS development

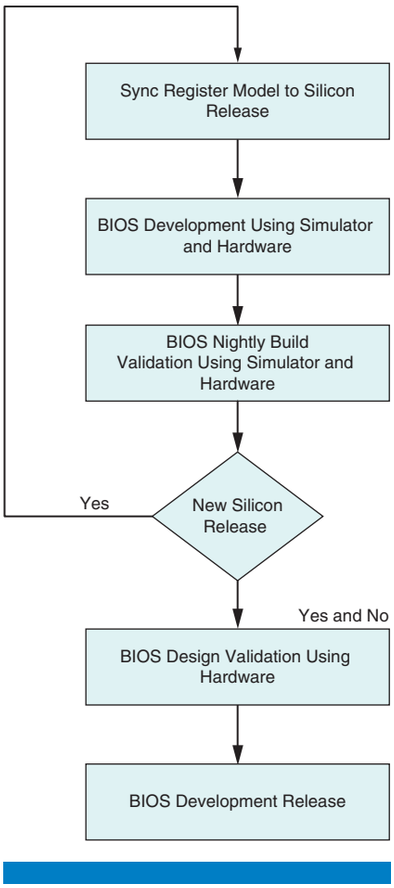

Figure 3: Post-silicon BIOS development flow
(Source: Intel Corporation, 2013)

The simulation tool flexibility and availability of a variety of debug and inspection tools provides a cost-effective and useful environment throughout the post-silicon phase. See Figure 3 for post-silicon BIOS development flow.

As more hardware becomes available and more complex configurations of hardware are supported, the use of the simulation tool will gradually diminish. In this stage:

- The development shifts from supporting basic configurations to supporting complex configurations.
- The validation teams continue to use simulation to prepare tests for more complex configurations.
- The simulator is the primary BIOS development tool, but power-on systems are increasingly used.
- The BIOS team continues to use the simulator as a development tool and as a check-in criteria test for software check-in due to low cost and accessibility.

Development and Debugging BIOS

The simulation tool is used heavily during early development and debugging of BIOS. Ideally the simulation tool will model the platform with enhanced features specific for debugging BIOS. However, for the specific case of BIOS development and debugging, many areas of the platform need not be simulated:

- Specific new processor instructions
- Registers that are not used by BIOS during boot or execution (such as performance measurement registers)

In addition, some of the BIOS requirements can be simulated in a fashion not necessarily true to the platform but still providing the necessary functionality for debugging. Memory channel level interleaving configurations can be supported without having to support the actual interleaving of reads and writes to memory. This allows memory accesses to be supported in a simple linear fashion, providing the best performance. Consequently, a simulation tool can provide a functionally complete simulation to satisfy BIOS pre-silicon validation requirements without simulating the entire platform.

"1) Simulation can provide a functionally complete simulation to satisfy BIOS requirements without simulating the entire platform.

In Simics the implementation of memory spaces enables the separation of memory channel interleaving configuration and application reads and writes. Memory channel interleaving is configured prior to memory being made available to the application level via a Simics memory space. As registers for channel interleaving are written the values are checked with the simulated memory configuration to confirm a valid interleaving configuration. If the configuration is invalid then an error is reported. The memory space for DRAM memory space is enabled later as a linear address space when the BIOS programs the decoders.

Understanding the specific BIOS requirements for platform simulation can allow a tool to focus on areas of most importance first when supporting BIOS pre-silicon validation. To demonstrate specific requirements for BIOS development and debugging we will examine the requirements in the context BIOS Memory Reference Code (MRC) and Intel® Quickpath Interconnect code (Intel® QPI).

Register Modeling

BIOS development starts when initial specifications and register definitions are available. The initial register definitions and specifications are incomplete and change frequently during the pre-silicon phase. Consequently, to avoid delays in BIOS development and minimize frequency of simulator releases, specific aspects of register modeling needs to be user configurable, such as:

2) The initial register definitions are incomplete and change frequently."

- Default values
- Attributes (Read/Write, Read Only, Sticky, and so on)
- Field definition
- Offset (where in PCI space the register resides)

Memory register sets can be very large and complex. In addition, due to the relatively fast pace of technology changes the register interface changes frequently. BIOS software must adapt to the register changes during the hardware design. As the design changes, giving the BIOS engineer the ability to change register values or get an updated release with new registers in a timely manner is critical. Ideally the simulator should provide a user configurable mechanism to change all the register values prior to starting the simulation. For example, a user-editable file that contains register definitions is read and configured at simulation startup.

"BIOS software must adapt to the register changes during the hardware design."

In addition, to aid the BIOS with discovering register issues, the simulator tool must provide flexible logging of all register accesses.

In Simics, default values are made user configurable via the attribute mechanism, and a variety of logging register mechanisms exist. However, the other register requirements are not user configurable. To mitigate this for BIOS development, scripting has been developed to take register definition files in XML format and convert them to Device Modeling Language (DML) register definition code[2]. This enables a very quick turnaround and rerelease of the simulator for register updates.

To support frequent updates of simulator packages, a utility was developed outside of the standard Simics product that automates simulator package updates, providing a simple yet effective method to update several different packages at once. The update utility examines what is installed on the system, queries the package repository server (location is configurable), and updates all packages selected in an automated fashion with very little required interaction from the user. See Figure 4.

Signaling an Error or Warning on Register Access

The BIOS configures many hardware systems during the initialization state. In many cases, the BIOS programming of the hardware system cannot be verified until much later during the runtime state. If the BIOS incorrectly programs the Source Address Decoders (SADs) to point to nonexistent memory, it will not be discovered until the memory is accessed, usually later during the runtime state. The SAD in hardware is responsible for mapping address requests to the correct hardware component. Consequently, the simulation tool must detect incorrect programming of registers (like the SAD) when the register is configured by BIOS so the issue can be detected and easily debugged. To achieve this goal the simulation tool must provide extra checking that the hardware normally does not provide.

Simulation model enhancements should be made in the following areas to support prompt error messages on register write:

- Register side-effect code is updated to check the values of the register write to confirm the BIOS values are correct.
- Memory spaces are not added until it can be verified that the underlying simulated hardware components have been initialized properly.

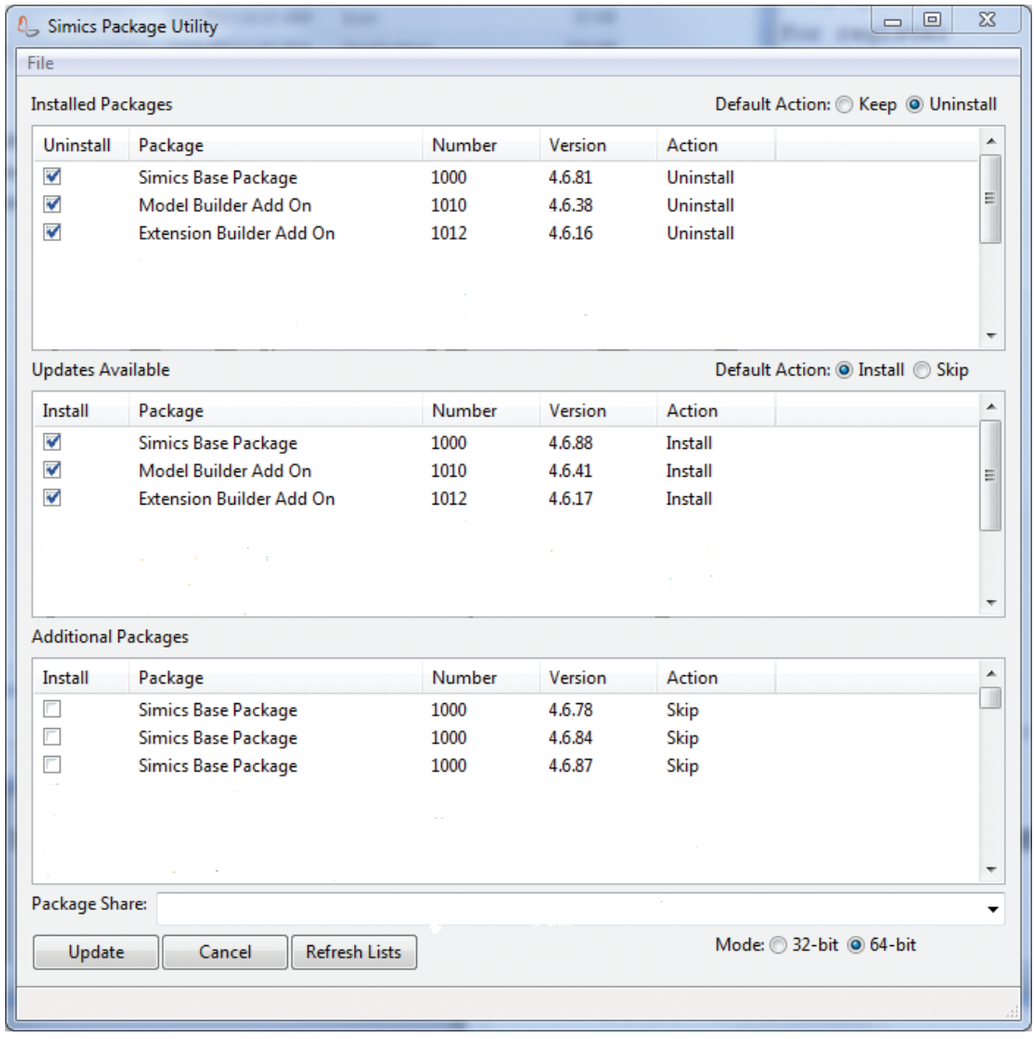

Figure 4: Simics package installer screenshot
(Source: Intel Corporation, 2013)

- Interfaces between the register side effects and internal component modules are defined so the register side effects can query the state of the underlying component.

Several key enhancements to a simulation model SAD register write side effect can be made to specifically support BIOS debug:

- Simulation memory spaces are mapped (enabled) only when BIOS writes the SAD register and sets the enable bit.
- The values written to DRAM decoders by BIOS are checked to confirm the underlying memory has been initialized. The simulator has knowledge of the amount of memory available so when the BIOS programs a SAD for greater than the amount of memory available, the simulator signals an error

at the time of register write to the SAD. Note that it is perfectly legal to program the SAD for less memory than what is available.

- Processor interconnect routes from the source processor's SAD being programmed to the destination processor is verified using a query interface to the processor interconnect simulation module. This mechanism enables processor isolation and bring-up of complex multiprocessor topologies.

"1) Processor interconnect routes are verified using an interface to the processor interconnect simulation module.

Modifying the simulator SAD implementation provides the following key benefits to BIOS development and debug:

- Decoder programming errors are discovered at the time of register write.
- Processor interconnect paths are validated as result of decoder programming.
- Uninitialized memory is exposed if an attempt is made to program a DRAM decoder to a region that is not yet initialized.
- Application access to invalid memory regions are based on BIOS programming of the memory regions.
- BIOS access to invalid memory regions is exposed because regions are mapped only when BIOS enables them, modeling the hardware enabling sequence.

Depth of Model to Support Initialization

Many hardware components require specific steps to initialize. Enforcing the specific steps for hardware initialization of a component requires a finite state machine (FSM) to be implemented in the simulator. FSM modeling is critical to BIOS pre-silicon validation. This includes proper input register modeling, state machine enforcement, and output register modeling. Furthermore, output register data should be customizable to allow for a variety of results defined by the user. Some state machines can have significant reuse from platform to platform, such as, Double Data Rate (DDR) or Dual Inline Memory Module (DIMM); some may need modification (Intel® QPI) and have to be rewritten due to platform changes (memory controller).

2) Some state machines can have significant reuse from platform to platform."

Modeling for Memory Initialization

The BIOS MRC refers to the BIOS module responsible for discovering and initializing the memory subsystem of the platform. BIOS MRC goes through three basic high level stages:

- Memory discovery: BIOS uses out of band interfaces to discover what memory and type are available.
- Memory training or DDRIO: BIOS configures the memory channels for optimum speed and efficiency.
- Memory consolidation and enabling: BIOS consolidates the memory from all sockets, configures reliability and performance features (DIMM sparing, interleaving), and programs the system decoders to allow access to the memory.

To effectively debug BIOS MRC the simulation tool must implement specific finite state machines. Memory initialization is the largest BIOS component

"The initialization of memory must be done with specific steps in a specific order."

that changes from one platform to the next. The initialization of memory encompasses several stages from reading basic DIMM data over SMBus, DDR initialization, initializing clocks and timing parameters, to programming SAD. The initialization of memory must be done with specific steps in a specific order. This ordering can be at the DIMM level, channel level, and memory controller level. Consequently, in-depth state machines are required by the simulation to enforce any required memory initialization ordering. In addition the state machines may cross component boundaries. Applying power may require interfacing with a power control unit on the board instead of the memory controller directly. Simulation of the following is required for BIOS memory initialization:

1. Configurable memory topologies and programmable SPD data files.
2. DIMM discovery.
3. I/O modeling state machine: validation of I/O programming for memory with error injection and programmable results.
    a. DDR I/O programming: (CS, CKE, ODT, and so on). This programs the I/O timings via the DDR bus. This includes enforcement of I/O programming state machine, returning random, pseudo-random, or preprogrammed training results values. With the DDR I/O model, having the ability to fail or pass read and writes is based on strobe delay values.
    b. SMI I/O programming: This programs voltage, electrical, and some timing parameters. Line equalization (EQ), voltage swings, and electrical parameters are programed. The SMI I/O model, including training, impedance and resistance (Icomp/Rcomp) compensation, and other FSMs on the I/O side.
4. DDR State Machine as specified in the DDR specification. Refer to the DDR specification for the full state diagram.[3] See Figure 5 for a simplified DDR state diagram.
    a. DDR states: Power, Reset Procedure, Initialization, ZQ Calibration, Idle, MPR/MRS Write Leveling (MPR 1/2/3), Self-Refresh, Refreshing, Pre-Charge Power Down, Activating, Active Power Down, Bank Active, Reading, Reading A, Writing, Writing A, Pre-Charging.
    b. DDR transitions commands: Active(ACT), Pre-Charge(PRE), Pre-ChargeAll(PREA), Mode Register Set(MRS), Refresh(RE), ZQ Calibration Long(ZQCL), Read, Read A, Write, Write A, Reset, ZQ Calibration Short(ZQCS), Enter Power Down(PDE), Exit Power Down(PDX), Self-refresh entry(SRE), self-refresh exit(SRX), Multi-Purpose Register(MPR).

To support the specific needs of BIOS MRC the following should be simulated in the model:

1. Configurable memory topologies and a variety of DIMM Serial Presence Detect (SPD) data files. This is implemented using the configuration tool as described in the section "Configuration Flexibility."

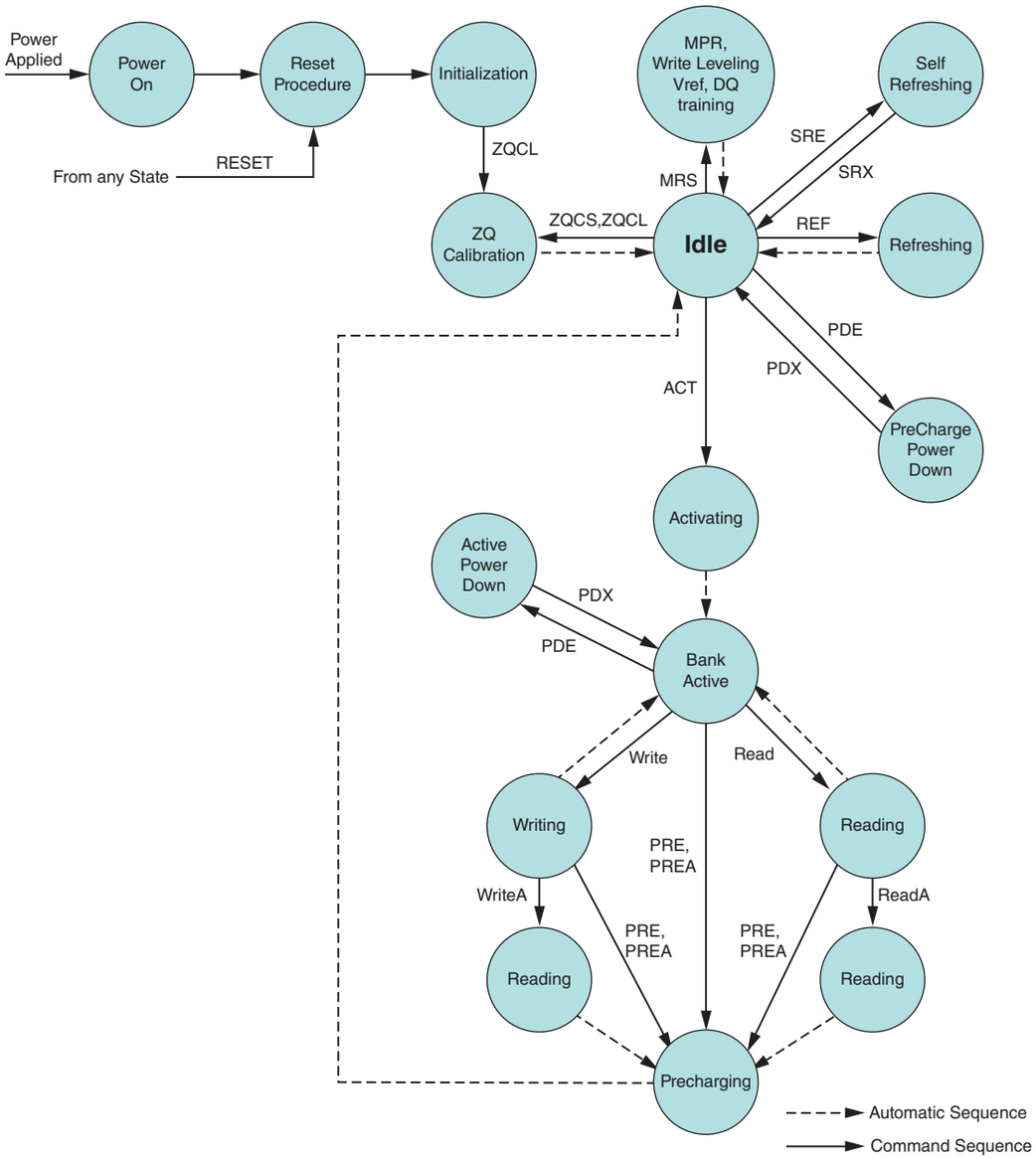

Figure 5: Simplified DDR state diagram[3]
(Source: From JESD79-3, Copyright JEDEC. Reproduced with permission by JEDEC.")

2. SMBus interfaces for platform DIMM SMBus topologies are implemented to enable BIOS MRC DIMM discovery. The implementation must include the support for memory buffering technology.
3. Training results for DDR I/O and SMI I/O are the driving factors in the BIOS execution flow. The simulator should provide a mechanism to customize the waveform pattern returned as a result of the BIOS pattern programming and the training stage. The values to define a waveform

pattern (period, phase, dutycycle, noisewidth) can be assigned via variables or attributes for the simulation memory model. In addition, attributes to define bit lane and nibble based skew are available to further customize the pattern across the lines on the DDR bus depending on the DDR training phase currently being executed.

"1) The simulator should implement a finite state machine modeling the DDR state machine.

4. A DDR state machine. The simulator should implement a finite state machine modeling the DDR state machine and enforces compliance on a per DDR rank basis during memory initialization. DDR memory training addressing is on a per rank basis. DDR DIMMs can have one or more ranks per DIMM (for example, a quad rank DDR DIMM has four ranks). As training commands are created and sent, a standard interface communicates with the simulator DDR model on a per rank basis to keep track and advance the DDR state machine.
5. The MRC training algorithms are in many cases defined late in the development cycle. Consequently, to enable debug and development of other BIOS features it is required to skip the memory training phase of boot. The BIOS needs to communicate to the simulator that it is skipping memory training so the simulator will not enforce the training state machines and cause a simulation halt. The simulator implements a mechanism for BIOS to communicate a specific feature is not developed, in this case memory training (mechanism detail is described in the section "Pre-Silicon Usage Model"). This allows other features of BIOS to continue development when memory training algorithms are not defined.

2) Even a functional simulator must provide timing-related information."

Timing and Analog Considerations:

In some cases even a functional simulator must provide timing-related information to aid in development and debugging of software. The BIOS memory reference code puts significant demands on a software functional simulator due to the timing-related nature of its process. The programming of the memory interfaces requires several stages of timing-related testing and programming commonly referred to as DDRIO. The DDRIO stage presents significant problems for a simulation tool. Due to the analog nature of the process and complex flows, significant detail and depth is required from the simulation tool.

The analog results for DDRIO are usually provided in results registers. The simulator must provide support for the BIOS engineer to customize the results of the registers holding the analog data. A key advantage of using a simulation tool over hardware is the ability to change hardware return values. In the case of BIOS memory code, the ability to provide user-customizable data to the simulation to return invalid and boundary condition values for analog data provides an effective means to test the BIOS memory training code in ways that cannot be done on hardware. See the sections "Configuration Flexibility" and "Fault Injection." The interface should be easy to use and understand due to the complexity inherent in the several stages of analog training of the BIOS MRC performs.

The extent of simulator support related to BIOS MRC validation and timing is localized to platform registers that return timing information for the purposes of hardware initialization.

Modeling for Intel® Quickpath Interconnect Initialization

Intel® QuickPath Interconnect provides a high speed inter-processor communication link.[4] The Intel QPI architecture supports a variety of topologies. See Figure 6. In conjunction with low-level firmware (FW) the BIOS is responsible for Intel QPI initialization across all topologies. The Intel QPI fabric initialization at a high level consists of[5]:

"The Intel QPI architecture supports a variety of topologies."

1. Path establishment: Hardware or microcode initializes a path from the System Bootstrap Processor (SBSP) to the location of the BIOS code. The SBSP is responsible for the overall platform initialization.
2. Topology discovery: The BIOS running on the SBSP controls the overall flow of topology discovery. However, it depends on firmware running on each Processor Bootstrap Processor (PBSP). The PBSP is responsible for initialization localized to the socket. After topology discovery the range of resources available are determined.
3. Link initialization: BIOS running on the SBSP with support from PBSPs programs the interconnect link registers to initialize the fabric for the topology discovered. A reset is performed for the values to take effect.

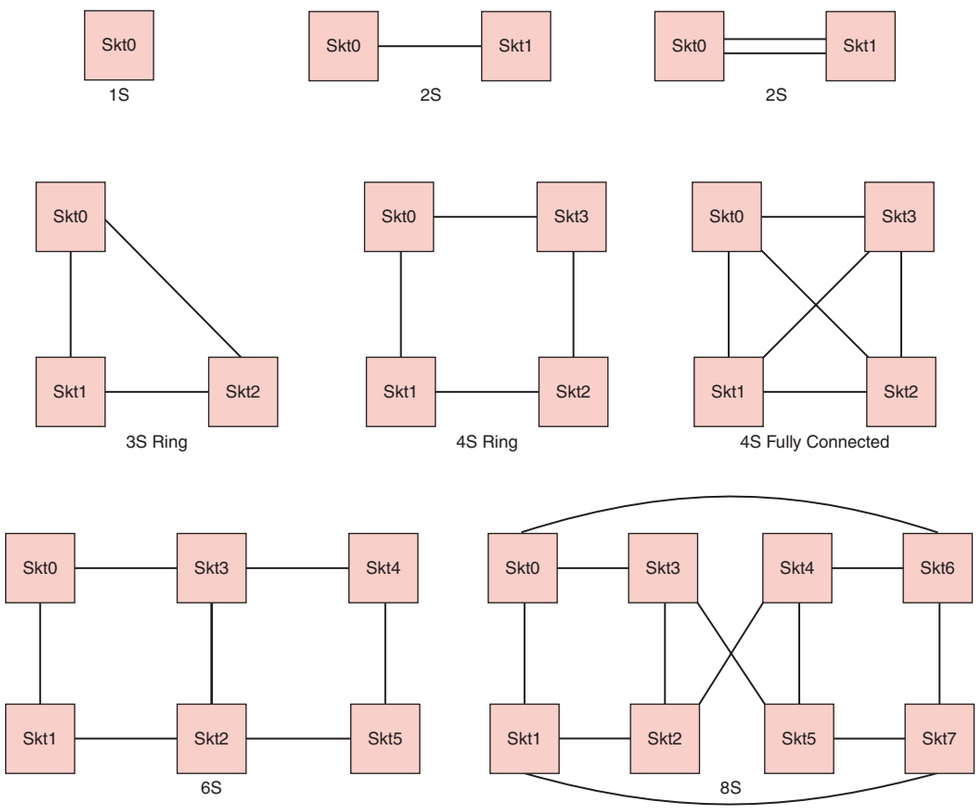

Figure 6: Examples of Intel® QPI topologies
(Source: Intel Corporation, 2013)

4. Route initialization: BIOS programs the interconnect routing tables and buffers to optimize path lengths and performance between sockets.
5. Decoder initialization: Once the Intel QPI fabric is initialized the decoders are programmed to enable cross-socket access.

To support Intel QPI initialization the following must be simulated:

"A significant portion of the Intel QPI initialization is driven by embedded firmware."

1. Customizable Intel QPI topologies.
2. The firmware role in Intel QPI initialization. A significant portion of the Intel QPI initialization is driven by embedded firmware. The flows of the embedded firmware must be modeled to enable validation of BIOS flows for Intel QPI initialization.
3. Inter-processor access constraints prior and after processor interconnect links have been initialized.
4. The Intel QPI initialization state machines that encompass both the BIOS and firmware flows.
5. Validation of BIOS route programming.

To support Intel QPI initialization flows, the simulation model should implement the following:

1. The configuration tool as described in the section "Configuration Flexibility" is used to define a variety of Intel QPI topologies based on the platform.
2. A simulation module that represents the firmware role of Intel QPI initialization. This module initializes to the state after power-on but immediately before BIOS starts execution. A communication mechanism between the firmware simulation module and the processor interconnect simulation module allows the processor interconnect simulation module to query state of the firmware.
3. An interface between the processor interconnect simulation module and the decoder simulation module is implemented to allow the decoder simulation module to query the processor interconnect simulation module prior to enabling a decoder that would require traversing the processor interconnect fabric to get to the destination processor. A route checking algorithm should be implemented that checks route existence using the BIOS programmed routes. This facilitates reporting errors at the time of decoder initializing.
4. Several small state machines are implemented to support Intel QPI initialization, link parameter exchange, and facilitate BIOS error checking.
5. As routes are programmed by the BIOS, the processor interconnect simulation module validates that the destination processor ID exists and is connected as defined in the route table entry.

Debugging Tools

The primary tool used by BIOS engineers for platform debugging after hardware power-on (post-silicon) is ITP (In-Target-Probe). An ITP is a tool used to control the target hardware at the register level. The ITP tool allows full control of the target hardware with access to all chipset registers, processor

registers, and instructions. In addition, the ITP tool provides standard debugging features such as breakpoints and code stepping. Consequently, it is required that at a minimum the simulation supports the ITP debugging tool interfaces, so long as it supports all the standard ITP commands and features required by the BIOS. This allows the BIOS engineers to validate test scripts as well as BIOS code prior to hardware availability.

In addition, support of an advanced source-level debugger is desired. Simics support of the Eclipse debugging environment advances BIOS debug capability beyond the support of existing debug tools.

The key requirements of debugging tools used for BIOS debugging are:

- break on register read/write
- break on specific register value read/write
- break on memory location read/write
- break on I/O port read/write
- conditional break at a specific location in code
- access both Machine Status Registers (MSRs) and Chipset Status Registers (CSRs)
- examining processor state (for example, dump processor status registers)
- stack trace
- source level debug, code stepping, variable examination, and so on
- support of microcode update
- read/write MSRs or CSRs manually
- read/write I/O ports
- ability to examine current memory map

Multiprocessor Support

In large multiprocessor systems there are many components that are initialized in parallel. A simulation tool must provide multiprocessor support to enable testing of parallel flows.

Most multiprocessor systems today contain multiple memory controllers to support balanced system performance. To enable faster boot times, the BIOS MRC executes memory initialization in parallel across the memory controllers in the system. A key component to the BIOS MRC parallel execution is consolidation and communication between the separate BIOS MRC initialization threads. To support testing of this key performance component the simulator must support multiprocessor execution and BIOS MRC parallel execution.

"Key to BIOS MRC parallel execution is communication between the BIOS MRC initialization threads."

Simics supports multiprocessor environments in a deterministic manner using well-defined time slices. This can restrict the developer's ability to find parallel execution bugs. It is possible to find timing bugs with Simics, but special care must be taken to systematically look for them.[6] To debug multiprocessor

issues in Simics, parameters are modified in a methodical approach to alter the execution flow and carefully change the behavior. A testing framework (Sim-O/C) has been developed to address discovery and debugging of parallel execution faults.[7].

Validation of BIOS

Validating BIOS using simulation tools in both pre-silicon and post-silicon environments requires specific features to be provided by the simulation tool. The goals of the features are to provide mechanisms to maximize code coverage of the BIOS and automated backend testing.

"To enable BIOS validation all possible supported configurations must be supported."

Configuration Flexibility

To support the validation of BIOS all possible supported configurations must be supported. In addition, an efficient simple interface must be provided so configurations can quickly be changed during development and testing.

A Python-based tool was developed using the Simics Extension Builder package to provide a graphical user interface to easily configure Simics based on parameters in a platform-specific configuration file. The tool creates a customized Simics session script based on the parameters set by the user. Options displayed are based on the platform-specific configuration file to ensure only supported configurations are configured. The tool supports changing:

- Platform
- Number of processors
- Number of cores
- Number of threads
- Memory topologies
- Memory DIMM types, sizes, and so on
- Processor interconnect topology
- Specialized modes (such as manufacturing mode)

The tool updates available options based on other selections. For example, depending on the platform selection, the available options for processor, cores, and so on will automatically be updated. See Figure 7.

Fault Injection

Validation requires the ability to inject faults into the system as part of BIOS regression testing. These faults can range from PCIe parity errors, network errors, and memory errors, to disk failures, processor interconnect link failures, and power and thermal issues.

The Simics memory space infrastructure can be used to create a fault injection module without affecting runtime performance under normal circumstances.[8] In this case, a private memory space is created for the specific device into which

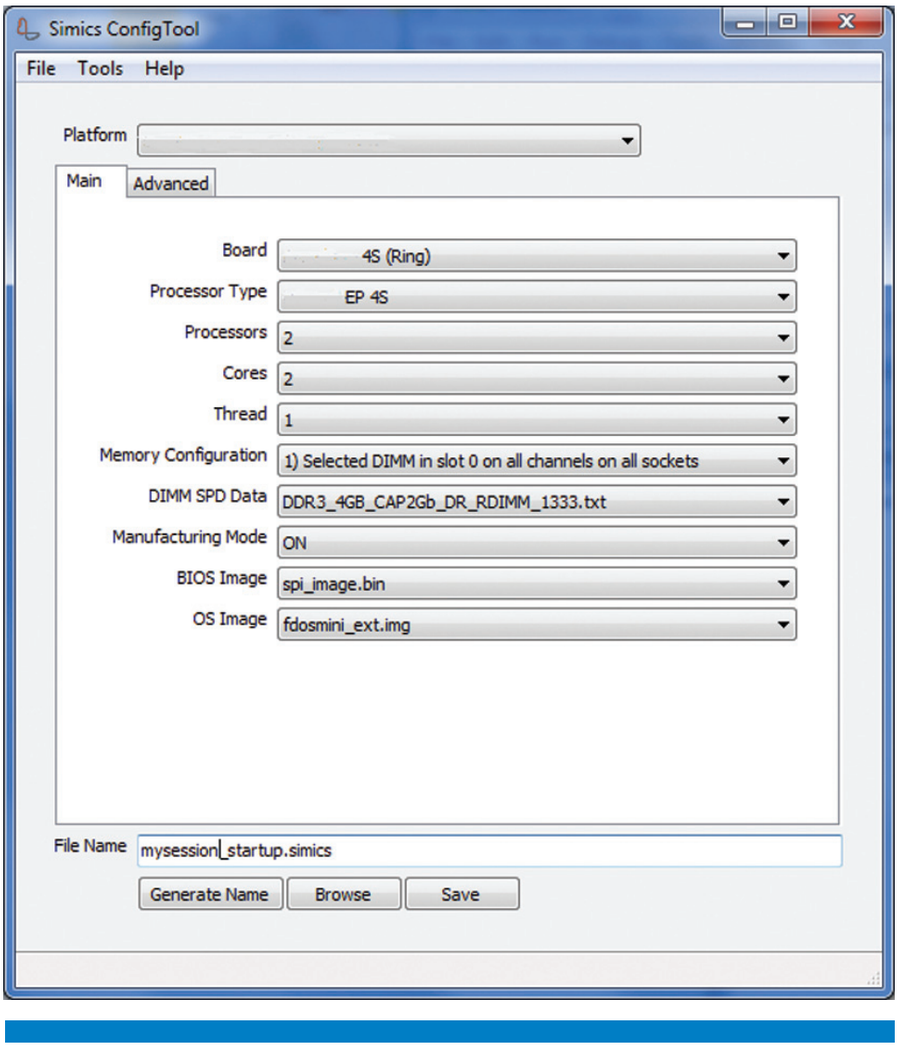

Figure 7: Configuration tool screenshot
(Source: Intel Corporation, 2013)

you wish to inject the error that is attached to the main memory space. The private memory space goes through a fault injector module prior to issuing the request. Scripting can be used to map or de-map the private memory space. The injection module can be implemented with scripts or as a DML module. A DML module is preferred since it will support checkpointing and reverse execution.

"Scripting can be used to map or de-map the private memory space."

A second methodology is to create a fault injection module that resides in the data path for the device and can be interacted with via scripts. In this case the fault injection module receives all the data. It is not selective and will cause some performance degradation. However, it has more flexibility in what type of errors can be injected (such as, for example, spurious data anomalies like line noise).

BIOS' primary function is hardware initialization. Consequently, the ability to inject faults in the initialization environment is required. Injecting errors into the initialization environment requires support of server management modules and system error interfaces:

- Machine Check Architecture (MCA), including System Management Interrupt (SMI) support. An SMI is raised when an error condition occurs. BIOS' SMI handler then interacts with the MCA to determine the type of error and take action if necessary.
- Baseboard Management Controllers (BMC) simulation: Ideally running the BMC firmware in conjunction with BIOS would provide the most benefit. When a management controller is present BIOS' role in error recording and recovery is limited and the management controller and firmware take on the primary responsibility.
- A centralized tool that allows injection of errors in a dynamic fashion.

"The simulator must support selection of a variety of configurations in an automated fashion."

Automation

The simulation tool must support the ability to configure and run a variety of configurations in an automated fashion. This enables automated regression tests to be run as part of the BIOS nightly builds and supports validation teams that are creating validation suites for hardware.

In conjunction with the configuration tool mentioned in the section "Configuration Flexibility" and the Simics session script architecture, automated testing infrastructure can run a variety of configurations with ease.

To improve the overall BIOS stability and enable the detection of defects earlier in the BIOS development schedule, a server farm can be used to automate simulator testing on every BIOS check-in. The implementation incorporates a source control system, build infrastructure, database subsystem, test launch and monitoring infrastructure, and a simulator server farm running on multiple virtual machines (VMs). See Figure 8.

The flow of the simulator test server farm is:

1. A user checks in code into the source control system.
2. The build system is triggered by the check-in. The BIOS binary for a variety of targets is built and the binaries are placed on a file server with the status of the build recorded in a database.
3. The test launch service finds that a job is available in the database and initiates a request to the test server to run a simulator test on a specific VM. A monitoring agent is activated to manage the request.
4. The test server receives the request and queues the job for the specified VM. When the VM becomes available, the test server pushes the job to the simulator and updates the database with the status of the job and VM.
5. The test monitor on the specific VM receives the test request and kicks off a simulator automated test.

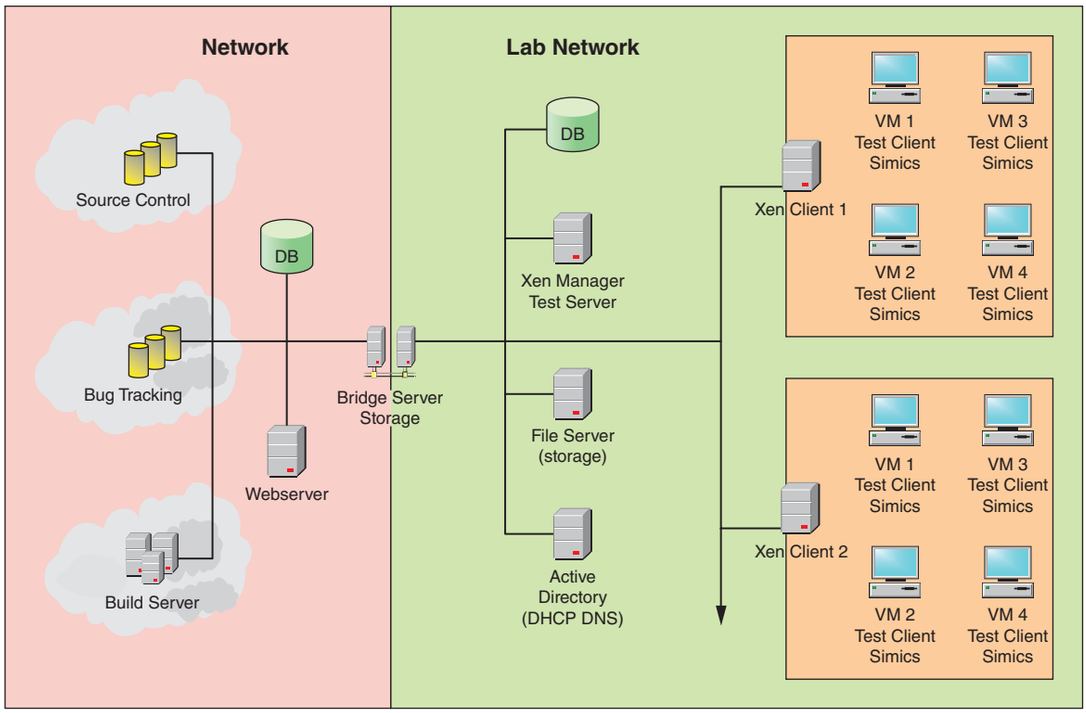

Figure 8: Simics test server farm
(Source: Intel Corporation, 2013)

6. The simulator performs all the tasks of the specified job request initiated from the test server.
7. After the job is complete, the simulator pushes the logs to the file server for examination by the test server.

The infrastructure can use the package update utility tool described in the section "Register Modeling" and the configuration utility described in the section "Configuration Flexibility" to update packages and create session scripts for automated Simics runs.

A server test farm is critical in both pre- and post-silicon development to ensure a high quality BIOS. In addition, in a post-silicon environment it can expose differences between the simulation and the initial power-on hardware that may be due to simulation bugs, discrepancies in specifications, BIOS bugs, or hardware bugs.

"A server test farm is critical in both pre- and post-silicon environments."

Summary

Creating a simulation tool to support development, debugging, and validation of BIOS presents specific requirements. BIOS requirements change depending on whether development is in the pre-silicon or the post-silicon environment.

The initialization state that the BIOS runs in requires more features, depth, and debugging aids than the runtime-state–based software like an operating system.

Simics' open architecture provides the means to create Simics enhancements in the area of utilities and core simulation behavior to meet the specific requirements of BIOS pre- and post-silicon development.

References

[1] Virtutech. Modeling your System in Simics. 2009. Revision 3004 http://www.virtutech.com/files/manuals/modeling-your-system-in-simics_0.pdf

[2] Alexey Veselyi, John Ayers. "Early Hardware Register Validation with Simics" in Intel Technical Journal 60, 2013.

[3] JEDEC® Solid State Association, September 2012. JEDEC® Standard DDR4 SDRAM. Reference number JESD79-4 http://www.jedec.org/

[4] Intel Corporation. "An Introduction to the Intel® Quickpath Interconnect." 2009. Document number 320412-001US

[5] Gurbir Singh, Robert J. Safranek, and Robert A. Maddox. Weaving High Performance Multi-Processor Fabric: Architectural Insights to the Intel® QuickPath Interconnect. Intel Press, 2009.

[6] Jakob Engblom. Simics and Multicore Systems Development. Virtutech, 2009.

[7] Tingting Yu, Witawas Srisa-an, and Gregg Rothermel. "Sim-O/C: An Observable and Controllable Testing Framework for Elusive Faults" in Intel Technical Journal 60, 2013.

[8] Jakob Engblom. Blog – Making a Faulty Serial Port. - Virtutech, 2012. http://blogs.windriver.com/tools/2012/01/making-a-faulty-serial-port.html

Author Biography

Steve Carbonari is a senior software engineer at Intel. He is currently working on architecting Simics simulation environments for BIOS development and debugging. He has ten years' experience in UNIX kernel development and over ten years' experience in system software and architecture. He holds an MS degree in Computer Science and a BS degree in Mathematics/Computer Science.

SIMICS*–SYSTEMC* INTEGRATION

Contributors

Asad Khan
Data Center Group and Storage Group, Intel Corporation

Chris Wolf
Data Center Group and Storage Group, Intel Corporation

In this article we discuss the integration of SystemC* architecture models with Intel's Simics Virtual Platform technology. Using a proof of concept, integration and synchronization steps and corresponding performance challenges of the integrated platform are highlighted and solutions developed. A logging and a save/restore SystemC API are added for seamless integration into Simics and to take advantage of Simics Checkpointing. Second level integration optimizations are implemented in the form of temporal decoupling. A complete software stack is ported to the integrated platform for system validation and software use cases including BIOS and OS boot, firmware and drivers development, and application stack execution demonstrating early software development with virtual platform (VP) methodologies.

Introduction

Simics[3] is the tool of choice for virtual platform modeling for many of the ongoing projects within Intel to enable software shift-left initiatives. Simics supports functional modeling at speeds of the order of 10-100 million instructions per second (MIPS) for fast OS, firmware boot, and software development. While Simics provides virtual platform models for many of the mainstream Intel architecture core/uncore based subsystems, and board level platform models, there are still many internal and external intellectual property components that need to be developed using SystemC[4], for the reasons of modeling fidelity and the use of standardized modeling environments. IEEE SystemC 1661 (Accelera) is the defacto industry standard for both functional and performance modeling at the system level.[5][6] Many teams within Intel are doing their model development using SystemC, while the same is true for intellectual property models developed in the industry by big and small intellectual property houses alike.[7] Besides standardization, SystemC also has the advantage of using models developed in such a way to serve both functional and performance needs of designers and software architects through the use of advanced modeling artifacts. These models can then be integrated with Simics to enable a complete platform for full software stack debug and development.

Different modeling semantics are provided by SystemC to serve both functional and performance modeling domains. Simics on the other hand supports functional models only for fast performance and software stack development. Any SystemC modeling paradigms are more magnified upon integration with Simics due to their impact on the performance of the integrated platform. Simics is also a complete simulation environment with supporting tools to enhance the users' experience. Advanced logging and

checkpointing features are part of the user experience to aid debuggability. Integration of a SystemC model should maintain the user experience by extending the Simics features to SystemC, while maintaining performance of the overall platform commensurate with Simics standalone platform performance.

Simics SystemC Integration Overview

Simics uses a time slice model of simulation, where each master is assigned a time slice for execution before control is passed to the next. Usually a master is a processor module implemented as an instruction set simulator (ISS), though this is not a restriction. In a functional simulation, an ISS runs a given number of instructions within its time slice, where each instruction represents a processor cycle. Any memory accesses corresponding to an instruction are blocking completing in zero time. The simulation model of Simics imposes restrictions on any co-simulation environment with corresponding integration challenges.

"Integration of a SystemC model should maintain the user experience by extending the Simics features to SystemC, while maintaining performance of the overall platform"

There are different ways in which a SystemC model can be simulated with Simics for an integrated platform. Either the SystemC kernel can be controlled by Simics, or run independently of Simics. This article addresses Simics controlling the SystemC kernel.

The communication model for Simics and SystemC co-simulation is asynchronous. For Simics controlling SystemC, the communication happens when there are memory, I/O, or configuration accesses between the two simulation environments based on the underlying memory map, or when interrupts occur from SystemC devices. Besides the apparent communication interaction, control is passed to SystemC by the Simics simulation engine when a scheduled SystemC event needs to be triggered.

The Simics/SystemC Bridge module[1] is the interface between Simics and SystemC, implementing the interaction between the two simulation engines. It provides a functional view of SystemC by implementing the interfaces for Simics to talk to SystemC. Similarly it lets SystemC access Simics' interfaces for upstream memory accesses and interrupts. It also encapsulates timing aspects of the co-simulation by synchronizing the two schedulers and their events.

For Simics controlling SystemC, the co-simulation runs in a single thread with a context switch between the two schedulers passing control from one to the other. The two schedulers are temporally coupled, meaning that they are synchronized at the interface point. Simics runs ahead of SystemC, and when the two schedulers communicate due to a memory, I/O, or configuration call, timing synchronization takes place. Temporal coupling however is not a requirement, as discussed later in this article, and an alternative synchronization mechanism of temporally decoupled schedulers is also discussed. SystemC however never runs ahead of the Simics time in any of these models.

Context switching between Simics and SystemC simulation engines is expensive from a performance point of view. This is especially true as the complexity of devices modeled using SystemC goes up and the frequency of switching also increases. This article delves into aspects of performance and solutions to address these issues. It also addresses how to best write the SystemC models to deal with performance bottlenecks. Also mentioned are the SystemC APIs for adding logging and checkpointing to SystemC to enable seamless integration with Simics and to extend such Simics' features to SystemC.

"Context switching between Simics and SystemC simulation engines is expensive from a performance point of view"

The article also deals with temporal decoupling between the two simulation engines and how temporal decoupling impacts performance. The co-simulation still runs as a single thread, with Simics controlling SystemC engine, but SystemC is set up as a master in itself by assigning a time slice to it and is scheduled through Simics. The two simulation environments are temporally decoupled by running SystemC for a fraction of duration of the time slice. Performance improvements achieved through temporal decoupling are presented along with why temporal decoupling makes sense. Case studies are used to corroborate the different methodologies and corresponding performance improvements.

Simics SystemC Bridge Module

The Simics/SystemC Bridge[1] provides synchronization mechanism between Simics and SystemC schedulers. Simics and SystemC schedulers run in a co-simulation mode with Simics being the master scheduler. The SystemC scheduler is triggered by the Simics master scheduler through the bridge module. It accomplishes the following:

1. Plays catch up with Simics to synchronize the two schedulers in time. It accomplishes this by running sc_start() for the time difference between Simics and SystemC simulations to synchronize the two schedulers.
2. Following synchronization of the two simulation engines, allows any transactions from Simics to SystemC or vice versa to execute as non-preemptive transactions.
3. On completion of a transaction between Simics and SystemC, the bridge posts any pending SystemC events onto the Simics event queue for future scheduling.

As illustrated in Figure 1: Simics SystemC Integrated Virtual Platform, the Simics platform interfaces with SystemC through the Simics frontend module, which provides C/C++ APIs to talk to the SystemC module. Depending upon the functionality of the SystemC module, different interfaces are implemented. A typical case is the memory interface providing visibility into memory, I/O, and PCI configuration spaces of the device.

An example of a PCIe device is used for illustration only, as the methodology is applicable to any SystemC device or System model. For a typical PCI device implemented using SystemC as shown here, the device is discovered during the BIOS enumeration stage, where the device's bus, device, and function IDs (BDF) are discovered, and then mapped into the memory and I/O spaces of the platform through the BIOS or OS configuration step by accessing its PCI configuration

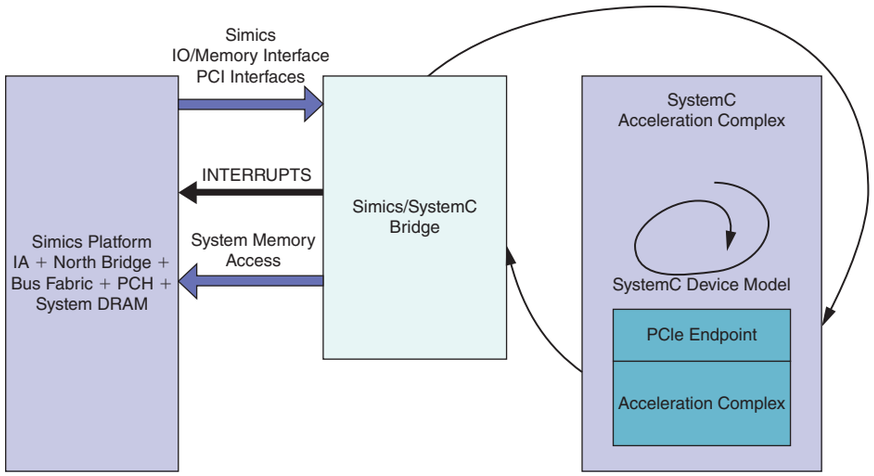

Figure 1: Simics systemC integrated virtual platform (Source: Intel Corporation, 2011)

Base Address Registers (BARs). At this stage, the device is programmed to be addressed through its memory or I/O spaces. The bridge also provides a mechanism for the SystemC module to do any upstream direct memory accesses (DMAs) and send Interrupts to the core. This is accomplished by accessing handles to the platform to access its functionality. Both signal and Message Signal Interrupts (MSIs) can be implemented through the appropriate APIs.

SystemC Device Model

The SystemC model is implemented hierarchically with a top and a SystemC TLM-2.0 (IEEE TLM 2.0, 2008) interface. An adapter with a SystemC TLM-2.0 interface on one side and a functional interface on the other connects the Simics frontend and SystemC device models—all are part of the Simics/SystemC Bridge component. Transactions from the Simics frontend are packetized using TLM-2.0's "tlm_generic_payload" tokens and sent over the TLM-2.0 interface to the corresponding SystemC device models. Information that can't be encapsulated using tlm_generic_payload can be passed to SystemC device model through the extension mechanism of the TLM-2.0 standard.

"An adapter with a SystemC TLM-2.0 interface on one side and a functional interface on the other connects the Simics frontend and SystemC device models"

The SystemC device model in our example system encapsulates PCIe endpoints incorporating PCIe configuration registers, along with the device functional model. The device is accessed through its memory or I/O space registered using the PCIe BAR configuration registers. Any upstream transactions from the device are sent to the Simics platform through the TLM-2.0 interfaces.

Simics SystemC Integrated VP Performance

The SystemC event simulator accommodates both event-based and clock-based modeling semantics. The latter may be used in some performance modeling scenarios to accurately predict the system performance and highlight issues like

deadlocks and starvation. While the integration methodology discussed here poses no restriction on SystemC modeling for integration with Simics, it does present different challenges to the overall platform based on the type of SystemC models integrated. Simics, being a functional simulator, prefers the SystemC models to be functional; however it does not impose this restriction. So for this discussion we will assume a modeling semantic based on events only, removing the performance penalty and redundancy imposed by clocked models. An event-based methodology has the capability to add every single event of significance for both functionality and performance. In the following discussion we describe the performance, corresponding optimizations, and their side effects for SystemC models.

SystemC Code Refactoring

SystemC uses processes for concurrency. Two types of processes are defined by the standard:

- SC_METHOD processes that run to completion when triggered. These are sensitive to events and port signals.
- SC_THREAD processes run for the duration of the simulation through an infinite loop. These can be halted in the middle of the process through wait() statements awaiting certain events. Upon a process halt, the state of the thread is saved on the heap or the stack.

"Speed incompatibility between the two simulation environments leads to the slowdown in speed of an integrated Simics SystemC VP"

SC_THREAD() processes are expensive from a simulation point of view because of the inline wait statements, which cause the context to be stored. SC_METHOD() processes on the other hand run to completion without any side effects. The resulting performance hit is more pronounced for an integrated model due to other performance issues discussed here. It is proposed to replace any SC_THREAD() processes with the more efficient SC_METHOD() processes with the wait calls replaced with corresponding SystemC sc_event semantics. In one case, replacing 30 odd SC_THREAD() with SC_METHOD() processes lead to a performance gain of about 15 percent.

Performance Optimizations

Simics VP functional simulations run at speeds of the order of 10–100 MIPS. Simics employs a non-preemptive time slice model for any master modules, where instructions of the order of 100,000 are assigned to a given master per time slice before control is passed to another master. Another factor contributing to speed is zero delay blocking transactions with no side effects. On the other hand, cycle- or clock-based simulation as in the case of SystemC device model achieves speeds only of the order of hundreds of kilocycles per second (KCPS).

Speed incompatibility between the two simulation environments leads to the slowdown in speed of an integrated Simics SystemC VP. Simulation performance was measured for an integrated VP compared to a standalone Simics VP. In this case a standalone Simics platform would boot Fedora OS within few minutes. Co-simulation with SystemC slowed down the overall simulation speed up to an order of 100,000:1. It needs to be clarified that the

SystemC model was a complete subsystem with its own firmware and hardware and not a simple memory-mapped device. While SystemC models with timing granularity of the order of clock periods are to blame, the end result is a direct consequence of the need to context-switch between Simics and SystemC at every event that needs to be triggered. This slowdown is worse for clock-based models, and is improved but not eliminated using an event-based methodology as discussed in the section "Temporally Decoupled VP Performance".

Since a complete software stack is to be run on the platform with billions of instructions, some mechanism had to be devised to speed up the integrated VP. As shown in Figure 2: Simics/SystemC Clock Scaling, performance improvement of orders of magnitude is achieved by down-scaling the SystemC clock frequencies by a factor of *X*. This implies slowing down SystemC simulation by a factor of *X* compared to the Simics clock. This indirectly translates into reduced context switching between the two simulators by the same factor, increasing the overall system simulation speed. A scaled-down factor of 10,000 was used to achieve optimal performance—a number obtained through empirical data. This translates to a SystemC clock frequency reduction by a factor of 10,000.

"…performance improvement of orders of magnitude is achieved by down-scaling the SystemC clock frequencies by a factor of X"

There are side effects to slowing down the SystemC model compared to overall platform speed. One obvious side effect is the slowdown of any stack that

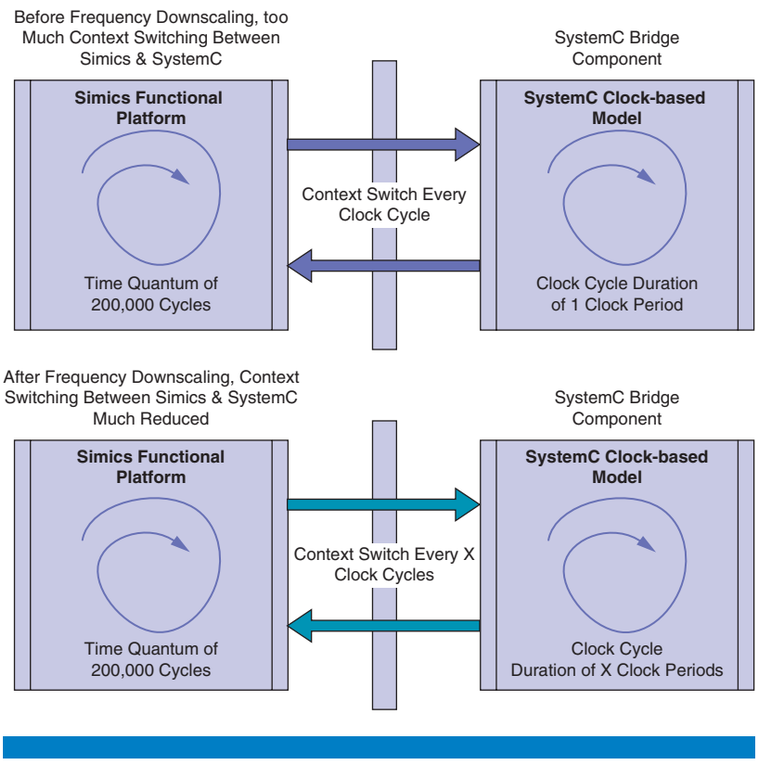

Figure 2: Simics/SystemC clock scaling (Source: Intel Corporation, 2011)

"Since the SystemC model is running slower, any timeouts for code running on the platform may need to be increased corresponding to the clock scaling factor to disable premature timeout expiration"

uses the SystemC models because it takes longer to run. However this cost is bearable because the platform boots much faster compared to a non-scaled model. There are other side effects as well. Since the SystemC model is running slower, any timeouts for code running on the platform may need to be increased corresponding to the clock scaling factor to disable premature timeout expiration. Another side effect relates to any tight polling loops for code running on the core monitoring the status registers on the SystemC side. Polling leads to a context switch, slowing down the simulation, especially when there is little useful work done by SystemC. The polling frequency of these status registers had to be reduced because of the slowdown of the SystemC models. This was accomplished by adding stall delays in the Simics/SystemC Bridge whenever SystemC status registers were polled.

Performance Metrics

Performance optimizations discussed so far are used to determine the overall performance gains. Base results are obtained using clock scaling, and any gains over and above that are highlighted for polling and SystemC process optimizations. "Table 1: Performance Optimizations in numbers" represents one set of data for a set of software tests running on the platform and actively exercising the Simics/SystemC interface. Any other results would vary depending on the type of the models and the frequency of Simics/SystemC interaction. However the numbers below represent a general trend highlighting the fact that performance optimizations implemented make significant gains in overall Simics/SystemC VP performance.

	Boot Time	Setup Time w/o Stall	Setup Time w/ Stall	Test Time w/o Stall	Test Time w/ Stall
Poll Mode Driver					
SC_THREAD()	197	376		408	230
SC_METHOD()	199	353		375	212
Interrupt Mode Driver					
SC_THREAD()	197	778	332	670	475
SC_METHOD()	199	670	313	660	452

Table 1: Performance Optimizations in numbers
(Source: Intel Corporation, 2011)

The matrix represents two sets of data—setup time for the SystemC device model and the test execution time. Columns represent the performance comparisons for the two sets with and without stall cycles. Rows represent the additional SystemC process optimizations using code refactoring. The following performance improvements were observed:

- A gain of 30–60 percent using stall cycles when polling status registers.
- A gain of 3–15 percent through replacement of SystemC_THREAD() with SystemC_METHOD() processes.

Simics SystemC VP—Temporal Decoupling

Simics SystemC integrated VP uses temporal coupling between the two environments to keep them in time synchronization. The two simulators interact at the interface layer within the bridge module. However, for functional VP, as long as event ordering is preserved on both sides, VP works correctly and there is no need for time synchronization.

Temporal decoupling takes advantage of the lack of timing interdependency between the two simulators by letting them go out of synch with each other. This is done by making SystemC a simulation master by assigning it an execution time slice, and placing it on Simics' event calendar. When SystemC gets scheduled by Simics, it is run for a fraction of the time slice—a number that can be dynamically changed during simulation through a Simics' attribute. The idea is to let SystemC get more execution cycles during busy periods, and only short durations during idle periods. No SystemC events are posted on the Simics event queue, leaving only time slicing to schedule SystemC.

"Temporal decoupling takes advantage of the lack of timing interdependency between the two simulators by letting them go out of synch with each other"

A side effect of temporal decoupling is that Simics' time runs ahead of SystemC time, and the aggregate time difference between the two schedulers keeps increasing as the simulation progresses, except for the case when SystemC runs for the entire time slice duration. Another consequence is that SystemC device interrupts encounter a scheduling delay up to a maximum of the time slice duration. This is not a problem for most functional VPs, except when there are real-time performance requirements.

Temporally Decoupled VP Performance

Through temporal decoupling, a much smaller scale factor (100) can yield performance similar to the temporally coupled case (scale factor of 10,000), thereby relieving some of the side effects of clock scaling. Also at the cost of a little bit of performance, time scaling of SystemC can be totally removed as shown in "Table 2: Temporal decoupling using SystemC time slicing"

SystemC Time Slice sec	SystemC run time psec	SystemC Scale Factor	Fedora OS Boot Time
.001	10,000	100	17:50
.001	1,000,000	1	24:15
.001	1,000,000	10	28:30
.001	1,000,000	100	21:45
.001	100,000	1	23:30
.001	100,000	10	18:55
.001	100,000	100	18:55

Table 2: Temporal decoupling using SystemC time slicing
(Source: Intel Corporation, 2013)

SystemC is allocated a fixed time slice of 1 msec, while the "run duration" of the time slice is changed as shown along with the scale factor. It is pertinent to note that a temporally coupled simulation took 19 minutes to boot. A temporally decoupled model with no scaling (scale factor of 1) achieves the same state in between 23 and 24 minutes. These numbers should be used only as a reference of the trends and not as a benchmark to determine the overall simulation speed, as the speeds are also a factor of the host OS, host machine, and complexity of the models.

Simics SystemC APIs

For effective integration of SystemC models with Simics, a set of APIs have been developed to take advantage of Simics features like checkpointing and advanced logging. These are discussed in detail here.

SaveRestore Checkpoint API

"The objective of the SaveRestore API is to make SystemC independent of the Simics checkpoint API for compiling and running standalone"

Simics[3] provides a checkpoint capability to store the state of a platform. Simics uses an attribute mechanism to store the state of a model. Attributes can be an integer, floating point, string, Boolean, list of values, a dictionary, or other compound types. For adding checkpointing to a model, its entire state needs to be saved and restored. For large amounts of data the best way is to save the state as an image, which stores data as a contiguous array of characters or any other data type. For saving model state as an image, a pointer to the image and corresponding length are provided. Simics provides a C++ API to add checkpoint capability to C++ or SystemC models.

The Simics checkpoint API is a complete API for saving/restoring the state of a model. However, it is intrusive if added to SystemC in that the models become dependent on the Simics headers and lose their ability to be compiled and run as standalone models. The objective of the SaveRestore API is to make SystemC independent of the Simics checkpoint API for compiling and running standalone. At the same time, when running as a VP with Simics, this API ties SystemC state to the Simics checkpoint database.

A "SaveRestore" base class is declared in Figure 3: SystemC SaveRestore API hierarchy, which sets up the base functionality for registering an attribute. Another "SrAttrBase" class describes the functionality for the data type to be saved. All attribute types derive from SrAttrBase and provide a set of functions to get/set the value of the attribute. A set of C functions are provided to register a given SystemC model attribute by calling the attribute registration function in SaveRestore class. These functions set up the attributes and provide links to the attribute set and get functions. Attribute set/get functions are to be called when saving or restoring the state of a model. For standalone SystemC models, the attribute registration function is a dummy function, which lets the standalone model compile, but does not have any functionality.

The ScdSaveRestore class ties up the SystemC model with the Simics database for checkpointing in a VP. This class includes Simics API headers linking the Simics

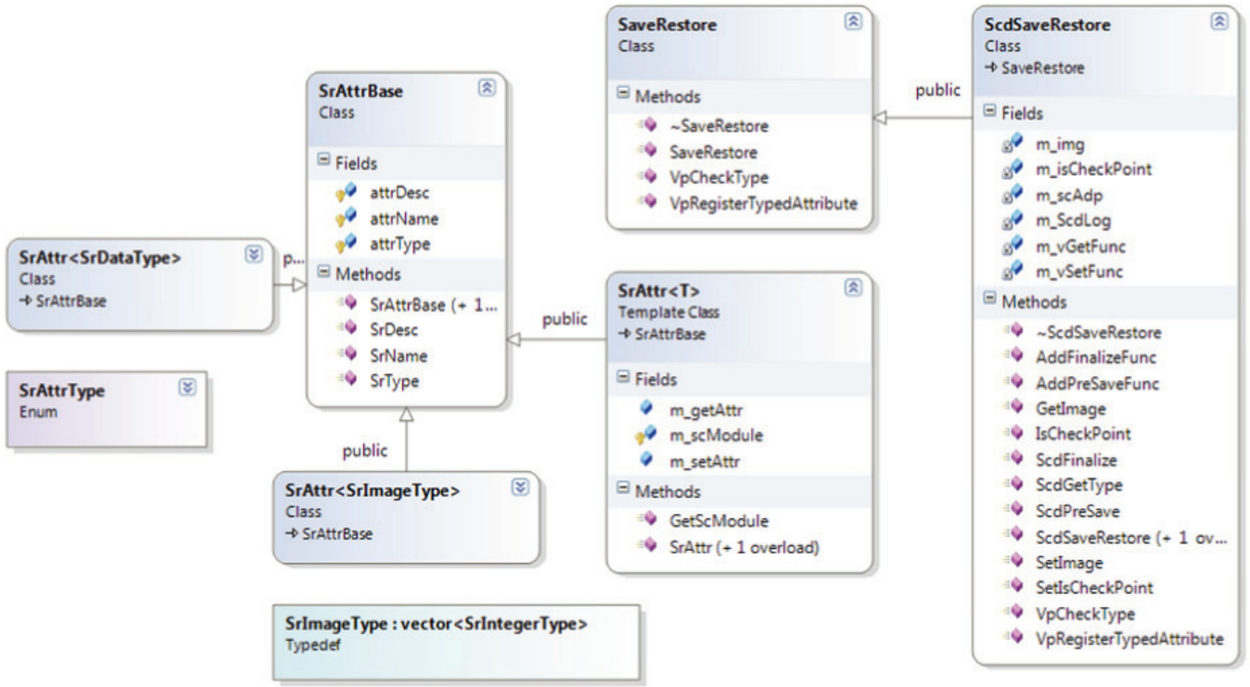

Figure 3: SystemC saverestore API hierarchy
(Source: Intel Corporation, 2013)

VP with the SystemC model. It derives from the SaveRestore class and overrides the base functions in SaveRestore for registering the attribute with Simics. In a Simics SystemC VP, the Simics/SystemC bridge module[1] instantiates the ScdSaveRestore class and registers it with a handler interface. This handle to the ScdSaveRestore class is passed to the SystemC model, tying the SystemC checkpoint state to the Simics database. In this setup any SystemC model attribute registers with Simics along with its Save and Restore functions.

This setup is enough to save the steady state behavior of the SystemC model, such as, for example, OS boot and the device initialization state. For dynamic SystemC checkpointing, the API has to be extended to include TLM-2.0 generic payloads and payload event queue (PEQ) semantics of SystemC. Work has been done on this front[2], but is outside the scope of this writing.

"Restoring a given Simics/SystemC VP restores the Simics time to the simulation time at which the checkpoint was taken"

Updating SystemC Time upon Checkpoint Restore

Restoring a given Simics/SystemC VP restores the Simics time to the simulation time at which the checkpoint was taken. When the checkpoint is restored, the SystemC kernel hasn't started running, and SystemC time is set to SC_ZERO_TIME. Restoring SystemC time to Simics simulation time would require running SystemC for the duration of the checkpoint. When

updating SystemC time this way, a significant performance drag was noticed at checkpoint restore, the duration of which varied based on the time for which SystemC had to be run to bring it in sync with Simics time.

A solution was to update "current_time" sc_time variable of SystemC to the Simics time after SystemC construction and elaboration phase. This time represents the value of the current simulation time within SystemC. As part of system restore this value was updated to current Simics time, leading to quick turnaround of the restore point, and preventing the need to run SystemC to the current simulation time.

SystemC Logging API

Simics provides a logging API for native C++ and Design Modeling Language (DML) models. The API classifies logging into groups and verbosity levels. Additionally Simics has the capability to halt the simulation based on a log message. This becomes extremely powerful for debugging the model. Although there is a standard logging mechanism within SystemC, there was no method for Simics to have full control of the SystemC logging. This drove the need for a generic logging API for SystemC.

"Studies show that performance of the integrated platform falls somewhere between the performance of a standalone Simics VP and a standalone SystemC platform"

The logging API (as well as the SaveRestore API) is based on a handler mechanism. A user can redefine the logging handler to intercept logging calls and redirect them to their own API; for example, in model X, the user would get a log handler object using its name and type.

```
AcLog = HandlerInterface<VpLog>::GetHandler("main");
```

This would get a pointer to the main handler. Using this handler, log statements are implemented as follows.

```
VPLOG_INFO (AcLog, LOG_VERBOSITY, LOG_GROUP, "value = %x", x)
```

The main logging handler is defined and registered with Simics within the Simics/SystemC bridge module tying it to the Simics logging API. This API also allows us to debug SystemC models outside of Simics and with the same logging API.

Summary

In this article integration of a complete system level SystemC model is presented with a Simics VP. Integration steps are highlighted along with the performance challenges due to the detailed nature of the SystemC model with its own firmware and functionality, and the different nature of the two simulation environments. Solutions to performance problems of the integrated platform are proposed to lead to a working VP for complete software stacks porting for debug and development. A set of APIs are developed for checkpointing and logging for the SystemC models to capitalize on the Simics checkpoint and logging APIs for an integrated solution. Studies show that performance of the integrated platform falls somewhere between the

performance of a standalone Simics VP and a standalone SystemC platform. This model can serve as a blueprint for enabling early software development activity, including BIOS, OS, firmware, driver and test development.

Complete References

[1] Wind River Simics, "Wind River Simics SystemC Bridge Programming Guide," 2010.

[2] Khan, Asad et al. "SystemC Checkpoint extensions for Simics/ SystemC Virtual Platforms," Intel DTTC 2013.

[3] Wind River Simics, "Model Builder User Guide," WindRiver, 2012.

[4] IEEE SystemC - Modeling Language Specification. [Online] Available at: http://www.accelera.org/home.

[5] Khan, Asad, and Leaming, Taylor. "MVP – A Pre-Silicon System Validation Design Flow Using Virtual Platforms," Intel DTTC, 2010.

[6] Heraty, Paul, and Khan, Asad. "Integrating Imagination Technologies' Cores into Virtual Platforms," Intel DTTC 2011.

[7] Discretix, "Discretix CryptoCell" – http://www.discretix.com

Author Biographies

Asad Khan is a senior staff engineer at Intel Corporation. He received his PhD in Electrical Engineering from Northwestern University in 1996. Asad has almost 20 years of experience working in the electronic system level design space at Cadence, ARM, Marvell, and Intel Corporation. His current focus is on virtual platform methodologies for system validation, software development, and power analysis. Asad is based in Chandler, Arizona.

Chris Wolf is the Virtual Platform Enabling Group (VPEG) lead in Intel's Intelligent Systems Group's Platform Enabling and Development. He has 9 years of experience in system validation and emulation and in virtual platform methodologies. He has made significant contributions to the shift-left strategy on Rangeley, Coleto Creek, Island Cove, and Bell Creek, with a focus on virtual platform and hybrid virtual platform enablement for validation and sofware teams. He continues to be a strong advocate for VP usage within Intel.

POST-SILICON IMPACT: SIMICS* HELPS THE NEXT GENERATION OF NETWORK TRANSFORMATION AND MIGRATION TO A SOFTWARE-DEFINED NETWORK (SDN)

Contributors

Tian Tian
Datacenter and Connected Systems Group, Intel Corporation

"Today the speed of growth and demand for infrastructure requires infrastructure vendors refresh designs and innovate at a much faster pace than before."

This article focuses on post-silicon usage of Simics, and discusses the huge potential for this technology to impact the ongoing network transformation. Today the speed of growth and demand for infrastructure requires infrastructure vendors refresh designs and innovate at a much faster pace than before. This opens a huge trend towards a software-defined network (SDN), Network Functions Virtualization (NFV), and Intel® Open Network Platform (Intel® ONP). The recent movement builds on the strength of x86 common communication platforms, as well as OS solutions such as Wind River Open Network Software*, and dramatically reduces time to market for infrastructure vendors with reference platform and reference software stacks. The biggest challenge in this transformation is on the software side, because customers need to migrate legacy solutions to the new paradigm. The effort associated with the process to debug, test, and maintain this solution moving forward can be quite daunting. Simics is an essential technology in this ongoing transformation and is nicely positioned to make an impact with support for security, Intel® Data Plane Development Kit (Intel® DPDK), SDN, and Intel ONP as well as a long-term roadmap for future products and technologies.

The Era of Network Transformation—The Challenges and How Simics Fits

Today's and yesterday's telecom equipment depends on a complex mix of CPUs and network processors and customized ASICs purposely built for the targeted applications. These heterogeneous solutions often are difficult to program and scale poorly. The cost of sustaining and improving such a solution is quite high, with vendors occupied with the burden to maintain a variety of software programming models, tool chains, and skill sets throughout product life cycle. Network usage and demand is going through a major explosion and there is no sign of it slowing down. This data explosion puts tremendous pressure on communication and storage infrastructure vendors and forces the industry to innovate and refresh solutions at a pace previously unseen. The solutions must scale to handle exponential growth of demand and at the same time be flexible enough to adapt to new services and new requests.

To tackle such a complicated situation, it is essential to have a technology that can address this kind of heterogeneous environment. Simics is a proven technology that supports all major CPU architectures and can be extended

to support new devices, modules, and components. Compared to other simulation solutions, Simics has very fast simulation speed and is nicely suited for full system level firmware and software development as well as a network of systems. All these strengths put Simics in a unique position to help vendors and developers transition existing hardware and software solutions and to respond to changes and new features during this network transformation.

"…Simics has very fast simulation speed and is nicely suited for full system level firmware and software development as well as a network of systems."

People often have the misconception about simulation solutions: "if I already have my hardware, why do I still need to use simulation technology such as Simics? Why don't I just do everything on hardware?" It turns out that Simics can contribute a lot in post-silicon usage, especially in networked systems. In this article, we will share our experience here in actual usage to discuss post-silicon values, using Intel next generation communications platform (codenamed Crystal Forest[1]) as an example. We also share our excitement about opportunities in front of Simics in the ongoing network transformation. The Crystal Forest platform and its follow-on platforms, with their unique combination of control plane and data plane capabilities, as well as security and virtualization features, are leading vehicles for Network Functions Virtualization and software-defined network solutions. Simics has solid support for Crystal Forest platform (Figure 1) as well as future generations of Intel® Communications platforms based on the architecture codenamed Haswell. It has CPU support for the latest Intel® Xeon® CPU as well models for Intel® 89xx chipset and 82599 networking card. It also runs Intel® QuickAssist software suite and Intel DPDK software.[2] The platform itself is virtualization ready and can be used for related development and test effort.

A couple of trends stand out in particular during this era of network transformation. One is workload consolidation and repartition. There are a lot of different workloads and usage models driven by different components on the network (access, wireless base stations, routers and switches, intelligent edges). Previously, each device was purposely built just to handle specific functions and services. Although it is a solid approach to tackle the problem, the amount of time it takes to come up with a new solution or update tends to be long, as it often requires hardware, software, and a tool chain to change at the same time. Now vendors are looking into having general purpose CPUs perform more and more of these specific workloads for better scalability and time to market. This shift allows vendors to focus more on services and value-adding software rather than spending the majority of resources dealing with the burden to maintain multiple versions of hardware, software, and proprietary solutions, and to find and sustain the skill sets required to program these proprietary solutions.

Another trend is increased usage of virtualization and other hardware abstraction technologies in the embedded and communications space. Devices and resources are now starting to be managed intelligently in real time based on usage profile and consumption. For a long time, embedded and communication platforms have been lagging data center servers in terms of adaption of virtualization technology, partially due to proprietary

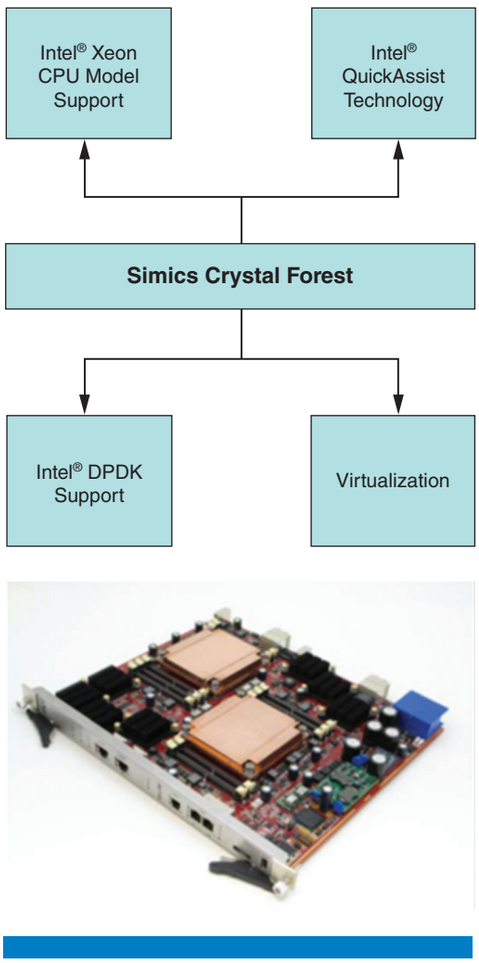

Figure 1: Simics support for Intel® next generation communication platform
(Source: Intel Corporation, 2013)

solutions. With the movement to general-purpose CPUs, suddenly adoption of virtualization technology becomes much easier.

"The technology supports a variety of architectures and can be a nice vehicle to begin repartitioning workload on future-generation hardware."

Simics is strong in both categories and is in a unique position to be a key contributor to this network transformation process. The technology supports a variety of architectures and can be a nice vehicle to begin repartitioning workload on future-generation hardware. Developers are the owners of virtual systems and can get backdoor access to virtual hardware information in a very developer-friendly fashion (run, stop, reverse execute, probe hardware states) regardless of security policies or virtualization partitioning. This "unfair" access breaks rules and allows developers to debug and test complicated scenarios with visibility in system states. Coupled with the highly integrated and easy-to-use Eclipse debug environment, developers will find utilities right at their fingertips when tackling challenging problems.

Post-Silicon Simics Usage Case Studies

In this section we share several real-life usage examples we encountered during the process of using Simics as part of our product development activities. We use the Intel® platform codenamed Crystal Forest as an example. Crystal Forest is a communications solution that is used by infrastructure vendors as a key building block for communication solutions. It is also an important piece for SDN, NFV, and Intel Open Network Platforms[3] (Intel ONP).

Before we start, we need to briefly touch on a few terms used here as we will use the shorter versions rather than referencing the full platform names:

- Crystal Forest Gladden[4] – Mobile platform name of Intel Xeon processor with Intel Communications Chipset 89xx Series
- Crystal Forest Server – platform name for Intel Xeon Processor E5-2600 and E5-2400 Series with Intel Communication Chipset 89xx Series

Concurrent Debug of UEFI BIOS (Virtual Platform and Hardware Platform)

Even when hardware platforms are already available, Simics can still contribute to the product development process by reducing the time spent debugging. During the board bring-up phase of the Intel Crystal Forest Server reference design, we successfully utilized Simics models of the target board to assist the debug process. The key components include two Intel Xeon Processor E5-2600 CPUs, four 89xx chipsets and two 10GbE cards. We have Simics models for all of these ingredients, and we used them for BIOS development.

"Even when hardware platforms are already available, Simics can still contribute to the product development process by reducing the time spent debugging."

During testing on the actual hardware platforms, we noticed an error message sometimes showed up during boot process and boards would hang afterwards. But the errors were sporadic and could not be reproduced every single time. Lab engineers were dealing with other issues and did not look deep into the problem. We loaded the same BIOS image onto the Simics ATCA environment and were able to recreate the same issue as shown in Figure 2. There are two windows shown here. The one in the back is VGA output showing BIOS is at the final stage of finishing setup. The second window shows the serial output and displays the same assertion error we saw on hardware platforms.

Once the issue was successfully recreated in Simics, the debug process became quite easy. Simics is a run-to-run repeatable environment, which makes it very easy to reproduce errors. We were able to step through the instructions and isolate the issue to a small module inside the test BIOS (Figure 3). Because in Simics one can save system states at the crime scene as a checkpoint, we provided the files to team members on a different geographic location and invited them check into the issue in parallel. They were able to recreate the error signature within minutes of getting the checkpoint and continued to debug.

In this example, we reliably recreated this error in Simics. On real hardware, the error showed up sporadically and was a lot of harder to recreate. Our debug data and trace collected in Simics isolated the issue within a module inside BIOS and later on the root cause was determined by our BIOS vendor. By

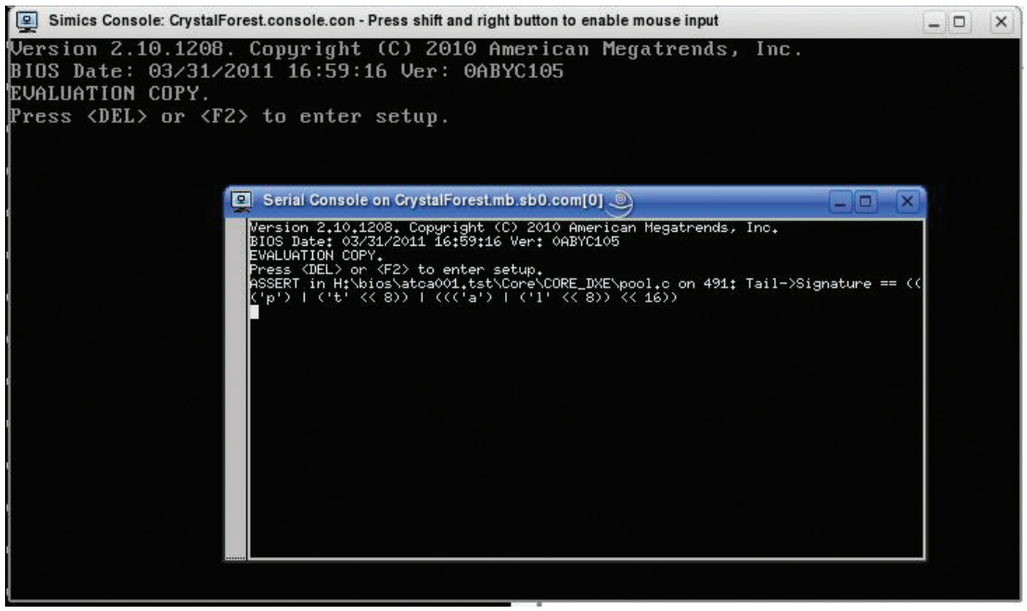

Figure 2: Simics ATCA used to debug OEM BIOS issue
(Source: Intel Corporation, 2013)

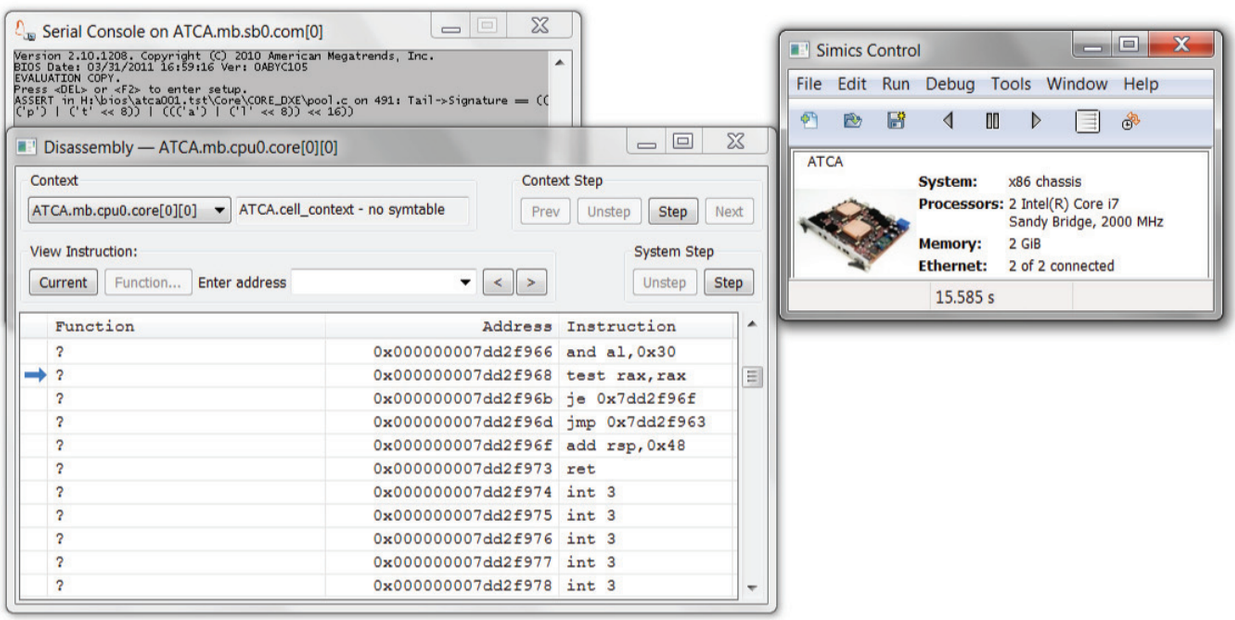

Figure 3: Single step through debug in Simics
(Source: Intel Corporation, 2013)

debugging the issue in Simics first, we saved precious time in our schedule. It was also a great experience where team members in different locations we able to contribute to the same debug process in real time.

Furthermore, this issue later on resurfaced on some other BIOS releases and in these situations we were able to immediately spot the issue and reuse the lessons from the Simics experience. The savings from this incidence alone is obvious across multiple releases. UEFI BIOS is a critical piece in a system solution; from this case study we can clearly see that Simics is a viable development and test tool for UEFI BIOS. Even after silicon is available, developers can get to the bottom of the software and firmware issues quicker and more efficiently from a collaboration point of view.

Simics x86 models also come with support for Intel® eXtended Debug Port (Intel® XDP) debugger and also can be integrated with other hardware debuggers (Figure 4). These debuggers are essential in the UEFI BIOS debug process on real hardware platforms. The scripts written for these hardware debuggers can use this as a bridge to get into the simulation environment, if needed.

"Simics x86 models also come with support for Intel® eXtended Debug Port (Intel® XDP) debugger and also can be integrated with other hardware debuggers…"

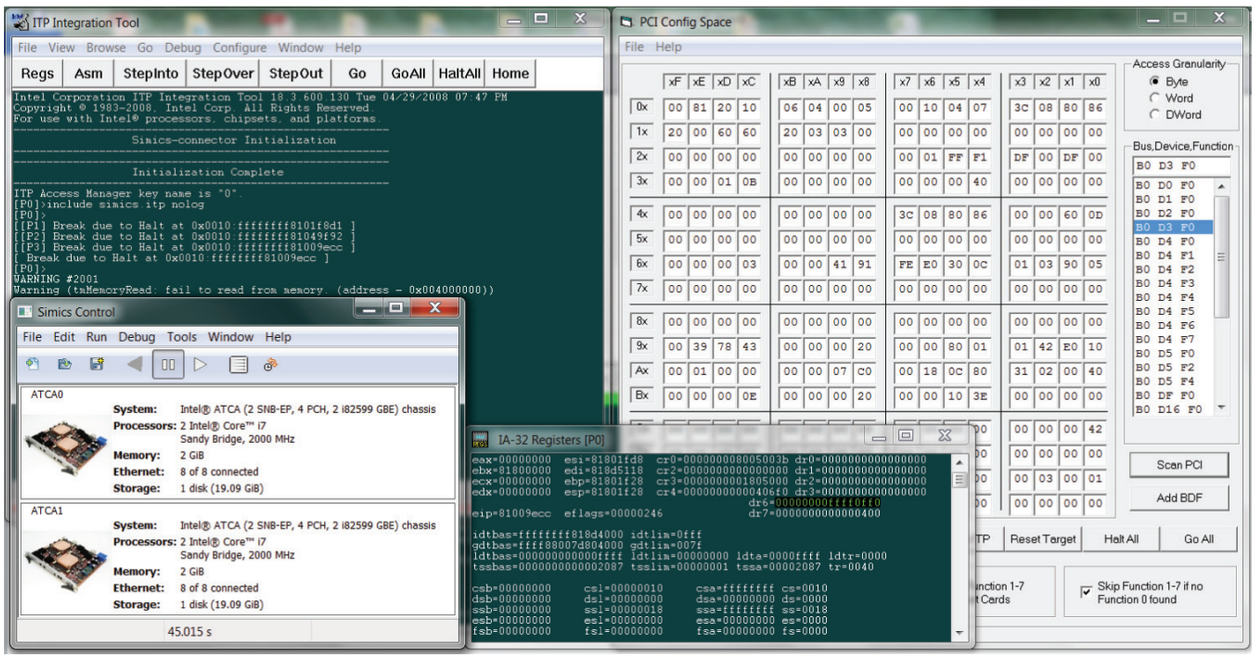

Figure 4: Simics virtual platform running virtual debug agent for Intel® XDP tool (Source: Intel Corporation, 2013)

Virtual Development Kit for Intel® QuickAssist

A security solution is an integral component of communications and is critical for solutions that control the safety and protection on the edge of the network. Intel® QuickAssist software has been used for a long time and continues to

"For software, things behave exactly the same from a functionality point of view whether it is running on hardware or running on Simics."

develop as an important piece of SDN and Intel ONP. Shown here is the modeling approach for an Intel QuickAssist hardware component (Figure 5). In this approach, Intel® QuickAssist hardware devices are modeled using Simics and the whole device functionalities are modeled. It simulates functional behavior of Intel QuickAssist and interacts with the software layers above. For software, things behave exactly the same from a functionality point of view whether it is running on hardware or running on Simics.

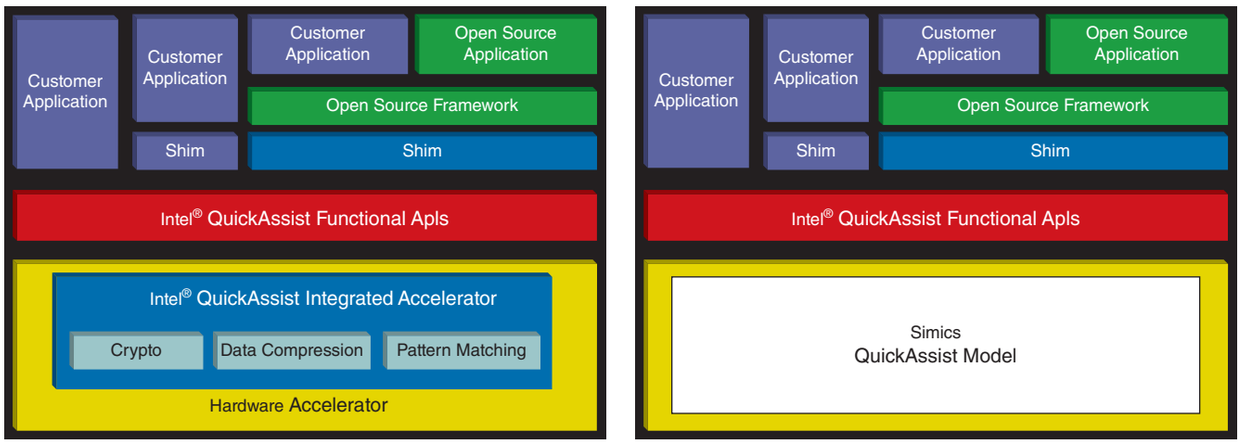

Figure 5: Simics Intel® QuickAssist modeling—DML
(Source: Intel Corporation, 2013)

With the Simics file system feature, users can move software packages easily from a local PC to Simics virtual platforms. The example (Figure 6) shows that moving software packages from a local drive and installing them onto a Simics virtual platform is as easy as operating on local directories.

Developers use exactly the same steps needed on real hardware to load and run the Intel QuickAssist test suite in Simics (Figure 7):

modprobe icp_qa_al
/lib/firmware/adf_ctl up
insmod build.ko

```
[root@metro simics_local]# ls
523127_DH895xCC.L.0.5.0_94.tar.gz   host
526848_QAT1.5.L.1.3.0_68.tar.gz     tt_batch.sh
526997_DH89xxCC.L.1.3.0_39.tar.gz
[root@metro simics_local]# cp *.gz /root/tian/
```

Figure 6: Using the simics file system to move production software onto the virtual platform
(Source: Intel Corporation, 2013)

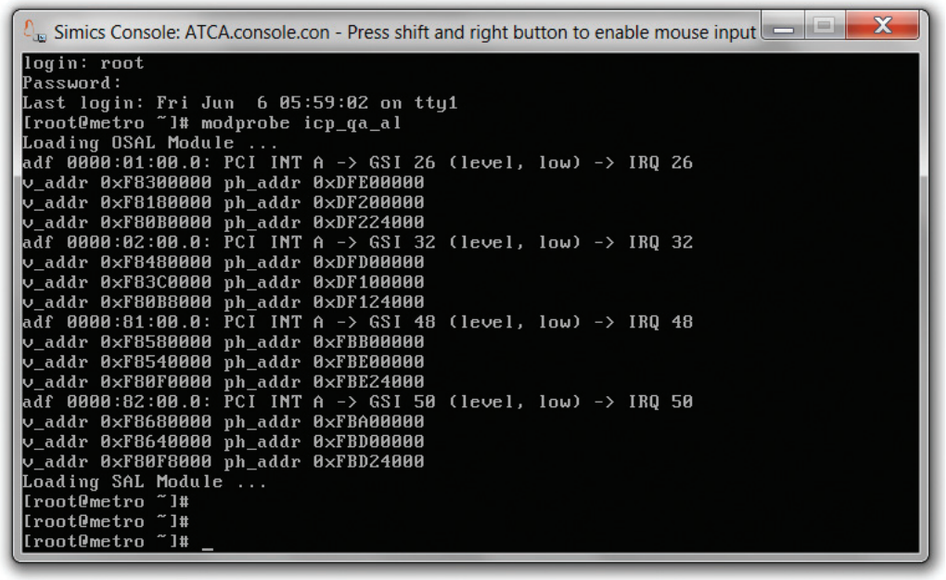

Figure 7: Simics Intel® QuickAssist loading software driver
(Source: Intel Corporation, 2013)

Within a few minutes, users can run an Intel QuickAssist test (Figure 8) and start get familiar with usage of the hardware functionalities.

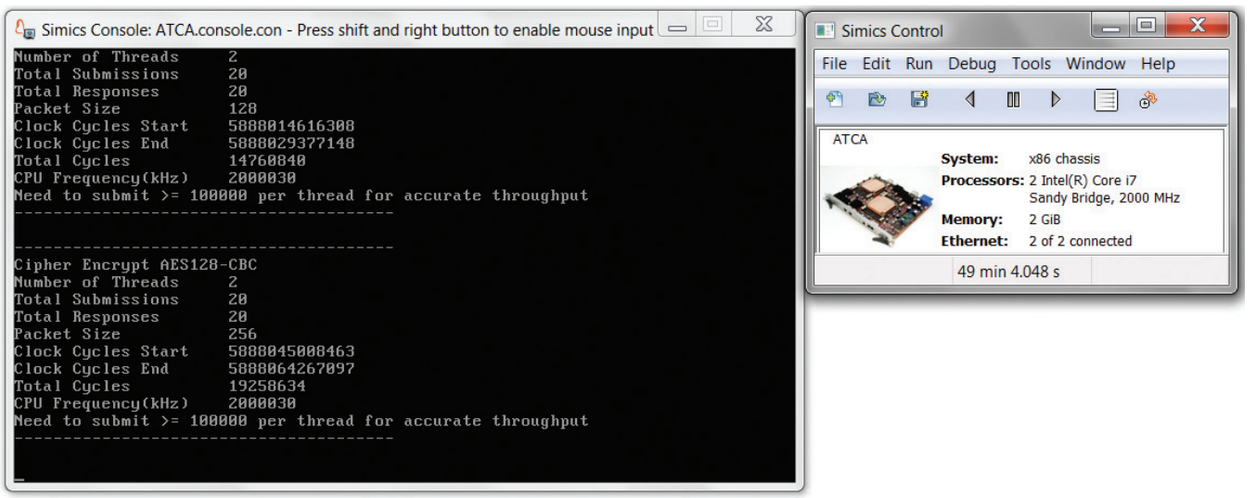

Figure 8: Simics Intel® QuickAssist running unmodified security test software
(Source: Intel Corporation, 2013)

"This usage case shows that customers can develop Simics models for their own ASICs or other hardware devices and use them for full system simulation."

In this usage case, Intel QuickAssist represents a combination of a purposely built hardware (Intel QuickAssist silicon) and software API (Intel QuickAssist API). With Simics, the module is able to respond to user and application requests as if there were real Intel QuickAssist silicon running underneath. This usage case shows that customers can develop Simics models for their own ASICs or other hardware devices and use them for full system simulation.

Ease of Intel® DPDK Adoption and Customization

Intel Data Plane Development Kit (Intel DPDK) is an optimized software stack provided by Intel that offers high performance packet processing. It is available in its example format or integrated and supported via several commercial and professionally supported solutions. Intel DPDK provides a set of libraries that can be used to optimize or improve performance over traditional and general-purpose Linux. For example, it provides a scheme to remove or reduce performance issues commonly associated with interrupt handler penalty, context switching, data copying and the Linux scheduler. These areas may be acceptable for general-purpose transactions but can be an issue for data I/O intensive workloads in the range of 10–40 gigabits.

However, just getting Intel DPDK software does not mean the job is done at the application side. The hard work tends to integrating the key optimization

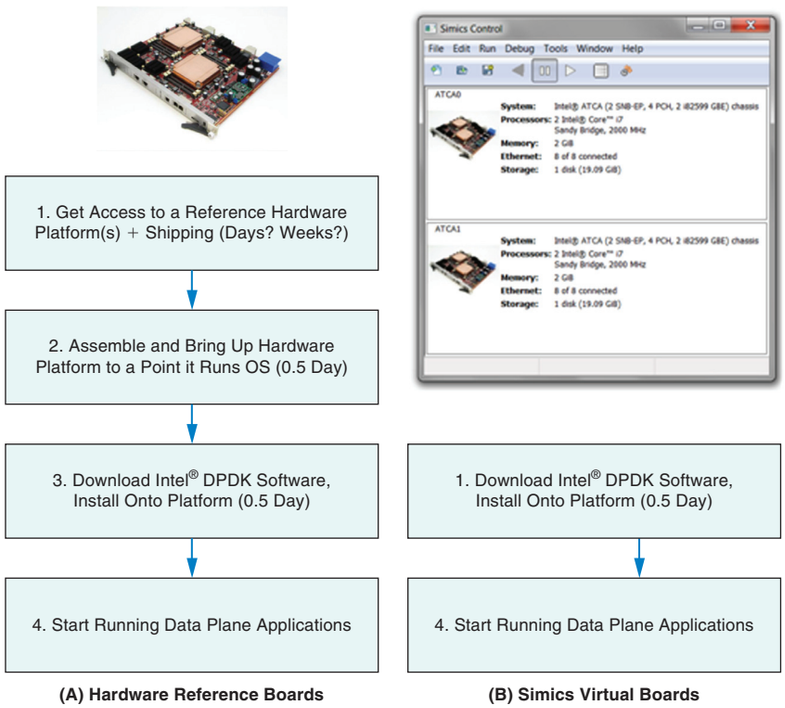

Figure 9: DPDK learning process on hardware and on virtual platform (Source: Intel Corporation, 2013)

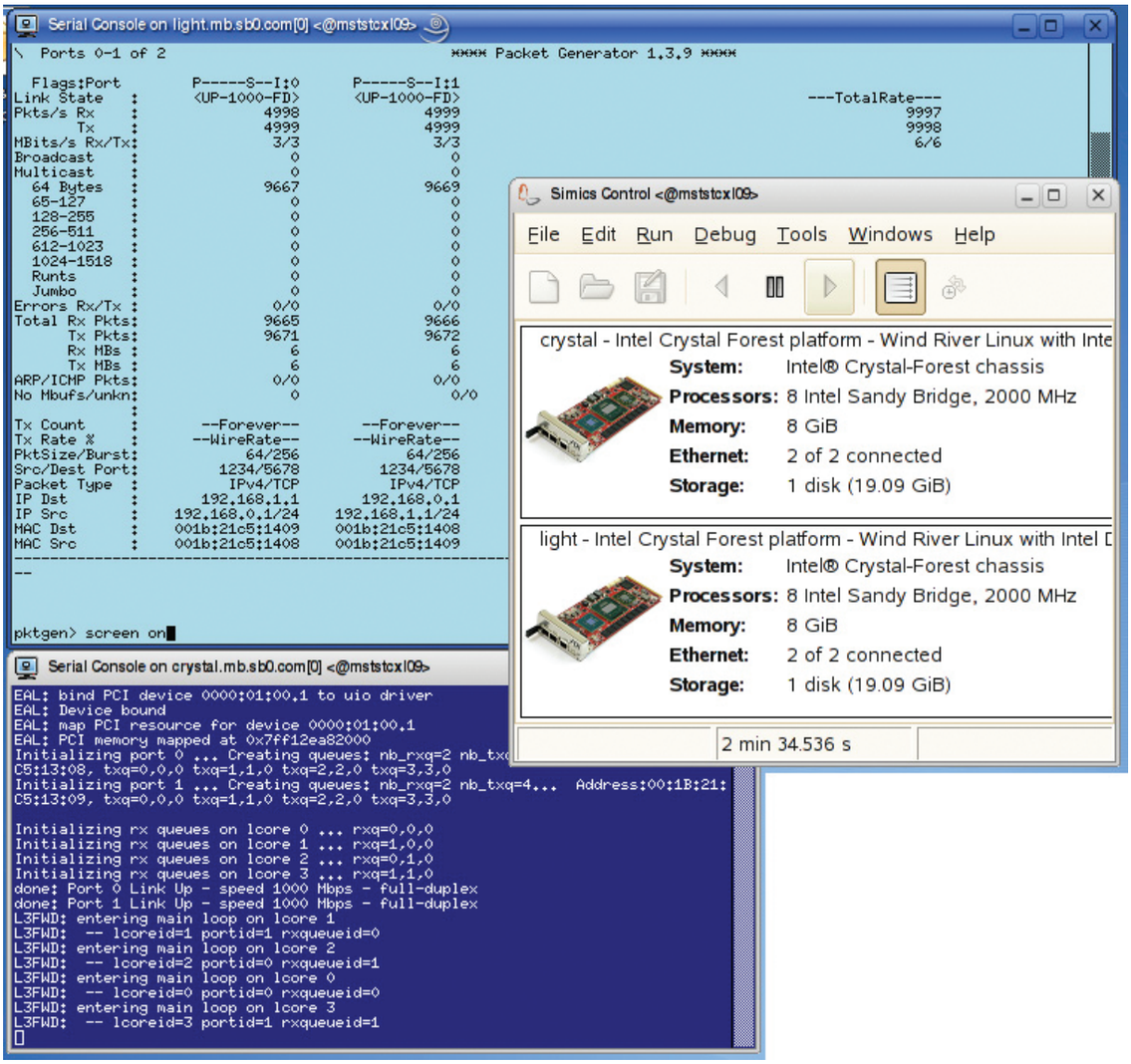

Figure 10: Simics crystal forest running Intel® DPDK
(Source: Intel Corporation, 2013)

principles learned from these software lessons into the customers' own software stacks and flows.

Simics can reduce the learning process for engineers on a new software stack such as Intel DPDK. As shown in Figure 9, the key bottleneck of the process tends to be the access to real reference platforms and the test harness. The process can take days or even weeks. Even when a board is available, the access can be limited. For example, for a team of engineers, sharing access to one or very few boards means only a small amount of time is available for an individual and this may translate to an impact to productivity. With Simics not only is the waiting process eliminated, all developers can have their own virtual boards. They can quickly launch Simics and begin learning the software, debug, build test cases, and explore new ways to

"Simics can reduce the learning process for engineers on a new software stack such as Intel DPDK."

do things. For example, with help from Simics, they can get an Intel DPDK demo traffic test going very quickly on their own PCs (Figure 10).

The actual customer usage of a reference software stack such as Intel DPDK requires a lot of customizations. The fact that Simics can run this type of software application proves that it becomes a viable solution in helping customers study the reference stacks and begin integrate their solutions on their virtual targets. This potentially brings significant saving in terms of schedule and time to market.

"With Simics, you can instantly turn your laptop or desktop PC to any platforms for which you have Simics models available."

Enable Access to Variety of Targets

One of the strengths of Simics is that it is a software package. With Simics, you can instantly turn your laptop or desktop PC to any platforms for which you have Simics models available. This is a huge benefit for engineers to have this kind of access and have the flexibility to select the target of interest. This means that engineers can conduct simulations independently of hardware in the lab. Shown in Figure 11 is an installation with three different configurations. AMC is a mobile version of the platform and ATCA is a server version of the platform. By installing both packages onto the PC, one can switch back and forth in between these platforms based on need, while in lab situations, it

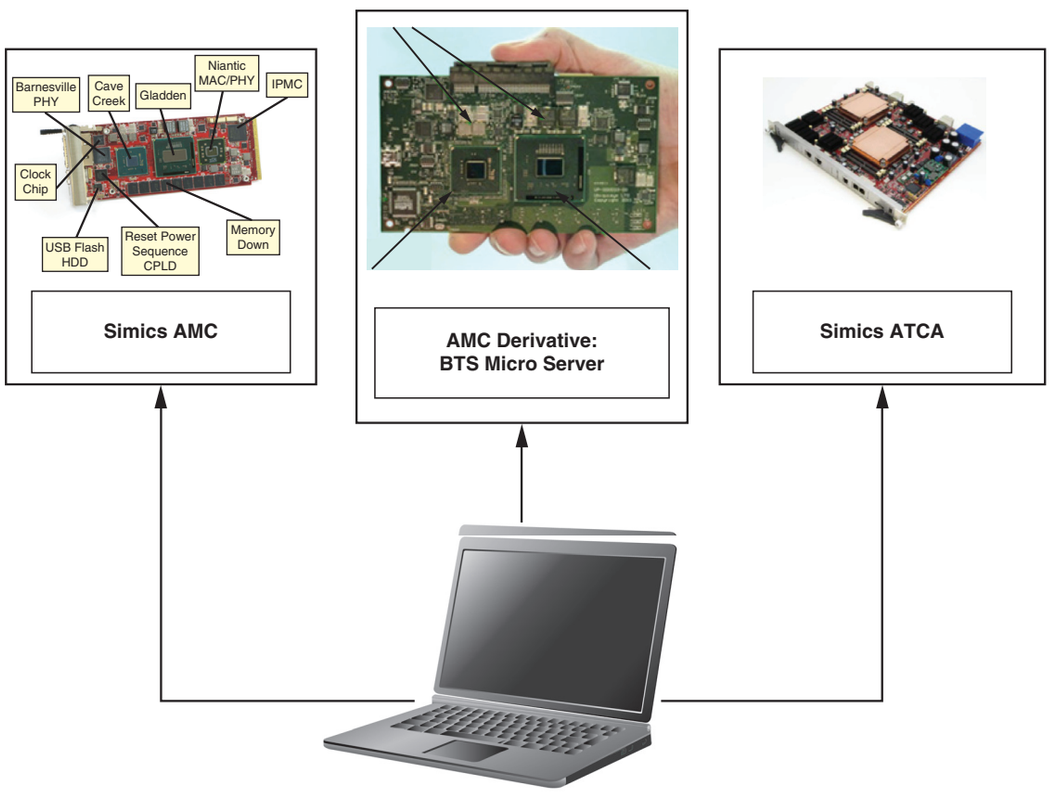

Figure 11: Simics crystal forest supports various form factors (Source: Intel Corporation, 2013)

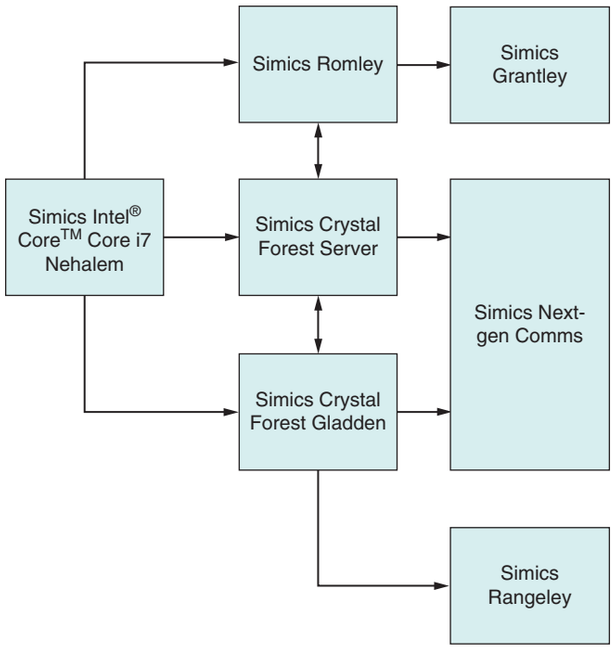

Figure 12: A variety of simics communication storage virtual platforms (Source: Intel Corporation, 2013)

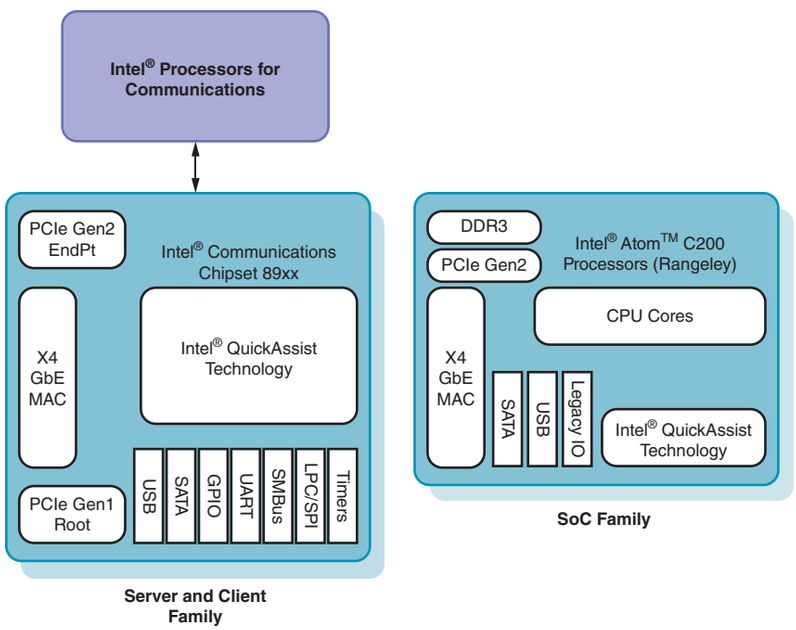

Figure 13: Reusing intel® QuickAssist module in rangeley virtual platform (Source: Intel Corporation, 2013)

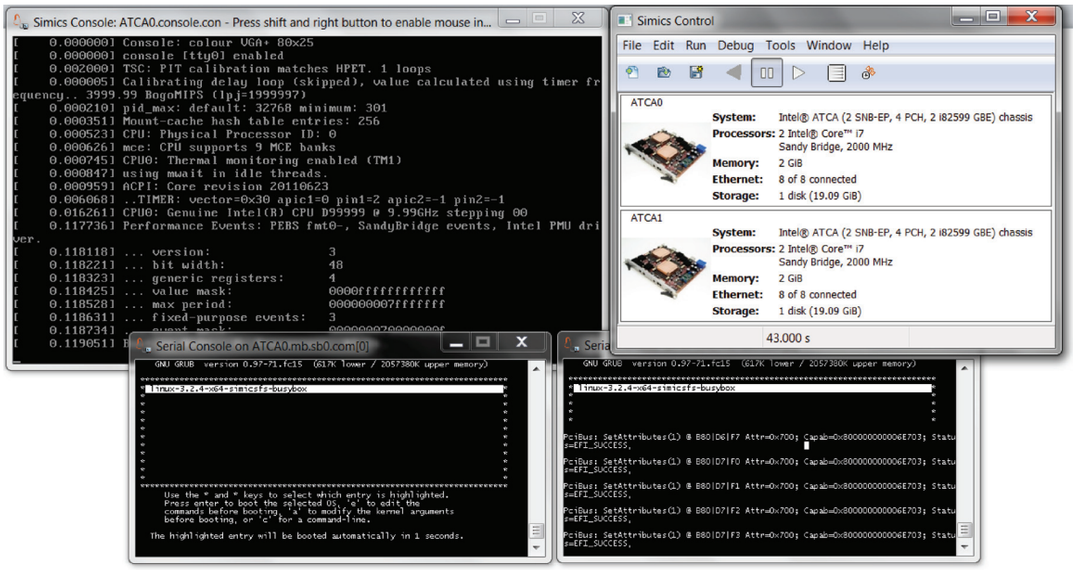

Figure 14: Dual ATCA simulation booting into busy box
(Source: Intel Corporation, 2013)

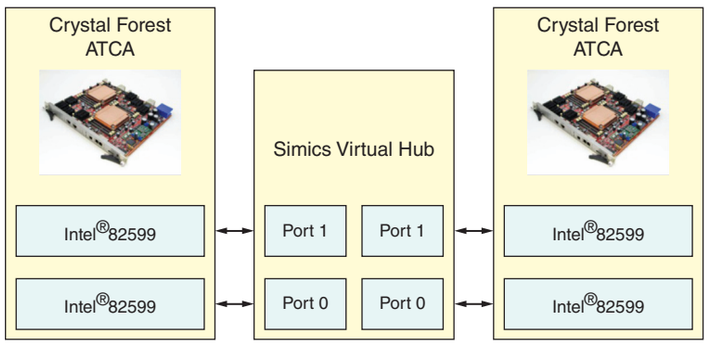

Figure 15: Dual ATCA networking setup
(Source: Intel Corporation, 2013)

is very rare for any given engineer to have that kind of unlimited access to a collection of systems anytime you need.

"We have developed Simics solutions for all our critical communication and storage platforms."

We have developed Simics solutions for all our critical communication and storage platforms. A few recent models are shown in Figure 12, ranging from server, mobile to System-On-Chip (SoC). SoC solutions such as the Intel® Atom™ C200 processor (codenamed Rangeley) are gaining a lot of traction in particular in the low-power and low-cost arena. These Intel C200 processor solutions share many common building blocks with their Intel Xeon counterparts.

As more and more products and platforms begin to have Simics models, another benefit starts to become significant: software reuse. On the Intel Atom C200 processor (codenamed Rangeley, see Figure 13), the Intel QuickAssist hardware module is part of the SoC, while in Intel Xeon families it is typically part of the south bridge. From a Simics modeling point of view, the existing solution from the Crystal Forest platform is drop-in compatible. This kind of software reuse further enhances the lead time software work now has ahead of hardware availability. As a result, both product solutions and Simics solutions benefit from this consistent approach across multiple generations and families. Developers working on future-generation solutions can get virtual hardware up and running a lot quicker because of effort invested in previous generations.

"…software reuse further enhances the lead time software work now has ahead of hardware availability."

Build Your Own Network Testing Infrastructure

Not many engineers have the luxury of controlling a large number of networking platforms. There are simply not enough platforms for every engineer. Sometimes the equipment is so expensive that it is not possible to enable everyone with enough access. Hardware schedule, shipping, and availability issues can also have a negative impact on productivity.

Simics solves the issue by giving access to anyone that needs it. Since it is software, it can be installed on a laptop or desktop; it can be carried around instead of locked in the labs. By loading different scripts and installing different packages, users have access to all kinds of platforms and can build and instantiate as many systems as needed for the purpose of the simulation.

Shown in the example (Figure 14) is a multi-board test that involves two Crystal Forest Server (ATCA) boards. Each board has identical settings (of course, one can easily create a network of different devices). The Dual-ATCA test later boots into Busy Box (Figure 14) and users can set up a network in between the two.

From this point on (Figure 15) we successfully created a small network of testers that involve two Crystal Forest ATCA boards. The boot time is good for developers. They can pause anytime they want to inspect the systems. From here we can add as many virtual boards as we want and run a network level of test, depending on the purpose.

Nicely Positioned for SDN and Intel® ONP Development

"The Intel ONP Server Reference Design is an SDN ready virtual switching building block."

The Intel ONP Server Reference Design is an SDN ready virtual switching building block. It utilizes virtualized network functions that define the hardware and software ingredients such as packet processing and management. At the heart of Intel ONP Server platform are the Intel Xeon processor and the Intel Communications Chipset 89xx series. Intel DPDK is also part of SDN and Intel ONP. The Intel ONP Switch Reference Design includes Wind River Open Network Software—an open and fully customizable networking software stack based on a Wind River developed abstraction layer and APIs.

Simics can simulate Crystal Forest hardware and sits nicely inside the overall Intel ONP architecture (Figure 16) and can interact with software stacks in

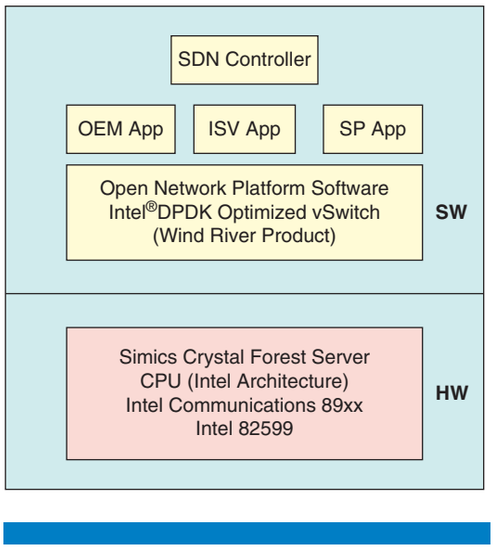

Figure 16: Simics crystal forest and intel® open network platform server architecture (Source: Intel Corporation, 2013)

upper layers (such as OEM App) as well as initiate networking with other Intel ONP or non–Intel ONP devices.

"…Simics can support SDN migration moving control and data plane operations to Intel ONP platforms."

The usage model here is that Simics can support SDN migration moving control and data plane operations to Intel ONP platforms. Simics can also help repartition hardware and software functionalities to support Network Functions Virtualization (NFV), where network functions (L3 forwarding, management plane, security) are shifting from hardware to software applications running in a virtualized environment. If the customers already have Simics models for their current solutions, they can begin that migration process to Intel ONP immediately without waiting for their own hardware availability. Even if they do not yet have their Simics device models, they can begin using Simics Crystal Forest building blocks and studying Intel ONP and SDN migration paths.

Conclusions

In this article, we walked through several examples of Simics post-silicon usage and demonstrated its positive impact to the product lifecycle. During our work on Intel® Next Generation Communications Chipset (code named Crystal Forest),we successfully used Simics to help debug system issues after hardware became available and proved various usage models for the post-silicon phase. There are several aspects that stand out from our experience:

- Simics enables us to apply more resource debug issues concurrently and it provides a unique approach in recreating the issue in simulation. The debug time is reduced because it is more efficient in software simulator as one can pause and even reverse-debug the problem. It is changing the way we approach BIOS and firmware development.

- Simics allows more developers to obtain access to hardware and software features such as the Intel QuickAssist security solution and Intel DPDK. This removes dependency on hardware so developers continue the learning or development process on simulation. This speeds up the learning and development process significantly.
- Simics brings a great deal of reuse between product families. Using the Intel® Atom™ C200 processor (previously codenamed Rangeley) as an example, we can see that the simulation model can be easily reused between families, which results in time saved both in Simics model development time and also overall product development time.
- Simics has tremendous support for Intel® Next Generation Communication platforms (such as Crystal Forest and future versions) as well as Intel® Open Network Platform and the software-defined network. These platforms are important vehicles for customers to create solutions for NFV and SDNs. Simics is positioned to be a key contributor during this transformation. As problems and systems get more complicated, Simics tends to get even more powerful with its total control on the system and excellent simulation speed.

We are in the middle of a paradigm shift where a more modularized, multilayered, flexible, demand-driven network solution is clearly where the industry is heading. The trend is driven by demand, cost reduction, time to market, and new usage and service models. Existing vendors are being pushed to come up with solutions faster and more scalable to meet explosive growth. The biggest challenge may be the effort required to move legacy software and hardware solutions and repartition workloads to run on new frameworks. Each vendor tends to have their proprietary solutions, which makes the transformation process a lot harder than it should be.

Simics is exactly the type of technology needed to go after this major challenge. Its values in post-silicon usage are well beyond a single isolated device or system. It is perfectly suited for heterogeneous and networked systems and provides a simulation speed that is highly desirable for firmware and software engineers. It gives users a golden key to prototype and pilot software migration paths even before hardware is built. It can give users a tremendous edge when developing virtualized solutions and/or environment with security constraints. Today, Simics already has support for key ingredients to be a significant contributor during the migration process to NFV/SDV and Intel Open Network Platforms. Simics may just be the difference maker for solution vendors in terms of time to market in the era of network transformation.

"Today, Simics already has support for key ingredients to be a significant contributor during the migration process to NFV/SDV and Intel Open Network Platforms."

Complete References

[1] Intel® Communications Chipset 89xx Series Datasheet, v2, June 2013

[2] Intel® Data Plane Development Kit Getting Started Guide, Reference number 326002-003, Intel Corporation, August 2013

[3] Implementing SDN and NFV with Intel® Architecture, Intel® Corporation, April 2013

[4] Intel® Processors for Communications Datasheet, Vol. 1 Document Number: 327405-001, Intel Corporation, June 2013

Author Biography

Tian Tian is a seasoned embedded system designer with 12+ years of product design, application, and market development experience. He developed a key voice component for Intel's IXP425 network processor family and architected the Access Software OS library for Xscale-based systems. In recent years he has been managing various x86 product families in the embedded segment including Intel® Centrino®, Intel Core Duo, Intel Xeon with a focus on the communication industry. Tian began use Simics as part of the design process in 2010, was involved with Simics Crystal Forest external release in 2011, and is leading the technical marketing effort for Simics usage of next-generation platforms. Tian holds a Master of Electrical Engineering from Arizona State University and has published many technical white papers and articles.

LANDSLIDE: A SIMICS* EXTENSION FOR DYNAMIC TESTING OF KERNEL CONCURRENCY ERRORS

Contributors

Ben Blum
Department of Computer Science,
Carnegie Mellon University

David A. Eckhardt
Department of Computer Science,
Carnegie Mellon University

Garth Gibson
Department of Computer Science,
Carnegie Mellon University

Landslide is a Simics module designed for finding concurrency bugs in operating system kernels, with a focus on Pebbles. Pebbles is a UNIX-like kernel specification used in course 15-410, the undergraduate operating systems class at Carnegie Mellon University, in which students implement such a kernel in six weeks from the ground up. Landslide's mechanism, called systematic testing, involves deterministically executing every possible interleaving of thread transitions in a given test case and identifying which ones expose bugs. In this article we explain the testing environment (the course, 15-410, and the kernel, Pebbles) and the testing technique; describe how Landslide takes advantage of certain features that Simics provides that other testing environments (such as virtualization) do not; outline Landslide's design, implementation, and user interface; present some results from a preliminary evaluation of Landslide, and discuss potential directions for future work.

Introduction

Race conditions are notoriously difficult to debug. Because of their nondeterministic nature, they frequently do not manifest at all during testing, and when they do manifest, it can be difficult to reproduce them reliably enough to collect enough information to help debugging.

Many techniques exist for dynamic testing of concurrent systems for race conditions. Systematic exploration, the strategy we focus on in this work, involves making educated guesses as to what points during execution a preemption would be most likely to expose a bug, enumerating the different possibilities for interleaving threads around these points, and forcing the system to execute all such interleavings to check if any of them results in incorrect behavior.[1] Systematic exploration provides a better alternative to conventional long-running stress tests, because it is less likely to overlook buggy execution patterns, and it enables a testing framework to report more thorough debugging information. Compared to other dynamic analyses, such as data race detection[2], systematic exploration is able to find a wider range of types of concurrency errors because of its ability to manipulate the execution of the system under test.

"Systematic exploration is able to find a wider range of types of concurrency errors"

In this article, we present Landslide, a Simics module that provides a framework for performing systematic testing on kernel-level code.[3] Landslide is designed with a focus on the testing environment used by students in course 15-410, the undergraduate operating systems class at Carnegie Mellon University (CMU). In 15-410, students implement a fully preemptible, UNIX-like kernel from the ground up over the course of a six-week project.[4] They

use the Simics simulator as their primary testing and development platform, although they must rely on conventional stress-testing techniques to find and track down concurrency bugs in their code. Landslide is an effort to improve this situation by making the more sophisticated technique of systematic testing accessible to developers of kernel code.

This article is structured as follows. In the section "15-410 and Pebbles," we discuss the course design, projects, and learning objectives for 15-410, with a detailed overview of the requirements of the kernel project. In the section "Systematic Testing," we introduce the technique of systematic testing, explaining its requirements, advantages, and challenges. In the section "Design and Implementation," we discuss the design of Landslide's architecture, describing the overall sequence of events involved in a systematic testing run, and the various components of Landslide and how they fit together. In the section "Use of Simics Features," we focus specifically on how Landslide and Simics fit together, highlighting the unique features that Simics offers that make Landslide's job possible. In the "User Interface" section, we present Landslide's user interface, describing the instrumentation process users must complete in order to use Landslide, and the interface Landslide offers for fine-tuning the search parameters and reasoning about uncovered bugs. In "Results" we discuss a user study we conducted with volunteer students from 15-410, in which Landslide was able to help the students uncover and fix previously-unknown race conditions in their own kernels, and finally, in "Future Work," we conclude with a discussion of the most promising future work directions for this research.

15-410 and Pebbles

15-410, the Operating Systems Design and Implementation course at CMU, is a semester-long project course comprising five projects. The projects are a stack tracer, kernel device drivers (for timer, keyboard, and console), a 1:1 user-space threading library to run on a Pebbles kernel, the Pebbles kernel itself, and a small extension to the Pebbles kernel. Simics is used as the main development and debugging environment for the latter four projects.

The course has many learning objectives, ranging from acquiring detailed factual knowledge about hardware features through practicing advanced cognitive processes such as open-ended design. Students study high-level concepts such as protection (least privilege, access control lists vs. capabilities), file-system internals, and log-based storage. We place emphasis on acquiring information from primary sources, including both manufacturer-provided hardware documentation and a non-textbook technical-literature reading assignment. Students begin with a blank slate rather than a kernel-source template or an existing operating system, so they must synthesize design requirements from multiple sources and must choose their own module boundaries and inter-module conventions. Due to the foundational nature of kernel code, the assignment design and grading encourage students to think about corner cases, including resource exhaustion, instead of being satisfied by "the right basic idea" implementations that handle only auspicious situations. Finally, most relevant to

"The assignment design and grading encourage students to think about corner cases, instead of being satisfied by 'the right basic idea' implementations."

this work, students gain substantial experience in analyzing and writing lock-based multi-threaded code and thread-synchronization objects. They practice detecting and documenting deadlock and race conditions, including both thread/thread concurrency and thread/interrupt concurrency.

Project Overview

In the course of a semester, students work on five programming assignments; the first two are individual, and the remaining three, including the kernel project itself, are the products of two-person teams. Here we are primarily concerned with the kernel project, though we will also briefly describe the others.

Introductory Projects

The first project is a stack crawler: when invoked by a client program, it displays the program's stack symbolically, rendering saved program-counter values as function names and printing function parameters in accordance with their types. This project enables students to review key process-model and language-runtime concepts from the prerequisite course[5]; it introduces students to our expectations about design, analysis, and making choices; finally, because C pointers are unsafe, it requires students to consider robustness.

The second project is a simple game, such as Hangman, which runs without an underlying operating system. The project requires students to implement a device driver library consisting of console output, keyboard input, and a hardware timer handler. This project and the remaining ones are written in C with some x86-32 assembly code, which is then compiled and linked into an ELF executable, stored into a 1.44-megabyte 3.5-inch floppy-disk image, and booted via GRUB. If the image is copied to a real floppy or embedded into an "El Torito" bootable compact disc image, it can be booted on standard PC hardware; however, students most often use Simics, to take advantage of its debugging facilities.

The third project is a 1:1 thread library for user-space programs, essentially a stripped-down version of POSIX Pthreads. Students begin by designing mutexes using any x86-32 atomic instructions they choose. They then write other thread-synchronization primitives (condition variables, semaphores, and reader/writer locks), infrastructure components (stack allocation/recycling and a thread registry), and low-level code to launch and shut down threads. Student library code is linked with small test programs provided by the course staff. The test programs run on a reference kernel written by the course staff and provided in binary form, the behavior of which is specified in a twelve-page document. In addition to providing a reliable execution substrate, the reference kernel schedules the execution of user-space threads created by student code according to a variety of interleaving policies.

"Two-student teams produce a kernel which implements the same specification as the reference kernel they previously relied on."

The Pebbles Kernel Project

For the fourth project, two-student teams produce a kernel which implements the same specification as the reference kernel they previously relied on. They design and implement some approach to synchronizing and blocking threads while they are in kernel space, a simple round-robin scheduler, basic virtual memory, a program loader, code to handle various x86 exceptions, and code

for setting up and tearing down threads and processes (they reuse their game-project device drivers). We briefly describe each of the 25 system calls in the Pebbles specification in Table 1.

Name	System Call Description
	Lifecycle Management
`fork`	Duplicates the invoking task, including all memory regions.
`thread_fork`	Creates a new thread in the current task.
`exec`	Replaces the program currently running in the invoking task with a new one.
`set_status`	Records the exit status of the current task.
`vanish`	Terminates execution of the calling thread.
`wait`	Blocks execution until another task terminates, and collects its exit status.
`task_vanish*`	Causes all threads of a task to vanish.
	Thread management
`gettid`	Returns the ID of the invoking thread.
`yield`	Defers execution to a specified thread.
`deschedule`	Blocks execution of the invoking thread.
`make_runnable`	Wakes up another descheduled thread.
`get_ticks`	Gets the number of timer ticks since bootup.
`sleep`	Blocks a thread for a given number of ticks.
`swexn`	Registers a user-space function as a software execption handler.
	Memory Management
`new_pages`	Allocates a specified region of memory.
`remove_pages`	Deallocates same.
	Console I/O
`getchar*`	Reads one character from keyboard input.
`readline`	Reads the next line from keyboard input.
`print`	Prints a given memory buffer to the console.
`set_term_color`	Sets the color for future console output.
`set_cursor_pos`	Sets the console cursor location.
`get_cursor_pos`	Retrieves the console cursor location
	Miscellaneous
`readfile`	Loads a given buffer with the names of files stored in the RAM disk "file system."
`halt`	Ceases execution of the operating system.
`misbehave*`	Selects among several thread-scheduling policies.

Table 1: The 25 system calls described in the Pebbles specification. Students are not required to implement the three system calls marked with an asterisk (*).
(Source: Pebbles kernel specification, 2013.[4])

"For most students in the class, this is the largest and most complicated software artifact they have produced."

For most students in the class, this is the largest and most complicated software artifact they have produced. Because the test suite and the grading criteria emphasize robustness and preemptibility of kernel code, there are many cross-cutting concerns. As students are responsible for ensuring the runtime invariants underlying all compiler-generated code in the system (kernel and user-space), they gain experience with debugging at both the algorithm level and the register/bit-field level.

Widely regarded as the most difficult concurrency problem in the project is that of coordinating a parent and a child task that "simultaneously" exit: when a task completes, live children and exited zombies must be handed off to the task's parent or to the system's "init" process, at a time when the task's parent may itself be exiting; meanwhile, threads in tasks that receive new children may need to be awakened from the wait() system call. Due to design constraints imposed by other parts of the kernel specification, solutions that are not carefully designed are prone to data races or deadlocks.

Students who complete the kernel project on time then work on a kernel-extension project, with varying content depending on the semester. Past projects have included writing a sound card driver, a file system, hibernation (suspend to disk), kernel profiling, and an in-kernel debugger. Two recent, more aggressive, projects have been adding paravirtualization so that their kernels can host guest kernels and adding multiprocessor support to their single-processor kernels.

"Unlike some emulators, which focus on fast execution of correct code, Simics provides very faithful support not only for correct code but also for kernels that accidentally abuse hardware."

Use of Simics

Simics serves as the main execution and debugging platform in 15-410. Unlike some emulators, which focus on fast execution of correct code, Simics provides very faithful bit-level support not only for code that behaves correctly but also for kernels that accidentally "abuse" hardware. Unlike hardware virtualization environments, Simics contains substantial debugger support: single-stepping, printing of source-level symbolic expressions, stack tracing, display of TLB entries, and even summaries of x86 hardware-defined descriptor tables. All of these features make Simics a helpful platform for students to test their code. A major advantage of using Simics over the QEMU emulator in particular is that QEMU issues timer interrupts only at basic-block boundaries, which would dramatically undermine our goal of teaching students that threads can interleave with each other at any time.[6]

Systematic Testing

The underlying idea of systematic testing is to view the set of all possible execution sequences, which can change due to concurrency nondeterminism, as an *execution tree*. The root of this tree denotes the start of the test case, each branch represents one execution sequence, and nodes in the tree are *decision points*: time points during the execution where Landslide should attempt to force a different thread to run, thereby making progress through the state space.

Example

Consider the example code in Code 1, which demonstrates how the *thread_fork()* system call might be implemented. If a timer interrupt occurs at line 4, the child thread can run, exit, and free its state, causing the access on line 5 to be a use-after-free. Here, the necessary decision point for finding the bug is at line 4. Landslide will know that there should be a decision point here because it automatically interprets new threads becoming runnable as important concurrency events. Other decision points may also exist, for example, during the construction of the new *thread_t* struct, or during the new thread's execution. Together, the set of decision points defines an *execution tree* that contains this bug, depicted in Figure 1.

```
1  int thread_fork() {
2          thread_t *child = construct_new_thread();
3          add_to_runqueue(child);
4          // note: at this point child may run and exit
5          return child->tid;
6  }
```

Code 1. Example implementation of the thread_fork() system call. This example contains a race condition, described in the comment on line 4. Source: Landslide: Systematic dynamic race detection in kernel space, 2011.[3]

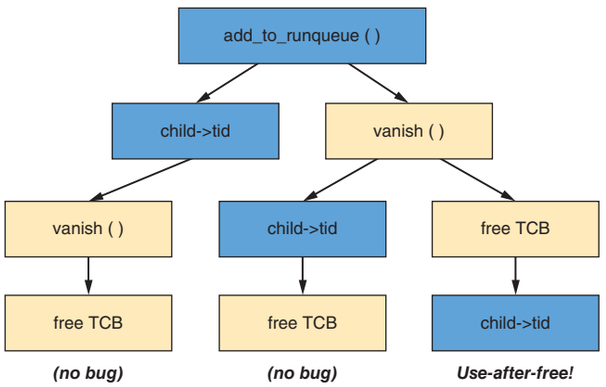

Figure 1: The set of possible execution sequences can be viewed as a tree of thread interleavings, in which a concurrency bug is only exposed in some branches. This particular tree is derived from the example code in Code 1. (Source: Landslide: Systematic dynamic race detection in kernel space, 2011.[3])

Challenges

In any systematic testing tool, there is an inherent tradeoff when defining the set of decision points: searching with few decision points results in coarser-grained interleavings, faster test completion, but less likelihood of finding unexpected bugs; whereas searching with more decision points results in the opposite. Accordingly, Landslide provides an interface for adjusting the set of

"There is an inherent tradeoff when defining the set of decision points."

"Combining systematic testing with a kernel-space execution environment presents some additional challenges."

considered decision points, which we discuss further in the section, "Use of Simics Features."

Combining the technique of systematic testing with a kernel-space execution environment presents some additional challenges. First, a testing tool must control all sources of nondeterministic input to the system, and account for all the scheduling options by each such source of input at each decision point. In the Pebbles environment, the only sources of nondeterminism are timer interrupts and keyboard input. With Landslide, we focus exclusively on timer interrupts, as they can be used to directly control the kernel's context switching.

A second challenge of systematic testing in kernel-space is that of the scheduler. Because kernels contain their own concurrency implementation, it can be difficult to find bugs in the scheduler itself while also being able to use assumptions about the scheduler's behavior to optimize our search for bugs in other parts of the kernel.

A third challenge is the issue of multiprocessor kernels: when multiple CPUs can be running different threads simultaneously, additional nondeterminism can arise from the order in which their instructions are executed. Some race conditions may even require multiple active CPUs in order to manifest. However, as 15-410 does not require student kernels to be capable of SMP execution, Landslide assumes kernels will only ever use one processor. Lifting this limitation is left to future research.

Design and Implementation

This section describes the important components of Landslide's architecture. Conceptually, Landslide is designed as follows. Students annotate their code so that Landslide knows which kernel thread is currently running. After one kernel thread has run for some time, Landslide triggers artificial clock interrupts to force the scheduler to run a different thread. When a test program finishes execution according to one pattern of thread switches, Landslide rewinds the kernel's state and resumes the test according to a different thread interleaving. After each instruction, Landslide applies several bug-detection predicates to the kernel's state to detect illegal heap accesses, deadlock, infinite loops, and panics. In theory, by forcing a thread switch after *every* non-scheduler instruction, Landslide could apply its bug-detection predicates to every reachable execution state. Because this would require a prohibitively large amount of time to complete, in practice Landslide uses a variety of techniques to thread-switch less often and to avoid repeating bug-equivalent execution paths.

In order to achieve this exploration of the state space, Landslide comprises several components, which are depicted visually in Figure 2 and described in the following sections.

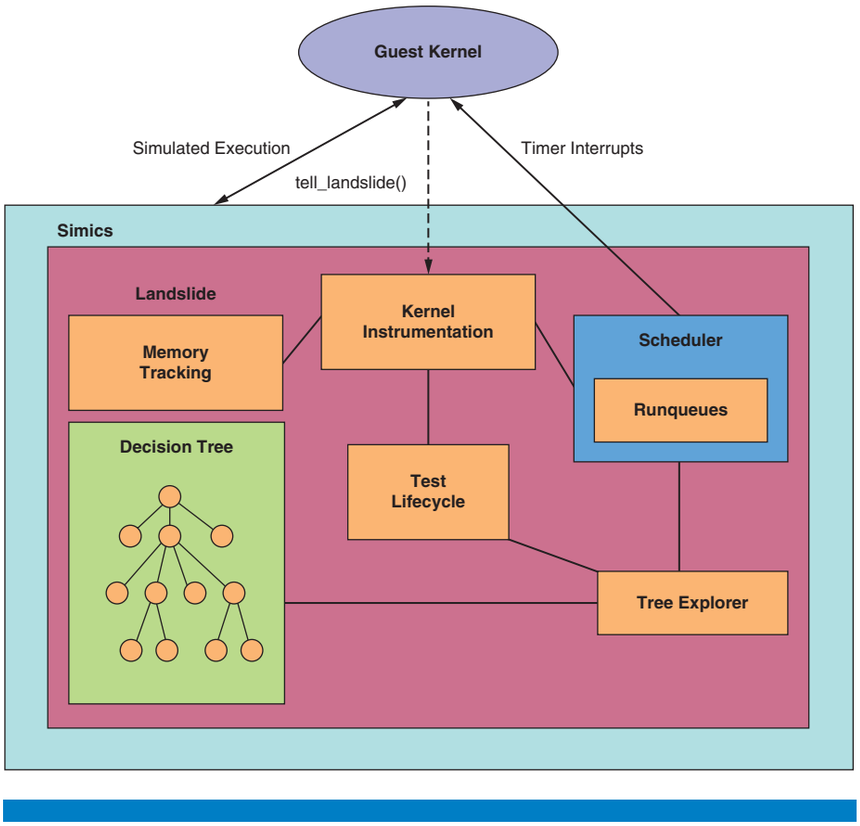

Figure 2: Visual representation of landslide's architecture and its interface with the kernel under test.
(Source: Landslide: Systematic dynamic race detection in kernel space, 2011.[3])

Thread Scheduler

The Landslide scheduler is responsible for keeping track of which threads exist in the guest kernel: which are runnable at any given time, and when they are created and destroyed. It maintains a "mirror image" of the guest kernel's scheduler state in the form of three queues, a pointer to the currently-running thread, and a pointer to the previously-running thread. The queues are the *runqueue*, containing the runnable threads, the *sleep queue*, containing threads which become runnable after a certain number of timer ticks, and the *deschedule queue*, which might not correspond to a data structure in the guest kernel, but contains all other threads that exist on the system that are not runnable for whatever reason.

Though we define timer interrupts as the only source of nondeterminism in our environment, it is more useful to view the concurrent behavior with a higher-level abstraction, in terms of the set of runnable threads and the ability to preempt the currently running thread with any different runnable one. Hence, the scheduler also contains the mechanism for translating the tree explorer's high-level decisions about which thread should run next into a lower-level sequence of timer interrupts (which trigger context switches). Note that multiple interrupts

may sometimes be necessary to force the desired thread to run; for example, if the kernel scheduler uses a round-robin policy and has a runqueue of thread IDs 1, 2, and 3 (with thread ID 1 currently running), if the Landslide scheduler desires to run thread 3, it will take 2 interrupts before thread 3 begins running.

Memory Access Tracking

Landslide maintains a mirror image of the guest kernel's dynamic allocation heap, so it can know at any point which memory ranges are allocated and which ranges used to be allocated but now are freed. This set is updated each time the guest kernel calls *malloc()* or *free()*. This heap tracking provides the ability to check for dynamic allocation errors (such as use-after-free and double-free bugs), in a similar fashion to the *Valgrind* debugging tool.

Landslide also maintains a set of shared memory accesses made since the last decision point, for use with the Partial Order Reduction state space technique (which we describe in the next section). This set of accesses allows Landslide to determine when certain actions of different threads may conflict with, or are independent from, each other. Landslide ignores shared memory accesses from the kernel's dynamic allocator itself, and it also ignores shared memory accesses from the components of the kernel's scheduler that run every transition.

Execution Tree Explorer

The execution tree explorer maintains a representation of the current branch of the decision tree. It is responsible for checkpointing the state of both Landslide and the guest kernel at each decision point, deciding at the end of the test which branch of the tree to execute next (that is, selecting which decision point should have been decided differently), and backtracking to appropriate points in the test's execution.

The explorer also identifies points during execution that should count as decision points. The selection is mainly controlled by the user, during the annotation and configuration process. However, the explorer also automatically identifies *voluntary reschedules*—points at which the kernel explicitly invokes a context switch of its own accord (for example, in *yield()*)—which comprise the "minimal necessary set" of decision points.

"DPOR analyzes the memory accesses to identify a set of candidate branches to explore next."

During the backtracking stage, the explorer applies a state-space reduction technique called *Dynamic Partial Order Reduction* (DPOR). Briefly, DPOR analyzes the memory accesses in a just-finished execution to identify a set of candidate branches to explore next. These branches represent reorderings of state transitions that conflicted with each other, with reorderings of independent transitions pruned out. For example, Figure 3 depicts a subset of a possible execution tree in which the highlighted transitions of threads 1 and 2 are independent from each other (that is, if they were reordered, the resulting kernel state would be identical.)

Bug Detection Techniques

During the test case's execution along each thread interleaving, Landslide applies several bug-detection predicates to the kernel's state, some accurate and some heuristic-based.

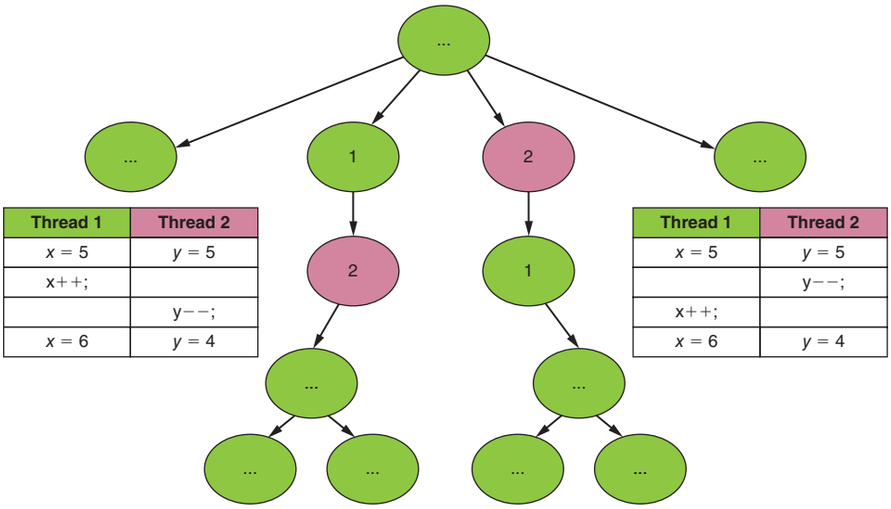

Figure 3: An example part of an execution tree that could be pruned using DPOR. The highlighted transitions of threads 1 and 2 are independent, meaning that to achieve full coverage, Landslide needs to explore only one of the two subtrees.
(Source: Landslide: Systematic dynamic race detection in kernel space, 2011.[3])

Landslide's "definite" bug-detection techniques include identifying kernel panics, use-after-free bugs (making use of the heap access tracking), and deadlocks (making use of mutex and scheduler instrumentation).

Additionally, Landslide can heuristically detect infinite loops by comparing the current execution of the test case against previous executions under different thread interleavings. If the current execution has lasted a certain proportion longer than the average of all previous executions, as visualized in Figure 4, Landslide assumes the deviation represents a nondeterministic infinite loop.

"Landslide can heuristically detect infinite loops by comparing the current execution of the test case against previous executions."

Use of Simics Features

This section discusses how Landslide and Simics fit together, and highlights some Simics features that Landslide makes heavy use of to enable systematic testing.

Landslide is implemented as a "trace" module, which means that Simics calls into it once per instruction and once per memory access, supplying information about the instruction or access about to be performed. Landslide uses this information to update its internal state machine to track the kernel's progress, by reading the values at memory locations, comparing the current instruction against certain known execution points in the kernel, and so on.

Landslide's control over the system consists of two parts. Together, these parts enable it to steer the kernel through the different branches of the execution tree, testing for bugs in each branch until the tree is exhausted.

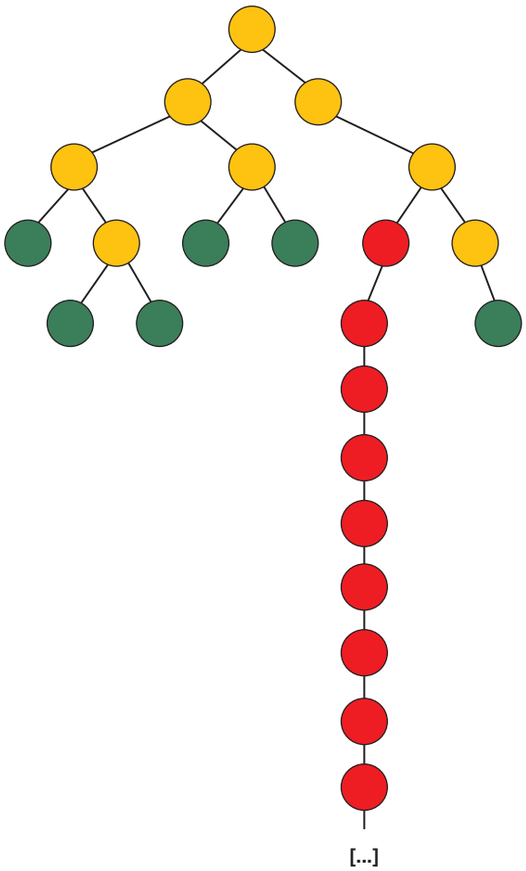

Figure 4: An example decision tree containing a nondeterministic infinite loop. If Landslide explores the highlighted branch after testing sufficiently many of the terminating branches, it assumes the kernel is stuck in an infinite loop and will report a bug. (Source: Landslide: Systematic dynamic race detection in kernel space, 2011.[3])

"If that thread is not the desired one, Landslide repeats the process, injecting more timer interrupts until the desired thread begins running."

The first part is causing a timer interrupt to occur at a given point during the kernel's execution. Landslide achieves this by manipulating the CPU's pending interrupt vector. When Landslide wishes to cause a particular thread to preempt another thread at a given decision point, it injects a timer interrupt before the pending instruction. In response, the kernel triggers a context-switch to the next thread on its scheduler run-queue. If that thread is not the desired one, Landslide repeats the process, injecting more timer interrupts until the desired thread begins running.

The second part of Landslide's control is backtracking. At the end of each branch of the decision tree, if Landslide wishes to explore a different interleaving

at a particular decision point, it must reset the system state to the past state at that point. Fortunately, Simics provides a facility for reverse-execution in the form of the *set-bookmark BOOKMARK-NAME* and *skip-to BOOKMARK-NAME* commands. At each decision point during execution, Landslide uses *set-bookmark* to ask Simics to set a bookmark. Then, when the current execution of the test case completes, Landslide uses *skip-to* to reverse-execute to the bookmark associated with the desired decision point, at which point exploration resumes. Because Landslide places itself outside the scope of Simics' reverse execution system, although the entire simulated machine state is reset to the earlier point, Landslide's memory of the entire state space tree is persistent.

User Interface

Instrumenting and testing a kernel with Landslide involves three stages of effort. These are required annotations, configuring decision points for a more efficient search, and interpreting the resulting traces Landslide emits when it finds a bug. This section gives a brief overview of each.

Required Annotations

Users annotate their kernels to inform Landslide of certain important concurrency events during execution. We provide a set of annotation functions, named with the prefix *tell_landslide*, for this purpose. The annotations denote when a thread runs *fork()*, *sleep()*, or *vanish()*, when a thread is added to or removed from the run-queue, and when a thread becomes blocked on a mutex. The annotation is placed just before the actual action being annotated. Code 2 shows an annotated sample of the code from the example in the "Systematic Testing" section.

```
1  void add_to_runqueue(thread_t *child) {
2          tell_landslide_thread_runnable(child->tid);
3          // ... more implementation follows ...
4  }
5  int thread_fork() {
6          thread_t *child = construct_new_thread();
7          tell_landslide_forking(child->tid);
8          add_to_runqueue(child);
9          return child->tid;
10 }
```

Code 2. The same example *thread_fork()* implementation, now with annotations for use with Landslide.
Source: Landslide: Systematic dynamic race detection in kernel space.[3]

There is also a configuration file, *config.landslide*, in which the student must specify constant information such as the function names of the timer handler and context switcher, which threads exist when the kernel boots, and which user-space test program Landslide should invoke.

Finally, there are two short (nominally two-line) functions used within Landslide itself that the user must implement. These are predicates on the kernel's scheduler

state and express potentially nontrivial conditions: whether the current thread is runnable but not on the run-queue, and whether preemption is disabled while interrupts are on. This logic executes within Landslide, inside of Simics, rather than as part of the simulated kernel's execution.

Configuring Decision Points

If Landslide uses only decision points that it automatically identifies on voluntary reschedules, the resulting interleavings will be coarse-grained and likely to overlook bugs. We provide an extra annotation for students to add more decision points for a finer-grained search, called *tell_landslide_decide()*. We recommend inserting it into concurrency primitives, such as at the start of *mutex_lock()* and at the end of *mutex_unlock()*.

However, this strategy may cause Landslide to identify decision points in unrelated parts of the kernel, such as when accessing mutexes in unrelated and/or already-trusted system calls. We provide interface options in *config.landslide* for the student to view currently identified decision points and to selectively eliminate them. For example, if a student were testing thread death and reaping, they might want decision points to appear in *wait()* and *vanish()* but not if unrelated virtual memory operations are also in progress. Accordingly, they could write *within_function wait vanish* and *without_function destroy_address_space*. The *within_function* directive requires that at least one of the specified functions shall be on the call stack when decision points are identified, and *without_function* requires the opposite.

Decision Traces

When Landslide identifies a bug, it outputs a *decision trace.* This trace reports what kind of bug was detected, and also reports each decision point in the current interleaving: which thread was running, a trace of its stack when it was switched away from, and the thread that Landslide caused to preempt it. With this trace, the user can better understand the concurrent execution that exposed the bug. In Code 3 we show an example decision trace, which depicts a sequence of thread interleavings that can expose the bug in the example from the Systematic Testing section.

"With this trace, the user can better understand the concurrent execution that exposed the bug."

```
USE AFTER FREE: read from 0x15a8f0 at IP 0x104209
Block 0x15a8f0 was allocated by thread 3 at (...)
   and freed by thread 4 at (...)
Decision trace follows:
1:  switched from thread 3 -> thread 4 at:
   0x105a10 in context_switch,
   0x1041f4 in thread_fork,
   0x10362b in thread_fork_wrapper
2:  switched from thread 4 -> thread 3 at:
   0x105a10 in context_switch,
   0x104681 in yield,
   0x104570 in exit,
   0x103708 in exit_wrapper
```

```
Current thread 3 at:
   0x104209 in thread_fork,
   0x10362b in thread_fork_wrapper
```

Total decision points 24, total backtracks 5

Code 3. An example decision trace that Landslide would emit when it finds a bug. This particular decision trace represents the example use-after-free bug in `thread_fork()` presented earlier.

Source: Landslide: Systematic dynamic race detection in kernel space.[3]

Results

We evaluated Landslide in two ways: first, by instrumenting two prior-semester student kernels to measure the exploration time needed to find different races, and second, by meeting with current-semester student volunteers, before they submitted their kernel for grading, to see if they could find bugs on their own with Landslide. (The volunteers were chosen from students with free time, and were therefore not chosen at random.)

In the first phase, we instrumented one kernel written by a teaching assistant in a previous year and also one student kernel later graded by that TA. We configured Landslide to search for five complicated well-known race conditions. In addition to finding all five races, Landslide also found a sixth previously unknown race in the TA's own kernel. Using additional decision points only on calls to *mutex_lock()*, Landslide found each of the six bugs in 11 to 57 seconds on a 2.6 GHz Intel® Xeon® server, executing between 1 and 377 distinct interleavings per bug.

"In addition to finding all five races, Landslide also found a sixth previously-unknown race in the TA's own kernel."

In the user-study phase, we found that students spent on average 119 minutes (60 to 158) on the required instrumentation, and a further 36 minutes (10 to 60) refining Landslide's search. Of the four groups who finished the required instrumentation, all four found previously unknown bugs in their kernels: two races and two deterministic errors. These bugs manifested as infinite loops, a kernel panic, and a use-after-free. Despite wishing the instrumentation were easier, the students reported that they found working with Landslide rewarding.

"All four groups found previously unknown bugs in their kernels: two races and two deterministic errors."

Future Work

There are several promising future work directions for Landslide that we would like to explore. These include incorporating new testing techniques, such as parallelized search, state space estimation, and new state space reduction techniques. They also include extending Landslide to support more complicated kernel features, such as symmetric multiprocessing and device driver nondeterminism.

Other Testing Techniques

The most notable bug-detection predicate that Landslide does not yet incorporate is data race detection.[2][7] A data race is defined as a pair of memory accesses done by two distinct threads on the same address, at least one

of which is a write, where there is no synchronization or dependency between the two threads at the time of either access. Many tools already exist for identifying data races, but we anticipate that searching for them with Landslide could additionally help guide Landslide's search towards thread interleavings more likely to have bugs based on such data races.

Ongoing research exists in several other techniques for coping with the exponential nature of the state spaces associated with systematic testing. Among these are parallelized dynamic partial order reduction[8] and dynamic interface reduction[9].

Extending Landslide's Concurrency Model

Landslide's present incarnation makes several limiting assumptions about the concurrency model of the kernel under test. Chief among these are the assumptions that the kernel schedules threads only on one processor at a time, and that the timer interrupt is the kernel's only source of nondeterminism.

We anticipate revising the concurrency model to incorporate SMP scheduling would be a relatively minor change, as the overall structure of the state space tree remains the same, though some context switches would instead be cross-CPU switches. Unlike all context switches in the current uniprocessor model, such context switches would not necessarily involve executing any scheduler code. Incorporating device driver nondeterminism, however, will be more of a challenge, as in addition to context-switching to an arbitrary thread at any decision point, nondeterminism can also arise from either taking interrupts to receive input from a device or from context switching to a device driver's dedicated handler thread.

Lifting these limitations would be a significant step towards making Landslide applicable to real-world kernels such as Linux. Overall, we are optimistic for the future of systematic testing for concurrency bugs, and we hope to see sophisticated bug-finding tools along these lines in due time.

References

[1] Patrice Godefroid. VeriSoft: A Tool for the Automatic Analysis of Concurrent Reactive Software. In Proceedings of the 9th International Conference on Computer Aided Verification, CAV '97, pages 476–479, London, UK, 1997. Springer-Verlag.

[2] Konstantin Serebryany and Timur Iskhodzhanov. ThreadSanitizer: data race detection in practice. In Proceedings of the Workshop on Binary Instrumentation and Applications, WBIA '09, pages 62–71, New York, NY, USA, 2009. ACM.

[3] Ben Blum. Landslide: Systematic dynamic race detection in kernel space. Master's thesis, Carnegie Mellon University, Pittsburgh, PA, USA, May 2012. CMU-CS-12-118.

[4] David A. Eckhardt, "Pebbles kernel specification," 2012. [Online]. Available: http://www.cs.cmu.edu/~410/p2/kspec.pdf.

[5] Randy Bryant and David O'Hallaron, "Introducing computer systems from a programmer's perspective," in Proceedings of the 32nd Technical Symposium on Computer Science Education (SIGCSE). Charlotte, NC: ACM, Feb. 2001.

[6] Fabrice Bellard. 2005. QEMU, a fast and portable dynamic translator. In Proceedings of the annual conference on USENIX Annual Technical Conference (ATEC '05). USENIX Association, Berkeley, CA, USA, 41-41.

[7] Stefan Savage, Michael Burrows, Greg Nelson, Patrick Sobalvarro, and Thomas Anderson. 1997. Eraser: a dynamic data race detector for multithreaded programs. ACM Transactions on Computer Systems 15, 4 (November 1997), 391–411.

[8] Jiri Simsa, Randy Bryant, Garth Gibson, Jason Hickey. Scalable dynamic partial order reduction. Third International Conference on Runtime Verification (RV2012), 25–28 September 2012, Istanbul, Turkey.

[9] Huayang Guo, Ming Wu, Lidong Zhou, Gang Hu, Junfeng Yang, and Lintao Zhang. 2011. Practical software model checking via dynamic interface reduction. In Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles (SOSP '11). ACM, New York, NY, USA, 265–278.

Author Biographies

Ben Blum is a PhD candidate in the Computer Science Department at Carnegie Mellon University. He first implemented Landslide as the research topic for his Master's degree at CMU and is continuing the work during his PhD studies. Ben has additionally served as a teaching assistant for 15-410 for three semesters. His web site is at http://www.cs.cmu.edu/~bblum.

David A. Eckhardt is an associate teaching professor of computer science at Carnegie Mellon University. He joined the faculty after completing his MS and PhD at Carnegie Mellon and BS in Computer Science at The Pennsylvania State University. Dave received an Intel Foundation Graduate Fellowship and was co-inventor of a patent with Intel Senior Fellow Kevin Kahn. He has taught Operating Systems at CMU continuously since 2003 and has supervised student projects based on the Linux, FreeBSD, Haiku, OS X, and Plan 9 operating systems. Dave's research interests include operating systems, wireless networks, and high-performance networking. His web site is at http://www.cs.cmu.edu/~davide/.

Garth Gibson is a professor of computer science at Carnegie Mellon University, the cofounder and chief scientist at Panasas Inc., and a Fellow of the ACM. He has an M.S. and Ph.D. from the University of California at Berkeley and a BMath from the University of Waterloo in Canada. Garth's research is centered on reliable scalable storage systems for parallel and cloud computing and he has had his hands in the creation of the RAID taxonomy and the IETF NFS v4.1 parallel NFS extensions. Garth is also an investigator in the Intel Science and Technology Center for Cloud Computing. His web site is http://www.cs.cmu.edu/~garth/.

EARLY HARDWARE REGISTER VALIDATION WITH SIMICS*

Contributors

Alexey Veselyi
Intel Corporation

John Ayers
Intel Corporation

This article describes the use of Wind River Simics*, a full-platform functional simulator, for early validation of hardware register specifications. The Simics model becomes one of the first consumers of the registers, and can find several types of errors earlier, and sometimes with a wider scope, than hardware-based validation. The article is based on an actual experience of collaboration between Simics model developers and hardware architects within Intel during development of an Intel® Xeon® chip. Simics was proved valuable as a validation tool and contributed to shift-left (reducing time to market) for hardware development.

Introduction

As the complexity of Intel hardware is increasing, the hardware register count in the new platforms is growing rapidly. Increasing amounts of control and status states are becoming architecturally visible to manufacturers, forming a key part of competitive advantage and a liability for customer-visible bugs. At the same time, hardware architects are being faced with the need for shorter project development cycles and a need for earlier (shift-left) engagement with teams that are using register specifications in their development. To handle these conflicting requirements (more complexity and having it come in earlier), the architects must use automated register validation techniques to validate register specifications. Early incorporation of validation in the register architecture definition process is vital for the early appearance of mature specifications.

"at the present time the problem of early pre-software register validation has no adequate solution"

At the later stages of development, validation of architectural definitions is performed by software developers (that is, during BIOS development), but at the present time the problem of early pre-software register validation has no adequate solution: some attempts at formal validation have proved to be slow and error-prone; most errors are found by simple observation, which cannot bring confidence in the status of register maturity.

RTL-based (hardware-based) register validation options are subject to significant dependencies on the integration of IP blocks into working models and the functionality of the global register access fabrics.

Register specifications lack an early target for validation in the project feature specification phase and design execution phase. They also lack a sense of delivery urgency until later in the project. Specifications are very likely to have impeding errors that face the initial bring-up of the RTL validation environment.

This article describes the use of Wind River Simics, a full-platform functional simulator, for addressing the need for early register validation during the development of an Intel Xeon chip. Simics is being used more and more for pre-silicon software validation by multiple groups at Intel. This validation approach proved very promising, and the intention is to adapt it for broader use.

The process of collaboration with the hardware design team is set up as follows. Model developers start implementation of the hardware model using early register specifications. The Simics model becomes one of the first consumers of the register specification and is able to find several types of specification errors significantly earlier than other RTL-based validation options, and in some cases with a perspective not available to RTL-based options. The approach detects not only register construction errors but also allows for validation of key architectural register specifications against legacy-derived behavior assumptions. Additionally Simics creates the possibility to define high-level functional tests for the new platform. These tests can cover most of the platform's functionality, with the exception of new or heavily redefined features. The team of Simics model engineers provides feedback to hardware architects for every register definitions drop.

"The Simics model becomes one of the first consumers of the register specification"

This article discusses in depth the scope of register validation using Simics and the ways to perform it. It is based on an actual experience of such collaboration between Simics model developers and hardware architects within Intel during the development of an Intel Xeon chip. Simics proved to be a valuable tool for finding bugs on the pre-software stage, thus speeding up hardware development and promoting the shift-left paradigm (reducing product time to market).

Early Register Validation Using Simics

Software simulation plays an important role in the shift-left of hardware development process. We will be discussing Wind River Simics, a functional full-platform simulator, which is becoming the de-facto standard simulator for many use cases at Intel. The usage of Simics is being incorporated into BIOS, driver, and other software development processes throughout Intel for pre-silicon, as well as post-silicon validation. This is a standard and well-known approach to the use of a functional simulator for validation purposes.

Simics model developers and hardware architects engaged early in collaboration on the project. At that early stage, the register specifications were only beginning to appear, and the overall development flow was just starting to form. So it became apparent that we were facing the opportunity to try Simics as a validation tool at the pre-software stage, which had not been attempted before.

"we were facing the opportunity to try Simics as a validation tool at the pre-software stage, which had not been attempted before."

We will first talk about register validation in general, determine the gaps in the currently established process, and then describe the collaborative process that was set up to tackle these gaps.

Hardware Register Validation

Architectural register definitions are a crucial form of an interface specification between software (or firmware) accessing agents and the implemented logic. As this interface connects two very distinct domains, there are multiple perspectives or validation targets. Simics advocates for the software validation perspective.

Current hardware-based validation of registers exists in two forms: validation of registers, and feature validation that implements register programming and tests for functionality. Hardware-based validation targets to be an interface validation with some visibility of the physical implementation side of the interface, specifically the register instance. It does not include the logic function behind the register, but instead validates that registers accesses are correct and that the register instantiation abides by several attributes defined in the specification. Example attributes are: address, visibility, lock, register type, access type, and reset/defaults (by asserting reset sequence). Hardware-based register validation is very explicit to not target register impact on chip behavior, leaving some of this for cluster, full-chip, or Uncore-scoped feature validation to incorporate register reads and writes within the validation of targeted features. Even feature validation lacks the perspective of the platform and the register programming and reading sequences used by software/BIOS.

Hardware-based register validation and feature validation do provide significant coverage of validating architectural registers against hardware behavior. However, these register and feature validations depend significantly on RTL functionality and integration. Functionality must be there for the register access fabric, for the emulators and monitors, and so on. Additionally the hardware-based validation targets never have the software and platform perspective that Simics has. As a result, the hardware-based validation effort starts with a register database that has not been validated sufficiently for rudimentary register construction faults or for architectural behavior of the integrated IP blocks scaled to the new project goals.

The diagram in Figure 1 captures the lifecycle of register definitions during a project when Simics provides an early register validation platform. The lagging enablement of hardware-based validation is shown in this drawing, as is the opportunity for the Simics platform to perform as a backdrop for the feature specification and design execution phases of architectural register validation. Simics serves to provide timely feedback on architectural behavior deviation from the prior reference product. With these earlier discoveries the project architecture owners have time to establish expectations before design teams have begun investing heavily in the specifications.

"With these earlier discoveries the project architecture owners have time to establish expectations before design teams have begun investing heavily in the specifications"

The shift-left of register validation and of external programmer specifications ahead of the slow ramp of RTL implemented registers are the key concepts in Figure 1. The subsequent sections define the place for Simics full-platform simulator in enabling this scenario.

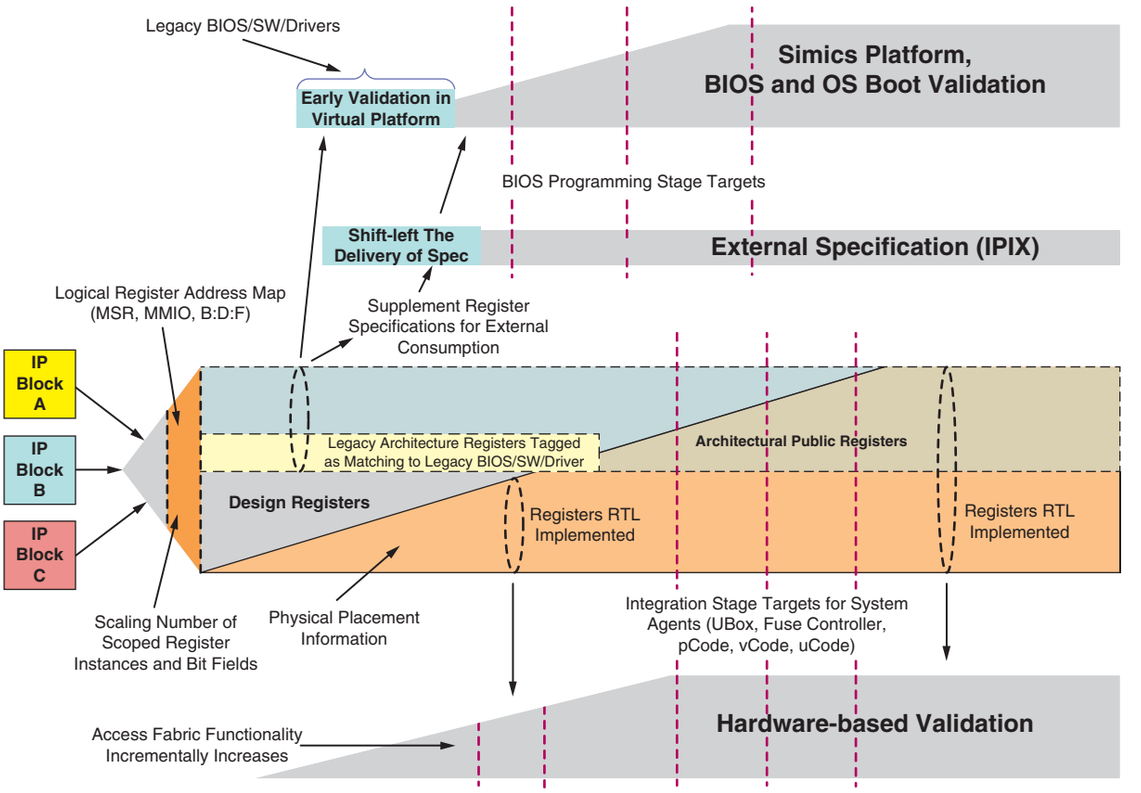

Figure 1: The register definition lifecycle with simics as a validation platform (Source: Intel Corporation, 2013)

Early Validation of Registers for Construction

The validation targets for Simics must consider that this environment is not based on RTL and that it therefore poses a higher risk of deviation from actual silicon behavior. To mitigate this there are two goals: to not target lower-level register implementation details, and to validate register data exactly as it is consumed for hardware-based validation tools. In this case, hardware-based register and feature validation matches very well with early register validation using Simics.

Simics offers an opportunity for earlier detection of register construction errors:

"Simics offers an opportunity for earlier detection of register construction errors"

- Register address errors: overlaps, erroneous placement in PCI Header address, and so on.
- PCI device definition errors: bus-device-function allocations contradicting the system address plan, headers not compliant with the PCI standard, erroneous PCI class-code, incorrect device header-type values.

Database checks find some register address errors; however, it is very difficult to manage the list of heuristic-based checks without some segmenting of the

registers according to their impact. Simics offers a focus point for the most architecturally significant registers to be reviewed for construction and ensures that legacy definitions take the precedence.

Early PCI header device definition and bus-device-function allocation validation is paramount for establishing the infrastructure of register addresses and for alignment with the project device plans. Project PCI endpoint, root complex, nontransparent bridge (NTB) configurations and so on should be initiated with confidence that the most basic definitions match legacy access expectations.

Early Validation of Registers for Derived Behavior

Simics offers a platform to plug in legacy BIOS routines and then validate legacy architectural registers for not straying behaviorally from seed BIOS expectations. After projects capture IP register definitions, they move into a stage of scaling for the IP instance counts, expansions, and structural changes to system agent and project defining registers (such as, for example, CPUID), and placement in a system address map for public MSR and CSR/CFG registers. These changes have consistently been shown to introduce unintended editing mistakes or incomplete edits, as well as compatibility breaks with other architectural specifications.

BIOS routines are each associated with a list of registers to be accessed. The architectural registers that are in support of legacy features are extracted from the database based on these lists of registers. BIOS routines are selected from key reference BIOS releases, such as the prior product in the product segment. For example, as the DDRIO in successive projects moves from two channels to three channels, there is significant value to both the BIOS development and hardware design in having the DDR training or configuration programming scale up in count in expected ways and in validating fundamental behaviors.

"Simics provides a very effective launch point for early BIOS programming targets, possibly ahead of related RTL development"

After validation of legacy architectural functionality, Simics, that is consuming early register definitions, provides a very effective launch point for early BIOS programming targets, possibly ahead of related RTL development. Examples are: MMCFG/SNC/SAD/TAD/MMIO translation tables, routing tables, range settings, and PCI enumeration flow.

Additionally Simics creates the possibility to define high-level functional hardware tests for the new platform. These tests can cover most of the platform's functionality, with the exception of new or heavily redefined features.

The Simics team provides feedback to the hardware architects for every register definitions drop.

Simics as a Validation Tool

We now describe the process of collaboration established between the Simics engineers and the hardware architects working on the project.

The Simics engineers are included in the iterative development process of the hardware design team. As soon as a new release of the register definitions is ready, they can begin exploring the registers using Simics. This, in turn, consists of several stages. Each one of these stages has its own functional scope to which the recognized bugs in the specifications belong. The stages are described in Table 1.

Development stage	Validation scope
Register structures parsing and processing	Register overlaps and collisions; misplaced registers Standard PCI configuration registers missing or incorrect Invalid class-code and header type values
Platform setup and compilation	Device list mismatch with platform HAS (Hardware architecture specification) Existence of registers against a closely-related predecessor platform (unless the new specifications explicitly state the difference): Registers that are expected to be programmed by a future BIOS, based on past experience; Registers with side effects used during OS boot IIO (PCIe, VTD, NTB, error detection registers)
Preliminary boot test with a simple GPL BIOS (SeaBIOS)	Registers required for BIOS boot: CPUBUSNO and MMCFG rule registers Some PCI enumeration issues
OS boot tests (different versions of Linux, Windows, SVOS, and so on)	MMIO mapping rules PCI enumeration issues Registers/fuses required for interrupts functionality
Specific feature testing	Registers required to support the features: PCIe ports Legacy and non-legacy interrupts NTB region mapping Reset etc.

Table 1: Simics platform development stages and their validation scopes (Source: Intel Corporation, 2013)

All of the observed issues are promptly shared with the hardware architects, who take the report into consideration for the next register definitions

"a reported bug in a single register can be an indication to the hardware team of a bigger problem"

release. Sometimes a reported bug in a single register can be an indication to the hardware team of a bigger problem in the specifications: for instance, a common error in register access types or a misplaced register bank.

Validation Based on Prior Definition

A large part of the mentioned validation scopes share a common validation assumption: the platform in development should adhere to some legacy expectations. On the one hand, this makes such an approach limited to only validating the functionality inherited from previous-generation specifications. New features, which are a big risk factor, and which are likely to become the focus of attention for early software developers, are only covered by the generic validation scopes. But on the other hand, legacy functionality forms the bulk of overall functionality of a modern Intel platform. After the hardware architects have formed the preliminary understanding of how the new platform differs from its predecessor, they can define which areas should be excluded from prior-definition-based validation.

Generic Errors

On the first stages of the described development cycle, validation is performed the following way. The Simics team receives register definitions (containing register offsets, access types, default values, fields) as an XML database, ConfigDB, which is one of the standard formats for register definitions exchange. The Simics engineers have developed tools to automatically process and convert the XML into a format that the Simics framework can understand (Device Modeling Language, DML). Another commonly-used register specifications format, CRIF, can also be used during this process.

During this stage, generic errors can be detected, such as overlapping registers, invalid register and field sizes, invalid placement of custom registers into the standard PCI header block, and so on. Also, on this stage Simics detects invalid class-codes and header-type values when they mismatch the rest of the PCI header definition—which happens to be a frequent bug in register specifications.

Here we should note that some of these errors can be found automatically by other validation tools, so this scope is not fully limited to Simics.

Legacy Expectations Mismatch

More features of Simics come into play on the next stage, where it is possible to validate the platform's behavior based on legacy expectations. This includes compliance with existing standards (PCI header configuration registers matching PCI specification; PCI enumeration) and preservation of functionality from previous platforms (MMCFG/SAD/TAD/MMIO translation tables, routing tables, range settings).

The Simics API allows for side effects of registers to be described separately from register definitions. The Simics team takes advantage of this capability

and always makes the effort not to mix register definition with register implementation. This makes it possible to port side effects from a previous model to the new one, after filtering out the side effects that the new platform should not have. Custom-register side effects constitute the major part of the overall model implementation.

"an attempt is made to combine old side effects, which are already thoroughly tested, with new register definitions"

As a first approximation of the new functionality, an attempt is made to combine old side effects, which are already implemented for a previous platform and thoroughly tested, with new register definitions. Then the platform is set up and compiled using the Simics framework. For registers that match the old definition, the amount of attention required from a developer is minimal. For registers that are different, the developer should consult the platform documentation and understand the nature of such differences. The Simics framework will point to every one of these registers during platform setup. Developers then have to look through every register and make a decision about the extent of the possible reuse of the old implementation.

For registers that changed their names, sizes, set of fields or locations, the developer usually has to make some trivial changes, after consulting with the platform specification. But if registers are missing or contain changes that are incompatible with previous specifications, these observations should be delivered as feedback to the hardware architects—some of the changes are purposeful, and should be taken into account by the model developers, while others are bugs in the register specification, which should be resolved in future iterations. The Simics model is not blocked by these bugs—some functionality of the model is just temporarily disabled (or legacy definitions are used) until the specifications are correct.

At this stage the developer also pays close attention to the rearranged bus/device/function (BDF) map of the new platform and captures possible discrepancies between the map and the register definition. This is achieved by first creating the skeleton of the platform (containing dummy devices) in conformance with the BDF map, and only then applying the register definitions. The overall flow of platform development is illustrated in Figure 2.

Running Workloads

As soon as the Simics engineers team is done with triage and gathering low-level register errors, they can proceed to the functional level of validation against legacy expectations. At this stage the platform is up and running and is register-accurate relative to the latest register specifications. Now Simics can attempt to run actual workloads to see how the platform is able to operate as a whole.

"Now Simics can attempt to run actual workloads"

The first workload that is run is SeaBIOS, a GPL implementation of a 16-bit x86 BIOS developed primarily for running on emulators. The BIOS requires minor modifications to run on new platforms—they are usually limited to changing device IDs in the source code; sometimes they include some shuffling of device or register locations. Any other discrepancy should be, once again, compared to the specifications and reported to the hardware architects. The BIOS boot validates the operation of PCI configuration space mapping, PCI

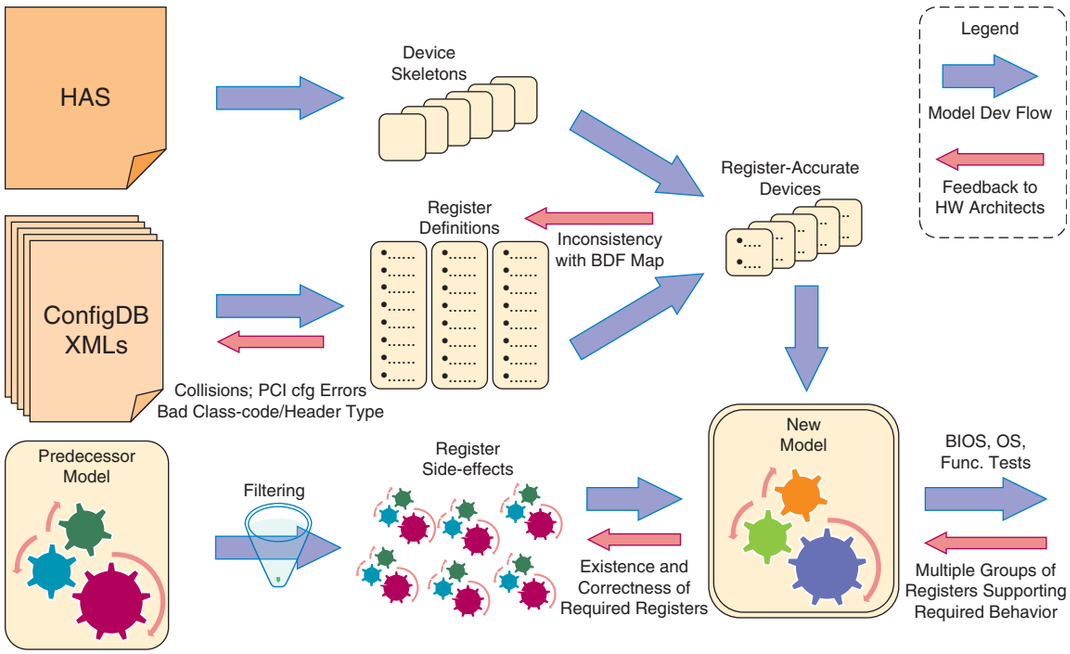

Figure 2: Simics platform development stages and their validation scopes (Source: Intel Corporation, 2013)

enumeration, and some other features, and also opens the path to running actual operating systems on the early model.

When the specifications are mature enough for SeaBIOS to successfully boot, the model attempts to boot operating systems. The Simics team has a set of disk images with installed systems that are used by their customers with various platforms. This includes different versions of Linux, Enterprise Linux and SVOS, desktop and server versions of Windows. The operating system's boot, also containing driver initializations, can validate a lot of device-specific functionality, MMIO, interrupts, and so on.

High-Level Behavior

The next important stage of the development cycle is the implementation of tests for specific high-level functionality. This includes PCIe ports operation, legacy and non-legacy interrupts, NTB, networking, different types of reset, and so on. The Simics team is working on increasing their pool of tests that can be later used for the validation of future platforms. Using these tests makes it possible to keep track of major functional areas of the register specification. In case of any issues with the tests, the root causes of the issues are determined by the Simics team, and the register bugs are reported to hardware architects.

"Using these tests makes it possible to keep track of major functional areas of the register specification"

After all the stages are complete and the hardware architects have received feedback, they can begin work on fixing the errors that were found. At the same time, they are implementing features that were previously missing in

the new register definitions. So, when a new drop of the registers is ready, a new iteration of the cycle can begin.

The mentioned tests were traditionally used by the Simics team for the validation of their own models. What we are proposing now is the use of the same (or similar) tests on early stages of hardware development for validation of specifications. This is made possible by the availability of many platform models previously implemented by the Simics team with a high level of detail.

Required Resources

With a substantial bank of mature models already developed at Intel, an early platform model can be assembled within approximately a week of work. On the following hardware specification iterations, the turnaround time is usually also under a week.

"an early platform model can be assembled within approximately a week of work"

The model that appears during the early specifications period is only an outline of the future model, although some parts that can be taken from another platform can already be fully working at this stage. As the specifications mature, we approach the BIOS development stage with an already existing functional model. That means the work that was put in model development is not lost, but is passed on to the later stages to be utilized for the more standard use cases: pre-silicon software and hardware codevelopment, and, subsequently, post-silicon validation.

Results

The results of using the Simics platform in the project for early register validation reflect that the initial implementation has focused more on construction validation.

From the standpoint of the hardware architecture team, Simics provided tangible bug discoveries in both the register address attributes and the PCI device configuration settings. In the project, the ability to have early validation testing provided timely feedback on the project HAS documents and helped shift earlier the delivery of key external specifications.

"the ability to have early validation testing helped shift earlier the delivery of key external specifications"

The Simics model became a destination for the delivery of register specifications that preceded the enablement point in hardware-based validation. This had a clear and compelling influence on pulling register specification to an earlier point in the project lifecycle than would have been there otherwise.

We provide some examples of actual bug discoveries that were made for the register specifications using the described process:

- Multiple errors concerning overlapping register address values: fully overlapping registers or a 64-bit register placed only four bytes before the start of the next one. These errors would have resulted in access aliasing and potentially corrupted Read/Write effects. *Found on the register structure processing stage.*

- Registers placed in Header Offset space (0x0–0x3F) that are not legitimate PCI Header registers. Since hardware redirects accesses differently based on offset, there would have been failed access and potential bad Read/Write effects. *Found during platform setup.*
- Arrayed registers that became noncontiguous, while being contiguous in a previous platform. While this is not, strictly speaking, an error, a future BIOS can anticipate that they would be contiguous and implement iterative increment addressing of this register set. So it is useful to point out this issue to architects to see if breaking such logic was intentional. *Found in the process of porting legacy register side effects.*
- Register offsets that are in extended offset range (0×100 and higher) when BIOS expects the CFG accesses to be in Legacy mode using CF8/CFC port-in/port-out, with only 8-bit offsets. This error makes the register unreachable in the early reset timeframe. *Found during BIOS boot.*
- Incorrect setting in a Function 0 header_type.multi_function_device field, which did not reflect the addition of Functions 2 and 3 in the Function allocations. This error would have caused a failure to PCI-enumerate those two new functions for subsequent accesses. *Found during the PCIe port functional testing.*

Each of these discoveries significantly preceded the readiness for coverage by hardware-based validation.

Summary

Wind River Simics, as a functional simulator, is being used for validations of hardware specifications and software on the pre-silicon and post-silicon stages. We have shown how its use as a validation tool can be extended to cover very early hardware architecture specifications.

Hardware architects reported having a gap in the established specification development process: on the early stages of the project, no validation was possible and no consumer existed for the register definitions. An attempt was made to cover this gap with Simics, by means of engaging in early collaboration between the team of Simics modeling engineers and the hardware architects.

"Hardware architects' feedback states that validation with Simics significantly helped shift-left the delivery of architecture specifications"

Overall, we have achieved very positive results. Hardware architects' feedback states that validation with Simics significantly helped shift-left the delivery of architecture specifications. However, this is still an early solution, and due to time and resource constraints its potential was not fully realized in the given project. Validation helped find many errors in the register construction area, so this type of validation can be considered well established. As for validation against legacy BIOS assumptions, only a few errors were found, although this approach looks very promising.

The net result was that while the theoretical value is high, in practice the impact was positive, but limited.

We expect to use the described approach for early validation of future projects, engaging the Simics team early on in the architecture specification development process. As the approach matures, we will be able to compare the value achieved in practice with its theoretical value. If we discover it to be successful, then the further intent is to make Simics one of the standard tools for early architecture definition validation at Intel.

Author Biographies

Alexey Veselyi is a Simics Intel architecture model engineer, and has been with Intel for two years. He can be reached at alexey.s.veselyi at intel.com.

John Ayers, HPC server architect, has been with Intel for 15 years. He can be reached at john.r.ayers at intel.com.

SOFTWARE POWER AND PERFORMANCE CORRELATION ON SIMICS*

Contributors

Parth Malani
Software and Services Group,
Intel Corporation

Mangesh Tamhankar
Software and Services Group,
Intel Corporation

Early estimation of driving forces like power and performance for future hardware platforms using Virtual Platform (VP) tools such as Simics can greatly improve the product design cycle, although a small gap between simulated performance and its actual value can adversely affect other simulation derivatives such as power. In this article, we mitigate any such gap through a streamlined system tuning methodology to achieve high degree of performance correlation on Simics, which is within 2 percent of actual hardware performance. Using the tuned performance as a foundation, we build a power model on top of Simics that provides accuracy of within 5 percent for various software multicore compute workloads. The benefits offered by this experiment are twofold. First, it can help system designers working on architecture exploration by providing insights into how to properly model and tune the Simics system to reflect crucial design details. Second, platform architects and application developers can take advantage of this accurately tuned system for early estimation and exploration of power and thermal, which are directly dependent on simulated performance. In a broader scope the beneficiaries of this work may include application/driver developers, system designers and architects, marketing professionals, process engineers, and so on.

"Improved performance accuracy can improve estimates on other derivatives such as power."

Introduction

In traditional product design flow, software design and exploration happens only after hardware is physically available. The Software Shift-Left phenomenon has created an interesting space in hardware-software co-design domain wherein software design phase is shifted ahead in the overall product design cycle taking place in parallel with hardware design. Such shift in design cadence shortens product time to market and enhances design quality. Application developers can explore and optimize power and performance of their software code without having to wait for silicon prototypes to be available. Pre-silicon platform level power and performance simulation through VP-based tools can help improve a variety of design steps ranging from architecture and power management exploration to power, cost, and area budgeting, as well as time to market. One of the major limitations of many current simulation methods is reduced scope of simulation and lack of system level details such as OS involvement. Today's dynamic applications execute beyond CPU boundaries by using other system components such as the GPU and ASICs. Traditional simulation methods cannot model the interaction between these system components. Virtual Platform (VP) based simulators such as Simics[1] offer an attractive solution to model system interaction.

"Pre-silicon platform level power and performance simulation through VP based tool can speed-up product design cycle."

However, modeling accuracy of functionally accurate VPs can significantly affect their potential benefits, making it imperative to correlate them against existing hardware to establish a reliable and accurate foundation. This article presents a streamlined performance tuning methodology for multicore software workloads running on a Simics-based X86 system. The experiments indicate that it is possible to achieve good performance correlation on Simics by deliberately tuning its system model configuration. Our results demonstrate a performance accuracy that is within 2 percent of the actual hardware. We then explore the possibility of adding a power modeling capability on Simics and present a Virtual Power Monitoring (VPMON) framework, which utilizes the tuned Simics performance and Simics tracing extensions to simulate software power. Correlation of simulated power with hardware measurements for various software workloads shows within 5 percent power modeling accuracy.

"It is possible to achieve good performance correlation on Simics."

The increased simulation speed offered by VP-based tools is a virtue of their functional performance accuracy. This accuracy can also affect other derivatives of performance, such as power and thermal. Correlating power consumption with performance has been proven very successful by many power modeling approaches in past. Estimation of runtime processor power consumption based on performance counters or events has been well explored on numerous architectures such as IA[2][4], AMD[6], and ARM[7]. Some of these works[2][3][4] also demonstrate microarchitecture level power characterization capabilities provided by their power models. Techniques have been proposed[2][6] to accurately estimate multicore processor power. All of these power modeling techniques rely on performance statistics measured on actual hardware. It serves as an efficient power analysis solution for the current generation of hardware products only. Simulation-based power modeling is an attractive solution when the target hardware is not available. Varma et al.[8] have proposed a simulation-based power modeling methodology and evaluated it using a modified Intel Xscale® cycle-accurate simulator (Xsim) with SystemC*-based transaction level models. They demonstrated results with power modeling accuracy within 10 percent of measured data while attaining simulation speed in excess of 1 MIPS. Although this methodology is meant to be generic for any embedded system, the authors did not discuss its applicability to a pre-silicon power modeling scenario.

We envisage VPMON to be used as a power projection tool wherein a user can configure Simics with performance models of future hardware and model their power consumption. Also it can be utilized as a power simulator, providing means for workload power exploration on current and future hardware. For example, application programmers may want to evaluate the impact of their software code changes on the power consumption. In many cases the accuracy may not be as desirable as the polarity of the power impact. Apart from these usages, VPMON can be extended to study thermal behavior of the system. Transforming performance counters or power consumption to temperature has been explored before by Chung et al.[9] and Bellosa et al.[5] respectively, and is proven to be useful for system thermal management.

"VPMON power simulator provides means for workload power exploration on current and future hardware."

"Our approach is highly portable within different Intel architectures."

To best of our knowledge, the only work targeting Simics based power modeling has been proposed by Bartolini et al. [10], which adds power and thermal modeling capabilities in the Simics framework by integrating multiple external tools such as Matlab and the GEMS memory simulator. The power model relies on performance counters modeled in GEMS as well as Simics internal registers for Intel® Core™ architecture, which is a platform-specific feature. Our approach relies purely on the instruction set stream and is thus highly portable within different Intel architectures. The authors tested their technique with synthetic workloads stressing various levels of cycles per instruction (CPI). We have tested the proposed method with real multicore multithreaded software kernels. It should be noted that the work in Bartolini et al.[10] also models Dynamic Voltage and Frequency Scaling (DVFS), which we have not targeted here.

"Accuracy and speed of power simulation depends on simulated performance."

To achieve the best tradeoff between accuracy and speed of power simulation is particularly challenging when modeled power depends on simulated performance. The correlation experiment is divided into two phases based on this fact. The two phases are outlined as below:

- Performance correlation through system tuning.
- Power modeling based on tuned system performance.

The rest of this chapter is outlined as follows. The next section, "Performance Correlation," explains the system tuning methodology applied to correlate application performance measured on Simics against the actual hardware and also shows the outcomes. The section "Power Modeling and Correlation" provides details of the VPMON framework and its implementation on Simics. This section also demonstrates the power modeling accuracy and simulation speed results. This is followed by "Summary and Conclusions."

Performance Correlation

A possible gap between software performance on a Simics-based system and actual hardware is first assessed. We then employ a streamlined tuning process to enforce a high degree of performance accuracy. Simics exhibits very attractive characteristics such as modularity, configurability, programmable APIs, OS awareness, and dynamic tuning of system parameters. Our performance correlation methodology is composed of two steps: 1) system configuration and 2) performance tuning and correlation. We first focus on configuring the Simics system as per the existing hardware specifications. It should be noted that a system may include diverse hardware components and devices such as CPU, GPU, memory, network card, and disks. Depending on the nature of the application, many of these devices may not get utilized at all. Removing such devices from simulation flow or using ad-hoc performance models can speed up the simulation.

We target a server system with Intel® Xeon® processors for the correlation experiment. Simics supports Xeon processor models and a multicore/multithreaded execution environment through its OS involvement feature. In

particular, as a reference platform, we employed a single-socket system using a Xeon processor with cores based on the microarchitecture formerly codenamed Sandy Bridge (SNB) . To model this system, we chose Simics 4.6, which has an SNB platform model available. Simics incorporates libraries of performance models supporting ISAs of numerous processors along with the standard chipsets and other system components such as buses, disks, and so on. Along with the virtual OS and user application program, it can simulate full system execution across entire platform.

Platform Configuration

We were able to run industry-standard multicore compute workloads such as Linpack "as is" in their binary forms on the Simics system. The performance reported by workload on the Simics system differed from its hardware counterpart because of the default Simics system configuration. However we were able to narrow this gap through tuning of various system parameters such as the number of threads, frequency, and instructions-per-cycle (IPC), and by adding performance models of crucial components such as instruction and data caches. The main idea we follow is to apply and limit the tuning to the components that fall in the workload's critical execution path. This is important because adding additional simulation models can reduce the simulation speed significantly.

"We were able to run industry-standard multicore workloads "as is" on Simics."

"The idea is to apply and limit the tuning to workload's critical execution path."

Table 1 shows the detailed platform configuration we used for the performance correlation experiment. We used Linpack and DGEMM (double-precision matrix multiplication) multithreaded workloads to represent compute-intensive user application programs. The host system on which Simics is running contains an Intel® Core™ i5 dual-core processor running at 3.3 GHz with 8 GB of RAM. We stuck to 2 GB of memory for the Simics target system to avoid simulation overhead for the host system. It did not affect the correlation because Linpack and DGEMM are compute-bound workloads. It can be inferred that the base Simics configuration differs from the reference platform in many aspects and their performance outputs thus will not be equal.

Configuration Space	Intel® Xeon®	Simics 4.6
System	Single socket Intel® Xeon® (formerly Romley) platform	
OS	Red Hat Enterprise Linux 6*, Intel® Compilers	
Application	Linpack, DGEMM	
CPU	Intel® Xeon® E5-2640, 6 SNB cores, Intel® HTT enabled, 3-level cache	6 SNB cores, Intel® HTT enabled, no cache
Frequency	2500/3000 MHz TDP/ Turbo	2000 MHz (tunable)
IPC	Variable	1 (tunable)
Memory	16 GB, 1333 MHz	2 GB (tunable)

Table 1: Simics vs. hardware platform configuration
(Source: Intel Corporation, 2013)

"Base Simics configuration is off from the hardware."

Tuning Methodology

We compared the actual GFLOPS performance reported by the workloads on the Simics system to the reference hardware and the correlation error is plotted in the chart in Figure 1. Data points, from left to right, on solid lines pertaining to each workload indicate progressively applied tuning and corresponding error in accuracy. As shown, the base Simics configuration stood at about -43 percent and -72 percent off from the hardware for Linpack and DGEMM respectively. We tuned the frequency as per the reference CPU and the gap was reduced, because the default Simics frequency was lower than hardware as shown in Table 1. Simics supports fixed IPC (finite number of simulation steps per cycle) because it is not cycle accurate. We tuned the IPC based on its architectural peak value and it significantly increased the performance pushing the error rate above 0 percent. Both frequency and IPC can be set per logical processor at runtime through attributes under Simics class hierarchy Romley.mb.cpu0.core[*i*][*j*], where *i* and *j* reflect physical and logical processor index respectively.

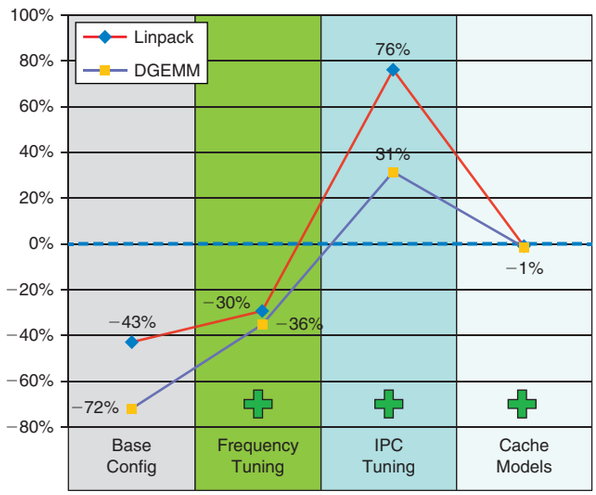

Figure 1: Application performance tuning on Simics (Source: Intel Corporation, 2013)

"We were able to achieve performance correlation within 1 percent."

Performance jumps significantly at this stage because the base configuration does not model memory hierarchy yet; that is, memory latency is not accounted for. Finally we added 3-level cache models as per the SNB core specifications and were able to achieve performance correlation within 1 percent. Adding memory hierarchy naturally degrades the IPC below its peak value. Also the compute-bound workloads we use here are cache intensive at best and thus we did not tune the memory model except to configure its frequency as per the reference platform.

Simics provides a sample global cache model called *g-cache*. We extended and modified it for multicore/multithread support. Table 2 provides the configuration details for cache models we used. The size of each cache is shown

along with its access latency. All the caches use random replacement policy by default. We used latency of 200 cycles for any accesses going to main memory.

Cache	Size	Snooping	Latency (# cycles)
L1 Inst	32 KB	MESI	3
L1 Data	32 KB	MESI	3
L2 Data	256 KB	MESI	8
L3 Data	20 MB	N/A	25

Table 2: Cache model configuration and timing
(Source: Intel Corporation, 2013)

This performance correlation exercise can be applied to any compute-bound workload. As we mentioned earlier, the idea is to apply and limit the tuning to the components that fall in the workload's critical execution path. This is important because adding additional simulation models can reduce the simulation speed significantly. We came up with two types of tuning: spatial and temporal. In spatial tuning, we make sure we only simulate the components that are required by workload. For example, we did not use an external memory model for a compute-intensive workload that is cache bound. The idea behind temporal tuning is to limit the tuning dynamically as per behavior of the workload.

Figure 2 depicts a typical execution scenario for a compute-intensive workload such as DGEMM. During the initial phase (INIT), it allocates memory and initializes data matrices followed by a phase that involves a computation process using this data. Because the INIT phase is not included in performance calculation (GFLOPS in this case), it may be beneficial to start the simulation with a basic configuration and add more details as and when desirable. Simics supports a halting mechanism and hotplugging of device/component models, which we utilized to trade off accuracy for simulation speed. We halt the simulation at the beginning of compute phase and attach the cache models to account for accurate performance modeling. We detach the cache models when the workload writes back the results from compute phase.

"Simics supports hotplugging of component models to trade off accuracy for speed."

It is interesting to note that the tuning input does not affect the high level behavior or functionality of the workload. However it has a direct impact on

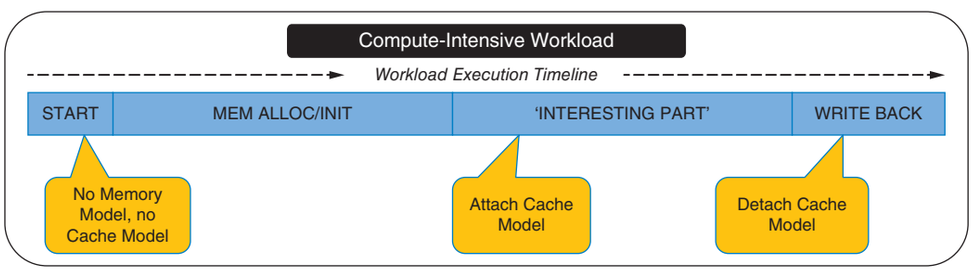

Figure 2: Simulation runtime optimization
(Source: Intel Corporation, 2013)

performance statistics such as C-state residency or instruction throughput, which we use for power modeling as discussed in the next section.

Power Modeling and Correlation

Besides performance correlation, our other important objective is to implement a power module on Simics that can output dynamic workload power at runtime. The power output can ideally be for an entire system including multiple components; however, we only explore the CPU power model here. Although Simics captures the performance part in fair detail, incorporating power simulation is challenging. However existing debug and profiling features of Simics can be utilized to come up with a power monitoring framework. The accuracy of power estimation will highly depend on how prolific the profiled information is, because most power modeling techniques rely on performance events or statistics. The user can create a robust foundation for power modeling by tuning system parameters such as timing and memory through Simics' programmable interface as mentioned in the previous section. It is also crucial how the power modeling framework utilizes and builds upon this underlying foundation to predict dynamic power. The VPMON power model provides runtime CPU socket power dissipated by the multicore workload running on top of Simics.

"VPMON provides runtime CPU power dissipated by multicore workload."

$$P_{dyn} = AF \cdot C_{dyn} \cdot V^2 \cdot f \quad (1)$$

Equation 1 shows how we calculate the dynamic power, which depends on variety of parameters, namely the Activity Factor (AF), dynamic capacitance (C_{dyn}), voltage (V), and frequency (f). The Activity Factor (AF) represents the dynamic power impact of the user application running on top of the hardware. It is the only variable in the equation and its value is modeled through Simics as explained in subsequent sections.

"Activity Factor represents dynamic power impact of the user application."

Power Monitoring Framework

Figure 3 shows the design and implementation of a Virtual Power Monitoring (VPMON) framework on Simics. A user application program is running atop a Simics platform, which contains a simulator API, a debug engine, and most importantly the performance models for various system components. It provides the simulation of application execution on a full system. The function-accurate component and device models are marked by *F* in Figure 3. Examples of such functional models include the CPU ISA model, external memory model, and cache models. Although we achieve reasonable power modeling accuracy using available Simics models, it may be required to improve them for platform- or component-level power modeling. For example, if a user wants to analyze the power of RAM due to data communication, the ad-hoc memory model may not be sufficient and should be improvised.

Performance simulation accuracy can be improved by tuning the system parameters based on the real system as well as by adding models of new components or devices as shown by the tuning block on the left side of the figure. The tuning and modeling effort will be reflected in the simulation.

A debug engine monitors system components at runtime and provides profiling, statistics, and debugging information.

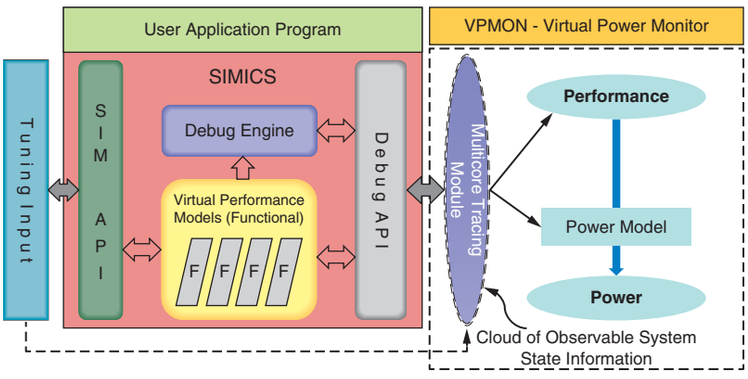

Figure 3: VPMON Power Modeling Framework (Source: Intel Corporation, 2013)

We implemented a multicore tracer module, which is essentially a dynamic cloud of performance-related information pertaining to each logical CPU. Simics provides a single-core instruction-tracing module written in C as a "plug and play" simulator extension. We modified it to support multicore platforms and added various monitors to calculate *AF* in Equation 1. In addition, the cloud may contain other system statistics and performance counters such as cache profile and thread topology. VPMON maintains a data structure *CPU_stats* to calculate and store performance statistics for every logical CPU trace entry. Tuning input such as frequency, number of cores, and cache information is also fed to the tracing module. The monitors capture crucial performance statistics observable within Simics through its debugging and profiling APIs. If the power model utilizes such information from all Simics components, there will be a huge cloud of data. However application power models can generally achieve sufficient accuracy with limited high level system observation. This is precisely the reason we were able to achieve good power modeling accuracy on Simics without going for cycle-accurate details.

"We achieve good power modeling accuracy without going for cycle-accurate details."

From the user perspective, VPMON execution flow is very simple, involving only two commands on the Simics command line. Whenever a user wants to analyze the power during workload execution, he or she has to halt the simulation and load the precompiled tracing module along with a VPMON wrapper script. The wrapper accesses VPMON tracer output at fixed sampling intervals in virtual simulation time through the *SIM_get_attribute()*and *SIM_set_attribute()* interfaces. The tracer module maintains a bucket of statistical information during each sampling interval, and the bucket is processed at the end of each interval through a callback function on an access through the VPMON wrapper. The power will be displayed every sampling interval on continuation of simulation from this point onwards. Optionally, VPMON can also output performance statistics if desired.

Power Modeling Accuracy Experiments

To evaluate the VPMON power modeling framework we used the same reference system as mentioned in Table 1. We added numerous workloads representative of diverse applications and correlated their performance against hardware based on the methodology discussed before. Next, we built a linear-regression–based power model, trained it using the measured power data for the reference hardware system with the Intel Xeon processor, and periodically sampled performance counters from a multicore tracer for various workloads. We set the sampling interval to 10 milliseconds of virtual simulation time; however, the framework can support any arbitrary value. We set the regression objective to minimize the sum squared error (SSE) between the modeled and measured CPU power for all workloads. The workloads exhibit variable execution time and we focus on average power inside their core compute loop for comparison. For each workload in the training set, an entry is made to the model consisting of observed performance counters such as C-state residency and IPC. VPMON samples these parameters through the multicore tracer.

"Training process correlates performance counters to measured workload AF."

During the training process, the power model correlates each performance counter to measured workload Activity Factors and outputs a single set of coefficients modeling the power contribution of each counter. We derive *AF* for each workload on an instrumented hardware system of Table 1 by measuring core power (P_{dyn}), C_{dyn}, voltage, and frequency. We used a synthetic power virus workload to calculate C_{dyn}. To make a single entry for each workload in the model, we take average value of sampled counters (and *AF*) over its entire compute loop. To summarize, the training set comprises of pairs of VPMON counters and measured *AF* pertaining to each workload. Training process is performed offline, and modeled coefficients and constants are incorporated into the VPMON multicore tracer to simulate runtime power.

To evaluate the power model we used a testing set that has additional workloads apart from the ones used in training. We compared the average simulated power within the compute loop of each workload to its hardware counterpart. Figure 4 shows the relative comparison of measured versus modeled power on Simics. Measured values are based on a fixed value of 1 and VPMON power is shown in a relative manner for nine different workloads. All the workloads execute a different number of threads ranging from six to twelve threads. SGEMM and DGEMM are matrix multiplication workloads that are compute bound in nature similar to Linpack. Stencil2D is a highly cache-bound workload. HiPwrWkld is a synthetic power virus workload used to stress the silicon Thermal Design Power (TDP). We also used these five workloads for the training process mentioned before. The remaining four workloads are fast Fourier transform (FFT) single- and double-precision kernels. For all workloads, the accuracy of the Simics power model is within 10 percent of measured value in the worst possible scenario. The average modeling (training) and projection (testing) accuracies are both within 5 percent of hardware measurements.

"Average modeling and projection accuracies are within 5 percent of hardware measurements."

Modeling average power may not be sufficient for workloads exhibiting highly dynamic power behavior. We also compared the instantaneous simulated power

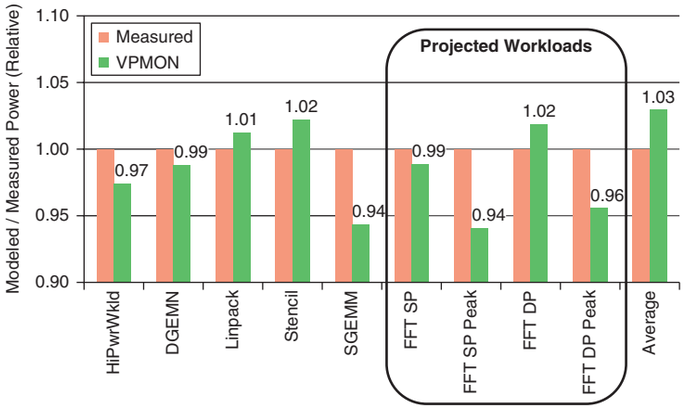

Figure 4: VPMON modeled vs. measured power
(Source: Intel Corporation, 2013)

of each workload to measured hardware power trace with an identical sampling interval (10 milliseconds). Both power traces were time correlated with help of an output flag set by the workload.

"Simulated and measured power traces are time correlated."

Figure 5 depicts the projected power versus time comparison between Simics and hardware using an Intel Xeon processor for a FFT DP workload that displays a dynamic power pattern. The worst-case accuracy in this example is still within 13 percent of measured value. The error in timing correlation is mainly due to two reasons: 1) a gap in Simics cycle-level performance and timing accuracy (that is, we were not able to correlate the performance well enough for this workload), and 2) this workload was not included in the training process and thus the power here is projected.

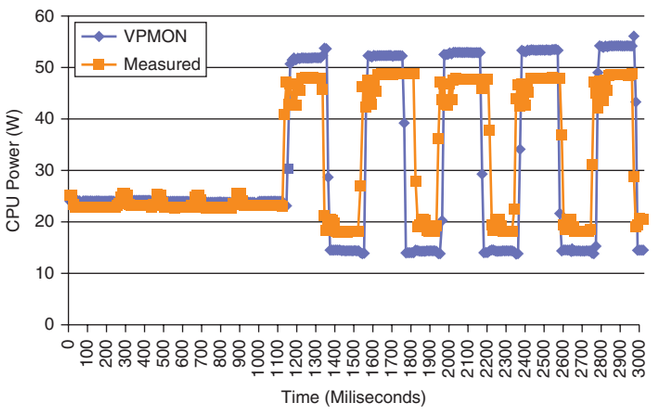

Figure 5: Power vs. time comparison for FFT DP workload
(Source: Intel Corporation, 2013)

It should be noted that for many usage scenarios such as peak power study for TDP analysis, the timing accuracy of power is less important. Instead it is crucial to have accurate peak power and its sustained duration in such use cases.

Simulating fine-grained instantaneous power through VPMON is limited by its overhead. The simulation speed slightly decreases with a finer sampling rate due to frequent calls to the power model. These periodic calls are achieved by a self-timed Simics event. To evaluate the efficiency of the entire VPMON framework, we measured its simulation speed for various workloads. We analyzed the simulation speed for the core compute loop of the DGEMM workload because it reported the highest number of instructions to process in each power sampling interval. This is directly proportional to the processing overhead of the multicore tracer in the VPMON framework. VPMON took 17000 seconds to simulate 1 second of real time. It processed 44 billion instructions in this duration and thus provided throughput of 2.58 million instructions per second (MIPS).

Summary and Conclusions

We showed that it is possible to build an accurate power modeling framework on Simics. The VPMON power module we discussed provided CPU power for realistic multicore/multithreaded workloads with more than 90 percent correlation when compared to measured hardware data. The simulation overhead we incurred from VPMON was minimal compared to the granularity of power modeling.

"Simulation overhead of VPMON is minimal compared to granularity of power modeling."

The power model was validated on existing hardware and can be extended to project the application power for future platforms. Once the whole flow of correlating Simics performance and power against existing hardware is completed, proper knowledge of design parameters (frequency, voltage) and architecture as well as process scaling (C_{dyn}) can be applied along with Simics functional models of future hardware to simulate future system power consumption. VPMON is currently portable within multiple Simics instances as well as different Intel Xeon CPU SKUs. Other system components such as external memory and GPU can be included for platform-level power modeling. VPMON can have a library of multiple power models in this case. However, it relies on functional models of system components available on Simics and thus its accuracy and efficiency is bounded by these factors.

Complete References

[1] Engblom, J., Aarno, D. and Werner, B., "Full-System Simulation from Embedded to High-Performance Systems", in Processor and System-on-Chip Simulation, Leupers, Rainer and Temam, Olivier (eds), pp. 25–45, Springer Verlag, 2010.

[2] Bertran, R. et al., "Decomposable and responsive power models for multicore processors using performance counters," in International Conference on Supercomputing (ICS), June 2010.

[3] Isci, C. and Martonosi, M., "Runtime Power Monitoring in High-End Processors: Methodology and Empirical Data," Proc. MICRO, December 2003.

[4] Wu, W. et al., "A systematic method for functional unit power estimation in microprocessors," in Design Automation Conference (DAC), July 2006.

[5] Bellosa, F. et al., "Event-Driven Energy Accounting for Dynamic Thermal Management," In Proceedings of the Workshop on Compilers and Operating Systems for Low Power (COLP), September 2003.

[6] Singh, K. et al., "Real time power estimation and thread scheduling via performance counters," ACM SIGARCH Computer Architecture News, November 2008.

[7] Sinha, A. et al., "Jouletrack: A web based tool for software energy profiling," in Design Automation Conference (DAC), June 2001.

[8] Varma, A. et al., "Accurate and fast system-level power modeling: An XScale-based case study," ACM Transactions on Embedded Computing Systems, April 2008.

[9] Chung, S.-W. et al., "Using On-Chip Event Counters For High-Resolution, Real-Time Temperature Measurement," in Thermal and Thermomechanical Phenomena in Electronics Systems, IEEE Computer Society, May 2006.

[10] Bartolini, A. et al., "A Virtual Platform Environment for Exploring Power, Thermal and Reliability Management Control Strategies in High-performance Multicores," in Proceedings of the 20th Great lakes symposium on VLSI (GLSVLSI) May 2010.

Author Biographies

Parth Malani is an engineer at Intel working on power and performance modeling for multi- and many core platforms. He holds a PhD and MS in Electrical and Computer Engineering from the State University of New York at Binghamton and a BE in Electronics and Communications from Gujarat University, India.

Mangesh Tamhankar has over 20 years of industrial and academic experience. He currently manages Intel teams working on software development, projections, and product development. He received his PhD in Computer Science and Reliability from the Indian Institute of Technology, Bombay, India.

SIMICS* ON SHARED COMPUTING CLUSTERS: THE PRACTICAL EXPERIENCE OF INTEGRATION AND SCALABILITY

Contributors

Grigory Rechistov
Systems Simulation Center, Intel Corporation

"…the task of running Simics distributed is not trivial; its challenges include maintaining simulation scalability, speed, and manageability."

Simulation of large computing systems is a challenging task, mainly because a single host may not be able to accommodate a full model. Therefore, a simulation itself has to be distributed across several systems. Simics provides such functionality with its individual parts communicating over a network transparently for target systems. Still, the task of running Simics distributed is not trivial; its challenges include maintaining simulation scalability, speed, and manageability. This article describes one practical case of simulating a large distributed cluster system with more than a thousand of target cores using Simics.

Introduction

This article describes our experience with creating and running a model of a large computing cluster system using Wind River Simics. Scale and resource requirements of workloads of this study made it necessary to run the simulation on top of a distributed multi-host system, resulting in a virtual computer cluster being simulated on a physical smaller cluster system. In the course of this work, we adapted Simics to be executed as a job of a cluster resource management application. In this article we present our instrumentation technique that was used to capture parallel application behavior. We present our observations of the simulation scalability that was reached and outline limitations we discovered during this study.

Applications, Target System, and Host Hardware

Applications that were run inside the target OS consisted of two packages for molecular dynamics, namely Gromacs[1] and Amber[2]. These applications are parallel and make use of the message passing interface (MPI) to communicate between processes. An MPI application's processes can be spread over multiple computing nodes, connected with a local high-speed network. Communication delays imposed by limited network bandwidth, nonzero latency, as well as other numerous factors—such as operating system buffering, specifics of network card configuration, suboptimal network topology, and so forth—can negatively affect overall performance. The described simulation was created to capture the whole system behavior related to MPI message delivery.

A system whose behavior was to be analyzed consisted of 112 identical multi-core nodes. Its configuration is shown in Table 1. All target nodes ran Debian GNU/Linux 6.0 x86_64.

Parameter	Value
Processor	Intel® Xeon® E5 (Sandy Bridge) 2.8 GHz
Number of cores per CPU	8 (16 logical with Intel® Hyper-Threading Technology)
Number of CPUs per node	2
Number of nodes	112
RAM per node	48 GB
Network configuration	Ethernet 10 Gbit/s
Total number of cores	1792
Total RAM	5376 GB

Table 1: Target system configuration
(Source: Intel Corporation, 2013)

The host system was a cluster itself, though of a smaller scale. Its configuration is outlined in Table 2.

"The host system was a cluster itself, though of a smaller scale."

Parameter	Value
Number of nodes	16
Processors	Intel® Xeon® 5580 (Westmere), 3.33 GHz
Number of cores per CPU	6 (12 logical with Intel® Hyper-Threading Technology)
Number of CPUs	2
Disk storage	3 TB
Network configuration	Infiniband* QDR 10 Gbit/s
RAM per node	32 GB
Total number of cores	192
Total RAM	512 GB

Table 2: Configuration of the host system
(Source: Intel Corporation, 2013)

The host system nodes ran the same version of Debian GNU/Linux 6.0 x86_64 as in the targets. It consisted of a single head node that served as an NFS server and several compute nodes that shared Simics installation and all required files on a network share. A network topology for both host and target

systems was a star (Figure 1). There were two separate networks, the first one (Gigabit Ethernet) was dedicated to service traffic (NFS, SSH, and so on) and the second, high speed Infiniband was used for applications traffic needs.

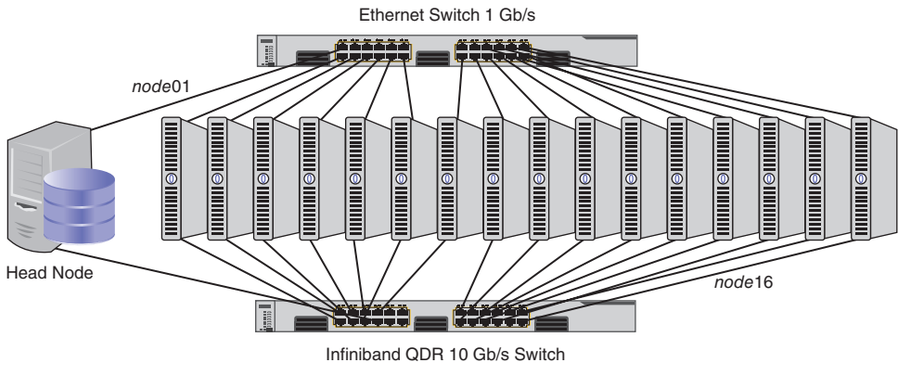

Figure 1: Host system interconnect configuration (Source: Intel Corporation, 2013)

Simics and Simulation Distribution

To utilize computing resources efficiently and speed up simulation runs, the simulation's parts should run in parallel, sometimes distributed over several nodes. This section describes capabilities Simics offers to achieve this goal.

Parallel Simulation on a Single Host

Simics allows modeling of several target systems in a single simulation run. Each of these targets can be run in a separate host thread. Effectively this allows the utilization of more computational resources of a host system and speeds up the simulation.

"For the described scheme to be possible within Simics, several conditions must be met"

Still, not every existing modeling scenario of a multi-core system can be arbitrarily partitioned onto multiple host threads. For the described scheme to be possible within Simics, several conditions must be met:

- All Simics device models that constitute a simulation must be marked as thread-safe. That is, some care should be taken when writing Simics modules if they are supposed to be used in multithreaded environments. When loading a new module, Simics checks it to be marked as thread-safe and disables the feature globally if it is not. The majority of Simics modules are already made thread-safe thus this is rarely a concern.
- Parts of a simulation that are to be run in separate threads must be loosely coupled; that is, the frequency of communication between them should be significantly lower than average frequency of their internal communication. For this study, this meant that an SMP system had to be simulated in one thread, and it could contact other simulated SMP domains run in different threads via simulated network. This is because LAN messages can

be delivered significantly less frequently than shared memory messages. This limitation on what can and what cannot be parallelized is dictated by tradeoffs between performance, determinism, and complexity of underlying Simics implementation.

For the scope of this study all conditions for multithreaded simulation were met: each host with 12 logical processors was able to simulate up to 12 target machines (nodes) in parallel. This means that each host processor core was responsible for simulating 32 logical target cores (2 processors, each with 8 cores and 16 threads).

Distribution between Multiple Hosts

To make even larger simulations possible, Simics supports running a simulation in a distributed mode, when its loosely coupled parts can be spread across several host machines. Corresponding simulation partitions (called *domains*) periodically synchronize over the host network transparently to the simulation. To provide a deterministic behavior a barrier scheme is used (Figure 2).

"To make even larger simulations possible, Simics supports running a simulation in a distributed mode…"

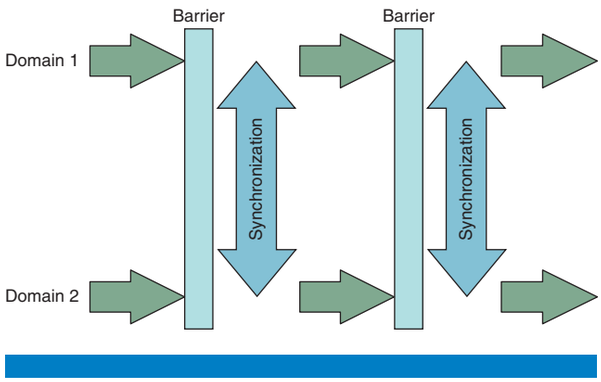

Figure 2: Domain synchronization scheme (Source: Intel Corporation, 2013)

In this scheme, each domain is running its own part of the simulation for a predefined amount of simulated time (called *quota*) without any communication with other domains. Then barrier synchronization is used, at which point all pending inter-domain messages are delivered.

For the distributed simulation to work, TCP/IP sockets are used. Each participating Simics process should be configured to use the same host:port pair, which indicates where a top-level synchronization domain is executing.

To make a clear view of placement and interaction between all parts, the whole setup of simulation is shown in Figure 3. Each host's logical processor core is used to serve for one target machine with all its simulated cores. To enable transparent interaction of target machines placed on different host systems (and also for global simulator time synchronization) the host local network is used to encapsulate and transfer packets of simulated network, which is also isolated from host LAN to prevent nondeterministic influences of the real world.

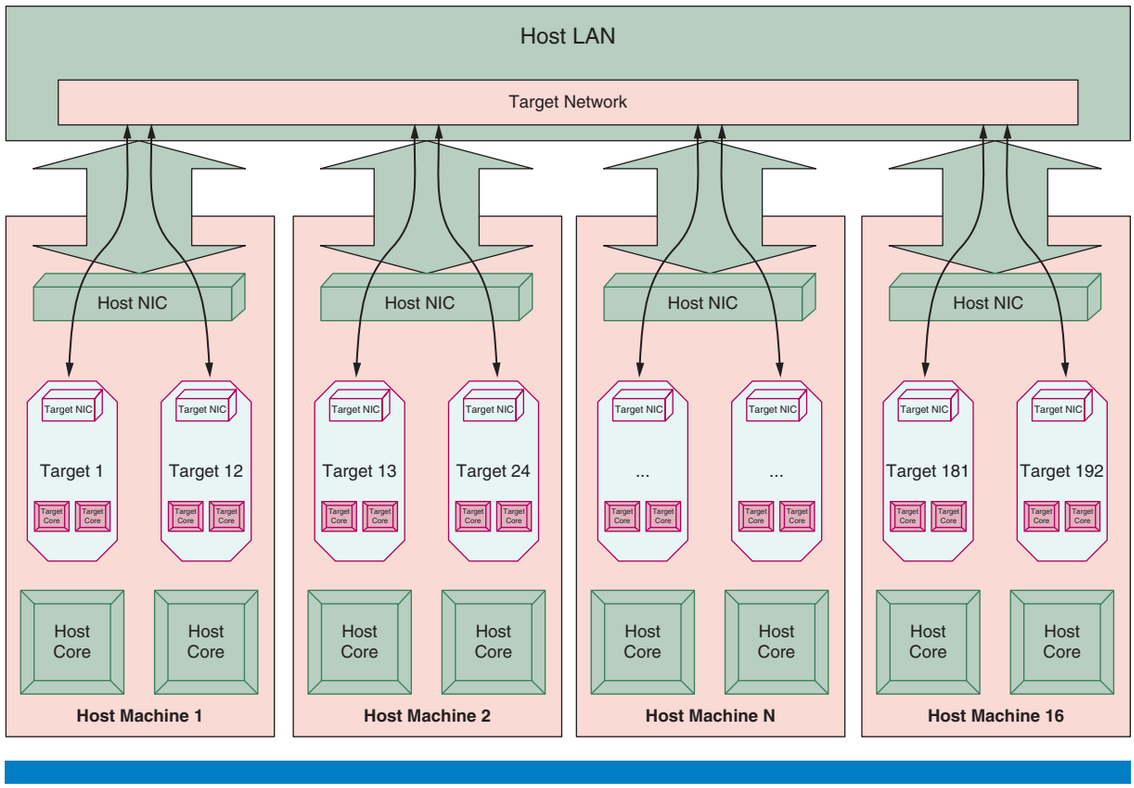

Figure 3: Distributed simulation setup
(Source: Intel Corporation, 2013)

Dynamic Node Allocation and Process Distribution

The original Simics scheme for a distributed simulation makes an assumption that it is known in advance before launch which hosts will be used to run Simics instances and which network host:port pair will be used for the centralized coordination. Additionally, it is a user's task to log on to each of participating hosts and start a new Simics process on it. If the number of participating hosts is high, this task becomes very tedious. What's more important, in our case not all host nodes were immediately available all the time — some of them may have been exclusively occupied with tasks run by other users, some could have been turned off for maintenance, and so on. It was therefore required to implement a method to free a user from these tasks by creating a mechanism that allocates all required resources.

"It was therefore required to implement a method to free a user from these tasks by creating a mechanism that allocates all required resources."

On the described computing cluster, Simple Linux Resource Manager (SLURM)[3] was used as a system-wide resource manager to be used by all working groups. In order to be able to open an SSH shell to a node, one was required to request SLURM for an exclusive allocation of the necessary number of nodes for a predetermined period of time first. If there were no free nodes at that moment, the request would be blocked until one or more existing allocations had finished or exhausted their time quota, thus releasing more resources.

Various SLURM tasks, such as allocating, freeing, querying nodes can be accomplished with command line tools such as *salloc*, *smap*, and *srun*. It is also possible to use API bindings existing for many programming languages, including Python. Therefore we implemented a set of Python scripts to tie Simics and SLURM. The overall high-level overview of the distribution processes is illustrated in Figure 4.

"…we implemented a set of Python scripts to tie Simics and SLURM."

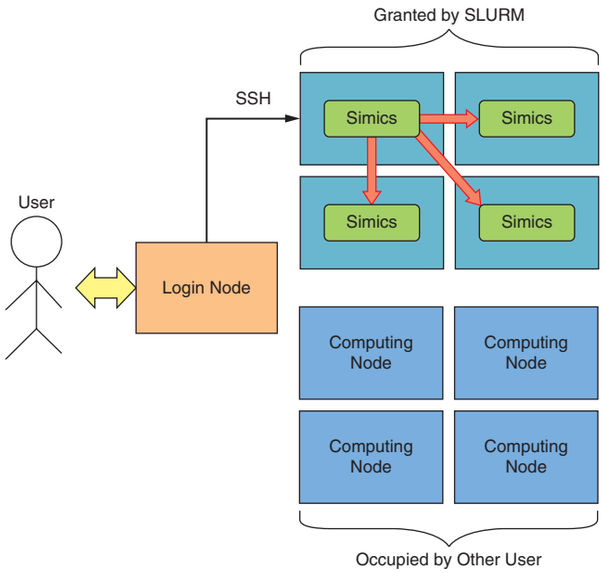

Figure 4: Dynamic simulation distribution sequence (Source: Intel Corporation. 2013)

A wrapper to the *simics* program, called *simics-slurm*, was written to automate tasks of host resources allocation and simulation distribution. This script is executed on a login node by the user. It accepts all regular Simics command-line arguments, such as a target script name, but additionally performs the following steps:

- *simics-slurm* asks the SLURM service to allocate the desired number of host nodes.
- SLURM checks if enough cluster nodes are unoccupied. If not, it stalls until some of already active jobs finish and release enough resources. It then allocates nodes for exclusive use and passes a list of their host names back to the *simics-slurm*. The list of nodes is then fixed and they are guaranteed to be free from other users' tasks until either nodes are returned to the free pool or the granted time period runs out.
- The script then opens an SSH session to the first host node granted by SLURM and spawns a master Simics process on it. This step is done to relieve the original login node from running resource-intensive programs as it only serves as an entry point to all users of the cluster and is not supposed to host their tasks.

- The master Simics process runs the *global_distrib.py* script, which spawns additional Simics slave processes through SSH on remaining allocated nodes with relevant command-line arguments. It also chooses a random TCP port number to be used for domain synchronization. Finally, it creates a target machine called *master0* that will serve as a head node of the simulated cluster. It served as an NFS server inside the simulation.
- Each of the Simics slave processes spawns one or more target machines, each of which is given an simulation-unique name of *node*N, where *N* varies from 001 to 112, and connects to a given host:port. All information necessary to establish connections is passed via SSH command-line arguments.

Global Commands Implementation

For a simulation confined to a single (possibly multi-threaded) host process, Simics offers a rich set of commands to control it. Commands exist to wait for an event, to delay a script for a specified time interval, to set a break point at instruction address or data, to stop simulation at target console string, to start or stop simulation, and so on. Unfortunately, there are no similar convenient methods to do it within multiple Simics processes. For example, to start a distributed simulation one has to issue a "continue" command at every Simics control console manually. This certainly contradicted with our goal to have automatic simulation runs.

"To have minimal required control over the distributed simulation we needed the ability to pass messages between separate Simics instances…"

To have minimal required control over the distributed simulation we needed the ability to pass messages between separate Simics instances and to write handlers for them. It turned out that some form for it is already present. There is an undocumented Simics function VT_global_message () that can pass an arbitrary text string that will be caught by a user-defined hap handler. These messages are always global, that is, broadcast to all processes, therefore for peer-to-peer communications every Simics instance had to filter out messages not targeted to that particular process.

Using this technique several global commands were implemented to start, stop, and exit simulation, to attach and detach the mpi-tracer tool (described later) and to run an arbitrary Simics command inside all processes. The global message was also used to inform the master script that secondary nodes had reached certain states in simulation.

Experiment Flow

The combination of Simics distribution scripts and the implementation of global commands allowed us to organize our experiments as described below:

- The master script spawns all slave Simics processes and waits until all of them report "ready" through sending of global messages.
- A global start of simulation command is sent back from the master process to the rest of Simics instances. All target machines start to run. It should be noted that a master target is allowed to pass GRUB bootloader stage earlier than remaining targets because it has to bring up an NFS server in advance so that it can be used inside the simulation.

- A script branch is created to wait for all simulated machines to report that they have booted.
- After a target machine has reached the Linux login prompt, a login/password pair is entered through a keyboard model. Then a Simics process that contains that machine sends a global message to notify the master process that one more target machine is ready.
- After all machines have booted, the keyboard model on the master target node is used to enter necessary commands to start an MPI application.
- After a predefined delay that is meant to allow the application to start up, a global command is issued that instructs all Simics instances to activate *mpi-tracer* to start recording all MPI activity.
- After another predefined interval of simulated time, another global command is broadcast to stop capturing MPI calls trace and to save already obtained results to disk. Shortly after that, a global shutdown command is sent for all Simics copies to quit.

The Study: Capturing MPI Calls Trace

"The ultimate goal of this study was to capture all calls to MPI communication functions that a distributed application issues…"

The ultimate goal of this study was to capture all calls to MPI communication functions that a distributed application issues during its execution on a large number of cores distributed across multiple nodes. The MPI standard[4] defines several dozens of functions to be used to create point-to-point and broadcast messages as well as a small number of auxiliary functions for local data manipulation.

We used a modified MPI library to catch every entry and exit from a set of MPI communication calls (we omitted tracing of auxiliary functions because they are not related to communications). To be able to do it we inserted a machine instruction that is ignored by a target application but is specially treated by Simics — a so called "magic instruction." For IA-32 targets it is CPUID with a nontypical leaf number. After Simics detects the magic instruction, execution is passed to the user-specified function handler, which inspects the machine state and logs a predetermined set of data: name of MPI function, contents of its arguments taken from the stack and value of simulated time. While this function is running, the simulation is standing still, and the state of the system is consistent. The target inspection takes zero time from the perspective of the target, happening between two target instructions. We modified a magic instruction macro definition shipped with Simics so that more than just two integers can be communicated between target system and simulator in a single magic call, as we needed to save up to 16 values containing MPI function number and values of all its arguments to be recorded in a trace. The overall control flow for handling an MPI call is shown in Figure 5.

It should be noted that a target application was not aware of any of the described activity—for it, the executing of CPUID was just as if it were completing a regular instruction.

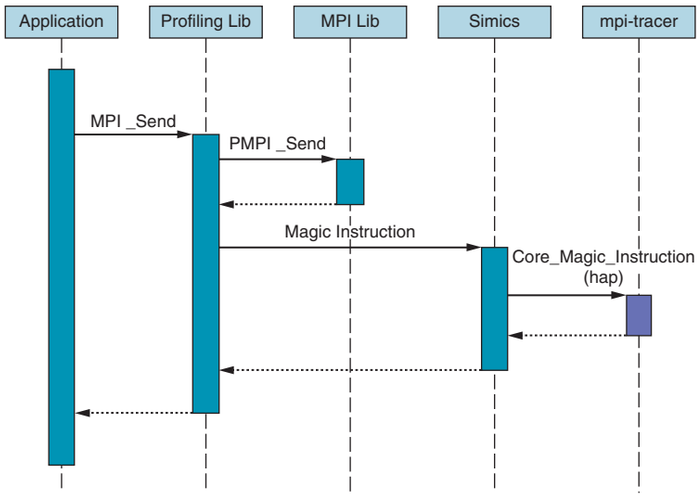

Figure 5: MPI tracing sequence
(Source: Intel Corporation, 2013)

"The analysis of collected data was done offline with a second group of Python scripts…"

The analysis of collected data was done offline with a second group of Python scripts that was created to extract useful information, such as MPI call frequency, distribution, and length, from collected binary traces. As an example of results that could be obtained with the system created, a characterization of occurrences distribution for all MPI functions observed for *mdrun* (the most important application of the Gromacs suite) is shown in Figure 6. The number of target machines participating in this series of experiments varied from 2 through 64.

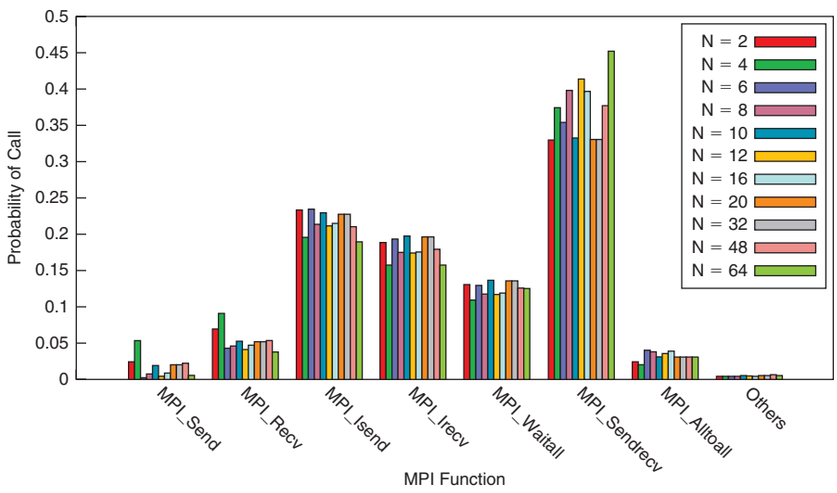

Figure 6: Probabilities of different MPI functions calls for Gromacs (application mdrun) for different simulation sizes. *N* is a number of simulated nodes, each of which contained 16 processes.
(Source: Intel Corporation, 2013)

From the data and additional statistics collected, a conclusion was made that this application spent most of its communication time in two data exchange calls, namely peer-to-peer *MPI_Sendrecv* and collective *MPI_waitall*. Further analysis demonstrated that another frequent pair of routines, *MPI_Isend* and *MPI_Irecv*, actually does not introduce significant delay into the application's operation. Therefore, in order to optimize a cluster configuration to this application, a system designer's attention should be focused just on optimizing for *MPI_Sendrecv* and, to lesser extent, for *MPI_waitall*.

Scalability Results

In this section we describe the largest simulation that we were able to carry out and what it took in terms of time and space.

We were able to simulate up to 1792 cores, which constitute the target system of interest, on about 150 physical cores. For such a large simulation 12 host cluster nodes had to be exclusively allocated for Simics. The collection of MPI traces for 200 simulated seconds of an application run took about two days of simulation. A single trace file from each host node contained about 2 GB of binary data, totaling more than 20 GB of logs for the whole simulation. To extract MPI call profile data from the raw logs, the typical processing time was about one hour.

"We were able to simulate up to 1792 cores, which constitute the target system of interest, on about 150 physical cores."

Slowdown

An initial booting process consisted of several phases: SeaBIOS boot, GRUB bootloader, Linux kernel, and userland startup up to the shell login. A slowdown for this part of simulation varied from 20 to 50 times. The value was relatively low because this part of simulation was executed with Simics VMP mode enabled.

For the MPI tracing part of the simulation, the observed slowdown varied from 800 to 2000. At this phase simulation was often (at every MPI call) interrupted to process a magic instruction callback. As a result a lot of data was dumped on disk during this phase, adding to resulting slowdown. Also, VMP was mostly turned off.

Scalability Limitations

The main concern and limiting factor of this study were the memory requirements of target applications. Each booted target machine required about 2 GB of host memory to work. This limited the number of target nodes on a single host node to 14. This limitation is hard to circumvent as the memory requirements are basically noncompressible. A swap might help in this case and Simics supports automatic offloading of its images to swap files; still it is possible that it would result in a catastrophic simulation slowdown.

A low simulation speed may also be a critical factor for an experiment in a case when its full execution time is approaching the MTBF (mean time

between failures) value of the host hardware. A process that takes too long to complete will be interrupted by a hardware/software failure with a high probability. A possible mitigation of this is using simulation state checkpointing.

Conclusions

"…Simics is capable of simulating thousands of processor cores distributed across hundreds of target systems…"

In this article Simics' ability to handle large multi-machine scenarios was demonstrated. When resources of a single host are not enough, simulation can be distributed across several systems. It was shown that Simics is capable of simulating thousands of processor cores distributed across hundreds of target systems connected within a network, and such a large simulation can be carried out with one tenth the physical resources while maintaining acceptable slowdown.

To efficiently share computational resources of a host system and ease experiment setup and automation, Simics was adapted to be run as a SLURM job.

Finally, a method to capture behavior of a parallel application based on mechanism of magic instructions was described. This method was used to study behavior of two applications that were run on top of distributed computing cluster simulation.

Acknowledgments

The author would like to thank the following people who participated in the joint MIPT-Intel research and helped to develop tools and methodology described in the article: Evgeny Yulyugin, Artem Abdukhalikov, and Pavel Shishpor.

References

[1] D. Van Der Spoel, E. Lindahl, B. Hesset et al., "GROMACS: Fast, Flexible, and Free," *Journal of Computational Chemistry*. 2005. V. 26. No. 16. Pp. 1701–1718.

[2] D. A. Case, T. A. Darden, T. E. Cheatham, et al., Amber 11 Users' Manual, University of California, 2010.

[3] Jette M. A., A. B. Yoo, and M. Grondona, "SLURM: Simple Linux Resource Management System," In *Lecture Notes in Computer Science: Proceedings of Job Scheduling Strategies for Parallel Processing* (JSSPP), 2003.

[4] Message Passing Interface Forum. MPI: A Message-Passing Interface Standard. Version 2.2, 2009.

Author Biographies

Grigory Rechistov is a software engineer in the Systems Simulation Center at Intel. He joined Intel in 2007 and his career has since been devoted to creating software models of upcoming Intel CPUs. He holds a BS and MS in applied mathematics and physics from the Moscow Institute of Physics and Technology and recently defended his PhD thesis in computer science. His email is grigory.rechistov at intel.com.

DEVICE DRIVER SYNTHESIS

Contributors

Mona Vij
Intel Labs, Intel Corporation

John Keys
Intel Labs, Intel Corporation

Arun Raghunath
Intel Labs, Intel Corporation

Scott Hahn
Intel Labs, Intel Corporation

Vincent Zimmer
Software and Solutions Group, Intel Corporation

Leonid Ryzhyk
University of Toronto

Adam Walker
NICTA

Alexander Legg
NICTA

"Device drivers are the major cause of operating system failures"

Automatic Device Driver Synthesis is a research collaboration project between Intel and National Information Communications Technology Australia (NICTA) that aims to synthesize device drivers automatically using formal OS and device specifications. We have built a tool chain that uses Simics* DML Device model sources as an input to the driver synthesis tool chain. The tool chain has a frontend compiler that extracts the device behavior from the Device Modeling Language (DML) model and outputs a formal representation of the device behavior that we refer to as a device specification. The driver synthesis tool combines this specification with a similar O/S specification and applies the principles of game theory to compute a winning strategy on behalf of the driver and eventually converts it into driver C code. This approach aims to use the existing device models for producing device drivers resulting in highly reliable drivers and faster time to market. We have synthesized a number of drivers using our tool chain. Some examples include legacy IDE controller, UART, SDHCI controller, and a minimal Ethernet adapter.

Introduction

A device driver is the part of the operating system (OS) that is responsible for controlling an input/output (I/O) device. There is wealth of research[1][2] showing that drivers are a primary source of bugs, and driver development is a major bottleneck for platform validation and time to market. Figure 1 shows the conventional driver development process, where a driver writer uses two informal documents, OS and device specifications, to convert a series of OS requests to device commands. The process of device driver creation can be error prone and tedious. One of the main reasons is that the driver writer uses informal documents that are susceptible to misinterpretation. In addition, the driver writer has to have domain knowledge of both the OS and the device. In many cases driver writers also reuse existing driver code to write a new driver, inheriting any existing bugs in the process.

We propose to improve the driver development process by automatically synthesizing drivers from formal OS and device specifications, as shown in Figure 2. This is based on the fact that all the information needed to control a device from software is available during the design of the device. The idea is to represent this knowledge, so as to enable synthesizing driver automatically.

For device formal specification, we plan to leverage the high-level device models either written by hardware designers or for software simulation for virtual platforms. We are building a tool chain that applies the principles of

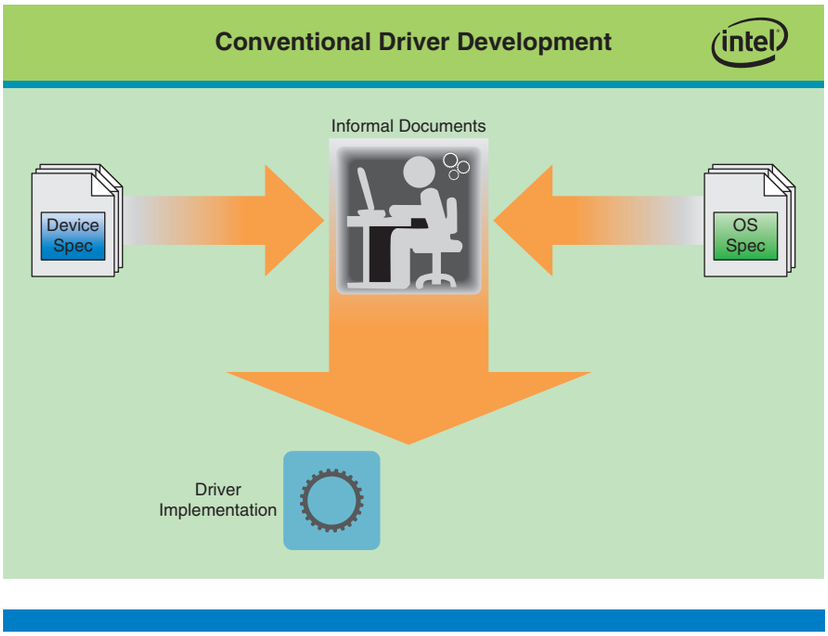

Figure 1: Conventional driver development
(Source: Intel Corporation, 2011)

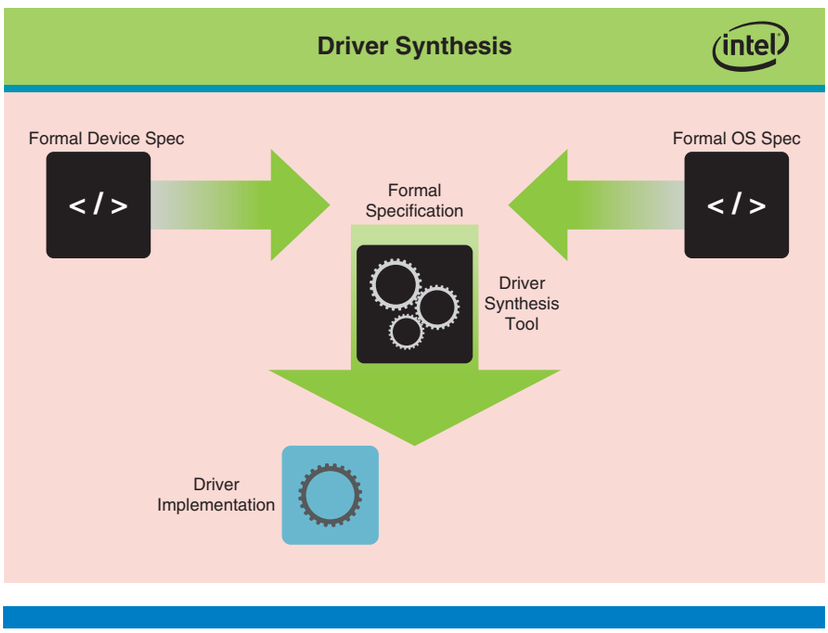

Figure 2: Driver synthesis
(Source: Intel Corporation, 2011)

"Driver synthesis from formal specifications"

"Game theory in driver development"

game theory and synthesizes the driver code from formal specifications. This approach improves driver reliability by reducing manual intervention, avoiding misinterpretation of device documents by driver writers. Moreover, given a device specification, drivers can be generated automatically for all supported operating systems, thereby eliminating the costs associated with porting drivers. With this approach of driver development, DML device models are used not only for simulation, but for driver generation as well. The driver synthesis tool chain also provides some additional capabilities like a state space explorer that aids in DML device model debugging. Overall this approach results in correct drivers and improves time to market by moving development earlier in design cycle, leading to cost reduction.

In the long run we plan to support large classes of devices with this tool, from very simple to complex devices, as long as their behavior can be represented as a state machine. We can't synthesize drivers that perform complex computation and are difficult to represent as a state machine. In addition, we don't plan to support drivers for devices that are based on programmable cores, such as high-end graphics or network processors.

High Level Architecture

Device driver synthesis aims to create device driver code automatically from hardware specifications of a device. Figure 3 shows various components in the driver synthesis tool chain that begins with formal specifications and converts it to various intermediate forms before finally emitting the device driver code. We formalize the driver synthesis problem as a game between the driver and its environment, which consists of the device, additional device interfaces (for example, network) and the operating system. The formal specification of the device and OS interface, together, define the "rules" of a two-player zero-sum

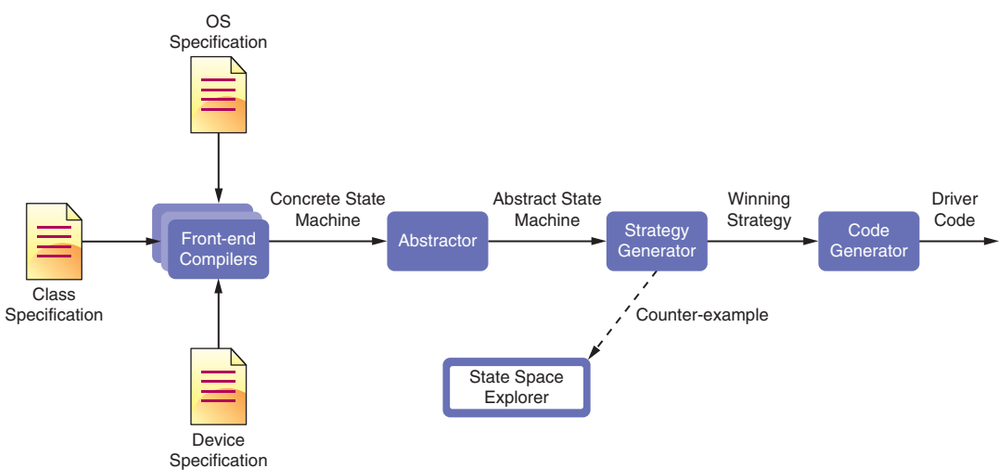

Figure 3: Driver synthesis tool chain
(Source: Intel Corporation, 2013)

game. The driver assumes the role of the first player and the environment (OS, media, and so on) describe the moves of the "opponent." In the context of the game, modeling the environment as an "opponent" puts more emphasis on the environmental events that lead to failure than those that are benign. The environment begins all games with moves that represent OS-to-driver requests. In response to these moves, the driver must try and make "moves" (that is, send commands to the device) to push the device to a winning state, corresponding to a correct device response for the given OS request. The moves chosen by the driver should be such that no matter what external event occurs, the device and driver can either correctly service the OS request or fail gracefully and continue to operate correctly in the future. Effectively the tool constructs a driver algorithm that guarantees that the driver is able to correctly satisfy all OS requests given any feasible driver behavior. We call such an algorithm a *winning strategy* on behalf of the driver.

Tool Inputs

The tool takes multiple formal specifications as input, as described in the following subsections.

Device Class Specification

The Device Class specification models states, events, and functions common to all devices of a given class in an OS-independent and device-implementation–independent manner. The specification describes events that represent interactions between the device and its environment (that is, connected media, external devices, and so on). Events may also represent completion of individual device requests such as setting a configuration. The states describe logical device states applicable to devices of class, such as configured states, initial state, and so on. In addition, the specification may describe sub-states that a device is expected to transition through in order to complete a device function. In addition, the specification also defines all constant values given to or received from devices of class, such as baud rates, configuration values, and I/O signals.

"Device class specification models states, events and functions common to all devices in a class"

Device Class specifications need only be written once per device class and can be used with different OS specifications and devices of the same class from different vendors. We believe a model similar to USB's Device Working Group (DWG) would work best for establishing industry-wide device class specifications. In this model, classes of devices are identified and a working group (WG) is established for each class, drawing WG membership from interested parties who tend to be the leaders and experts in a specific device class. The WG then develops a class specification by consensus, with the result typically being subject to approval of the parent organization.

OS Interface Specification

The OS Interface specification describes legal sequences of interactions between the driver and the OS as well as the expected device response on completion of each OS request. It models when events defined in the device-class specification must be raised in response to OS requests. This specification does not specify how the events in the device-class specification are generated, since that should

be part of the device specification. It is up to the synthesis algorithm to derive the necessary steps for generating these events in response to OS requests.

Ideally, the OS specification for a specific OS will be produced by the entity that produces the OS. This specification needs to be written once per OS per device class and when a new OS release occurs; minimal change should be required to adapt the specification.

Device Specification

Device specifications are device-specific instantiations of device class specifications. They model the device behavior and the externally visible artifacts of the device. In particular, they model externally visible registers and device operations that result from the reading or writing of said registers. The device response depends on the register values and device internal state, such as, for example, whether the device is initialized or waiting for a request to complete. These responses include but are not limited to updating register values, generating interrupts, triggering one or more external events, and interactions with other subsystems. These specifications are written at a high level of abstraction and ignore detailed internal architecture and timing.

Individual device specifications must be produced by the device vendor. In the case of industry-standard devices such as EHCI and XHCI (USB) and SDHCI (MMC/SDIO), a single device specification can be produced by the entity responsible for the standard and used for any device that meets the standard. In the case where a device is industry standard but also contains vendor-specific extensions, the device vendor becomes the responsible party. The vendor can import the industry-standard specification to specify device core functionality, but still remains responsible for specifying the vendor extensions.

Tool Outputs

The tool processes the input specifications and applies the principles of game theory to produce driver code.

"The tool produces "C" code as output if it finds a successful strategy"

Driver Code

The tool produces C code when it finds a successful strategy. In some cases driver writers will need to develop manual wrappers to integrate the code with the OS.

No single entity can be identified as the entity responsible for producing device driver binaries. Industry history suggests three potential sources: OS vendor, hardware vendor, and platform integrator. OS vendors generate large numbers of device drivers, tied to OS release cycles. Hardware vendors produce drivers when 1) the target OS vendor does not support the device (in particular for new hardware), and 2) when the need for the driver falls between OS release cycles. Platform integrators generate device drivers when the driver is not provided by the OS vendor or the device vendor, or they built the device themselves.

Table 1 illustrates the interdependence between the three entities

Entity	Produces	Consumes
Device Class WG	Device Class Specification	n/a
OS Vendor	OS Specification	Device Class Specification
Device Vendor	Device Specification	Device Class Specification
Platform Integrator	n/a	Device Class Specification OS Specification Device Specification

Table 1: Specification Producers and Consumers
(Source: Intel Corporation, 2013)

DML Models for Driver Synthesis

Device Driver synthesis aims to synthesize drivers automatically from formal specifications, so availability of a device specification is a key to success of the tool. If a device specification has to be created specifically for synthesis, then we've only accomplished the shifting of efforts from driver development to specification development, rather than solving the problem. In addition there is no way to validate the manually developed model to make sure that it models the device operation properly.

"Availability of a device model is key to the success of the tool"

There are many high-level device specification languages that are currently used by hardware manufacturers including SystemC, System Verilog, and Simics DML. To ensure that the driver synthesis tools are widely applicable, the architecture provides for multiple frontend compilers that convert specifications written in a given language into an intermediate language Termite Specification Language (TSL) developed by us. TSL provides a means for concise description of FSM states and transitions and is used as the FSM external representation by all other tool-chain components.

Wind River Simics* is becoming the platform of choice for virtual platforms at Intel. Many DML models already exist and are being used successfully in virtual platforms. If a particular DML model doesn't exist, then writing the model contributes to synthesis as well as virtual platforms. We have developed a frontend compiler for DML for using DML models with our tool chain.

DML to TSL Compiler

DML has been designed to facilitate fast model development by software engineers. It is a very forgiving language in general, allowing forward referencing, type casting, and automatic C-style type promotion. TSL, on the other hand, is very restrictive. For example, it does not provide type promotion or casting or allow forward references.

"DML compiler extracts the relevant device behavior from DML device models."

One of the goals of the project is to not modify the actual device models, since we do not want our use of the models to impact their original use in virtual platforms and we do not want to force a fork of the models, which might lead to issues with bug-fix propagation. We have built a DML compiler that tries to deal with the DML to TSL conversion automatically, but in some cases we do need to modify the model. Currently we do modify the model directly, but all of the modifications we currently make to the actual model could instead be kept in a separate annotations file, thereby leaving the model pristine. This support will be added in the future versions of the tool.

Extracting Device Behavior from DML Models

Conceptually, DML architecture is very similar to event-driven GUI architectures. A DML model can be thought of as a collection of responses, where each response corresponds to a message or a set of inputs. Responses execute instantaneously; that is, simulation time does not advance while an individual response is executing, and blocking in a handler is prohibited. Response execution always begins with an external call of an interface method and completes with the return to the external caller.

TSL models express device behaviors as a collection of variables that represent device state and a collection of transitions to these state variables. Given a set of input state changes, each individual transition describes the cascade of changes to other state variables in response to the input changes. In addition, each transition may have guarding constraints that allow it to be enabled or disabled depending on current device state. Similarly to DML, TSL transitions are also instantaneous. While they resemble code, a TSL transition can also be thought of as a formula that computes next state S' given current state S and inputs I: $S' = f\text{Trans}(S, I)$.

Conceptually, DML model structure closely corresponds to the TSL structure. A single TSL transition maps directly to an execution trace of a DML interface method and its called methods. The TSL state variables map directly to the collection of DML registers, fields, attribute objects, and data objects.

Before we can begin extraction, we build out an in-memory representation of the model. This involves application of templates to DML objects, evaluation of parameters, expansion of *select* and *foreach* keywords, and evaluation and pruning/expansion of *if object* statements. Each of these steps can result in significant model changes so evaluation of the model really cannot be performed without these steps.

We begin the extraction process by collecting the model variables that will become the TSL state variables. All data objects and attributes are added to the collection as they are encountered. Fields are added only if their *alloc* parameter is *true* (that is, model space is allocated for its contents). Registers are added only if they do not contain fields and their *alloc* parameter is *true*.

We identify the individual transitions to be extracted (transition entry points) by identifying and collecting all exported interface methods contained in the models. As well as explicit interfaces, this set also contains the read/write bank

access methods for all register banks present. We also add transitions for each DML *event* object and *after* keyword encountered in the model, along with a 1-bit guard variable for each event or after transition.

After identifying the entry points, we can begin extraction of the transitions. This is done by first copying the method containing the entry point, then replacing each *call* or *inline* statement with the body of the target method. This is repeated recursively until no *call* or *inline* statements remain and we are left with a full code trace through all branches of the call. As an optimization, we concurrently evaluate *if* statement conditions to prune branches that will never be taken because they will always be false.

Besides state variables, TSL allows for temporary variables. These are global in scope but do not retain values across transitions. TSL has no notion of transition-local variables. As part of the transition extraction, we must convert all local variables found in DML methods to TSL temporary variables. Because of TSL's global scoping, some amount of variable name mangling is required to ensure unique variable names.

TSL restricts transitions from modifying a variable more than once per transition. This requires us to analyze each extracted transition and introduce new temporary variables and assignments when violations are identified.

TSL also requires that any single transition must update all state variables. To meet this requirement, we analyze each branch in the transition for assignment statements. For each variable assigned, we add an identity assignment (state' = state;) to the corresponding branch. We complete this requirement by adding identity assignments to the end of the transition for all remaining unassigned state variables.

The following subsections describe how our frontend DML compiler deals with the conversion from DML to TSL.

DML Templates

Development of the compiler caused us to study several of the import files in great detail, specifically dml-builtins.dml and utility.dml, leading us to realize the power of well-planned template and parameter use. This in turn allowed us to write "extensions" in DML itself, rather than extending the language.

"The tool converts models into an intermediate representation called TSL that is amenable for analysis and synthesis"

The file dml-builtins.dml provides the glue that ties banks, registers, and fields together, as well as providing default methods and parameters for most types of DML objects. Unfortunately, it is so closely tied to the Simics DML compiler, dmlc, that we could not use it without porting it. Our first porting task was to create our own versions of the methods that are "intercepted by the DML compiler." These methods are involved in the read/write access fan-out from bank objects to registers and fields.

For Simics device I/O, the bank method access() serves as the primary entry point for the I/O-memory interface (register read/write operations). Instead of a single method that takes direction and size as parameters, TSL uses a set

"Built-in templates for register access provide a software interface to the device internals with appropriate constraints"

of entry points: read8(), write8(), read16(), write16(), read32(), and write32(). To accomplish this change, we modified the behavior of our bank objects to create parameters containing lists of mapped registers of specific sizes: mapped_regs8, mapped_regs16, and mapped_regs32. We also defined an iioregion interface with methods corresponding to the TSL requirements and modified the default "bank" template in our dml-builtins.dml file to implement the iioregion interface and instantiate the individual access methods as applicable. In addition, we added the ability to turn off access for banks we were not interested in. For instance, we may be working with a PCI-based UART where we are interested in the UART register banks but not the PCI configuration space register banks. This control allows us to extract UART register-related transitions while ignoring PCI-configuration related ones.

Early on, we discovered that our game-playing solver did not always follow the rules that driver writers do. Specifically, it would attempt device register access before the driver's probe() routine had been called. To solve this issue, we added a guarding constraint to the access methods, blocking them until probe() had been called. The following is a portion of our dml-builtins file illustrating these changes:

```
// io_waits_for_probe – define to block IOs before probe() is called
parameter io_waits_for_probe  default undefined;
// conditionally create a variable to track if probe() has been called
if (defined $dev.io_waits_for_probe) {
  data uint1 probe_called;
}

template bank {
        .
        .
        .
        // extensions for tsl
        parameter mapped_regs32          default undefined;
        parameter mapped_regs16          default undefined;
        parameter mapped_regs8           default undefined;

        // controls if bank-related transitions will be emitted
        parameter emit_accessors         default true;

        if ($this.emit_accessors == true) {

                // not emitted if bank has no visible registers
                if (defined $this.mapped_registers)  {

                        // The TSL access interface
                        implement iioregion {
                                // Does bank contain mapped 8-bit registers?
                                if (defined $parent.mapped_regs8)  {
```

```
        // emit guard if we need to wait for probe
        if (defined $dev.io_waits_for_probe) {
          parameter guard_read8 = ($dev.probe_called == 1);
        }
        // and emit the read access method
        method read8(uint32 roffs8 ) -> (uint1 rstatus, uint8 rval8)  {
```

Event objects presented another challenge. In Simics, execution of an event object's event() method is constrained by its posted state. It can only be called if it has previously been posted to an event queue. In TSL, no such queues exist. This is further compounded by the almost 100-percent rate of models overloading the default event() method. We needed to constrain the event() method to only run when posted, and we needed to retain control of the event's entry point so we could apply the constraint and perform constraint housekeeping. Again, we were able to perform the bulk of this work by modifying the default event template:

"Event handling is challenging in TSL, as there are no queues. "

```
template event {
    .
    .
    .
    // variable to track posted state
    data uint1 _posted_;

    // methods to manipulate posted state
    method post(when, data) {$this._posted_ = 1;}
    method remove(data) {$this._posted_ = 0;}
    method _cancel_all() {$this._posted_ = 0;}

    // instantiate an event "wrapper" entrypoint
    implement event_entry  {
        //entry point only enabled when event is posted
        parameter guard_pre_event = ($_posted_ != 0);

        method pre_event(void *param)  {
            // housekeeping – reset posted state
            $_posted_ = 0; //Clear posted flag and call event
            // call control to real handler
            inline $parent.event(param);
        }
```

Unused Code

There is some code in DML device models that is for DML infrastructure and not for device operations. Our tool has no need for such code and we needed a way to eliminate such code from models without modifying the models. We have defined a few annotations for use in the models. They all begin with the sequence //@ and so are transparent to the Simics DML compiler. We use the pair *//@ignore* and *//@resume* to hide portions of DML from our DML tool.

"The TSL compiler performs strict type checking requiring the DML compiler to coalesce types by rewriting expressions in the emitted TSL"

We have used these to some extent in the models but mostly use them in our copies of the system import files, the DML equivalent of user/include/*.h.

Width Conversion

TSL does not support type promotion or casting, so our DML compiler performs a significant amount of expression rewriting in order to provide explicit width conversions. Width conversion to a wider type requires the original assignment be converted to a conjunction of two assignments, the original assignment and a second assignment to the extra bits. For example, assuming a 32-bit variable named *foo* and a 16-bit variable name *bar*, the statement:

```
foo = bar;
```

becomes:

```
((foo[15:0] = bar) && (foo[31:16] = 0))
```

In some cases, the format of a DML expression may prevent our tool from being able to make this modification. For instance, the DML expression:

```
foo = (somevar == 0) ? bar :0;
```

cannot be modified because the conversion is only needed conditionally but can only be expressed in terms of the global *foo*, not the conditional *bar*. In these cases, we rewrite the DML in a form that allows for the conversion:

```
if (somevar == 0)
        foo =  bar;
else
        foo = 0;
```

This rewriting provides separate conditional assignments to *foo*, allowing each to be converted as needed.

Arithmetic Operations

Current version of TSL does not support arithmetic operations (such as +, −, ×, ÷, or modulo) or magnitude comparison operations (such as <, <=, >, or > =). At this point this is just a limitation of our tool and we plan to add this support in our tool soon. For dealing with this issue for now, our tool detects cases where power-of-2 techniques can be used instead and performs automatic conversion. The detection depends on one of the operands being a constant power-of-2 value. In cases where this is not obvious, we have to modify the model by hand.

Some models contain complex arithmetic expressions that calculate some binning value based on one or more inputs. In these cases, we have replaced the

arithmetic expressions with if-else trees or switch statements coded to achieve the same result without arithmetic.

Driver Verification Using DML models

We use the same Simics model that is used to synthesize the driver in the Simics framework to execute and test the synthesized driver.

For some of the devices for which the hardware is available, we also tested the driver on actual hardware.

Tool Chain Capabilities

The synthesis tool chain has some additional capabilities that can be useful to a DML model writer. In the following sections we describe these capabilities and how a DML model writer can use it to their advantage.

State Space Explorer

The driver synthesis tool chain includes a utility that allows a user to visually inspect the combined device and OS state machine. The utility is a state space explorer, a graphical user interface that allows the user to perform various operations on the state machine, like analyzing available driver actions in a given device state, applying an action from the current state and inspecting the changes to the device state, and viewing the effect of external environment events.

"State space explorer allows the model developer to visualize the device model"

While the state space explorer is a critical component of a tool chain that synthesizes driver code, it also offers capabilities that can be quite useful to a DML model developer.

Visual Model Debugger

As illustrated in Figure 4, the state space explorer GUI allows a DML model developer to visualize the device model as a directed graph where each node in the graph represents a state (or a set of states) and each arc in the graph represents a transition from one state to another.

The GUI allows a model user to inspect the values of any device internal variable in a given state by simply clicking on the node in the graph representing the state. A pane on the left lists all the device internal variables, and clicking on a particular state node causes this list to be updated with the values of each variable in that state.

Further, from a given state, the GUI allows a user to pick the next transition which would move the device state machine to another state. While this feature is somewhat similar to the *step* or *next* operation in a traditional software debugger, the event-driven nature of a DML model requires the tool to provide more flexibility. The events triggering state transitions are broadly classified into events that can be controlled by software and those that depend on the environment (like platform hardware interrupt, line unplugged, and so on) and therefore cannot be controlled by the device or software. The tool allows a user to choose which event occurs next in a

given device state. The choice includes both controllable and uncontrollable events. In the case of software-controlled actions, the user can also specify the parameters of the action.

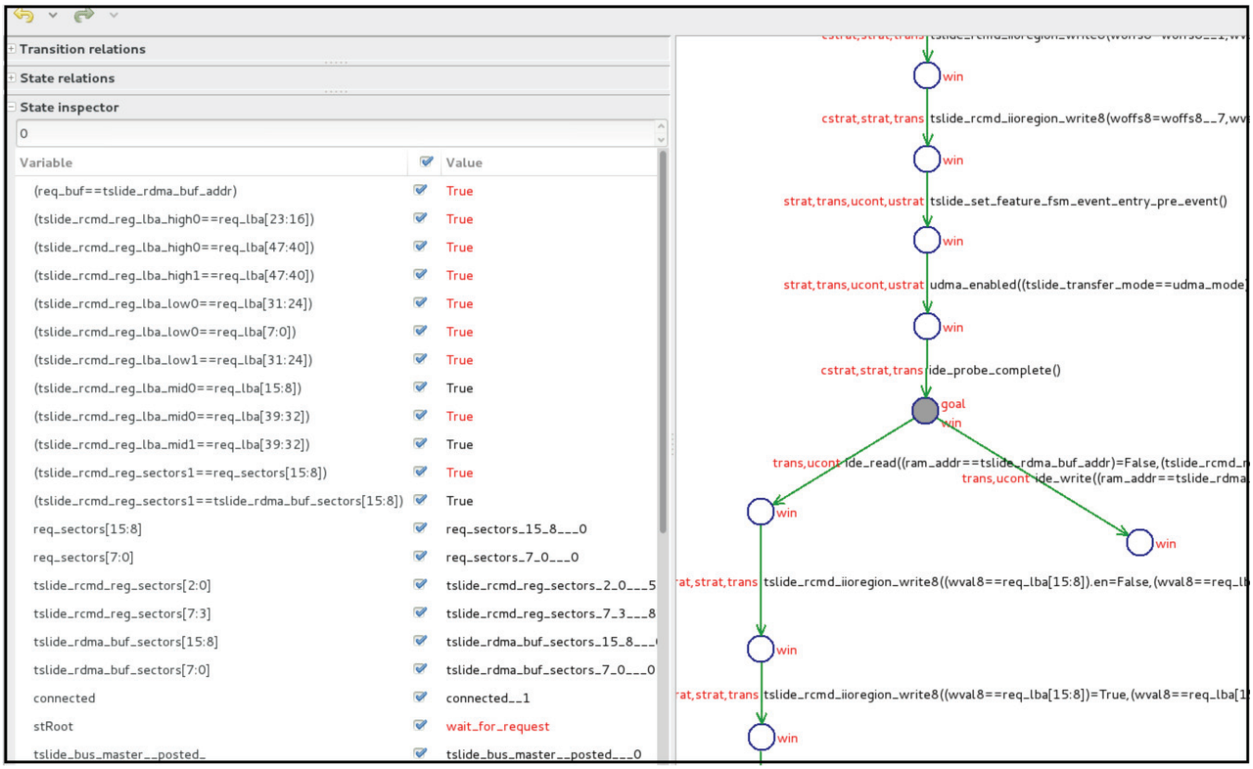

Figure 4: State space explorer GUI. The right pane shows the device model as a directed graph. The left pane shows device internal variable values.
(Source: Intel Corporation, 2013)

"State space explorer provides a counter example when no winning driver strategy exists"

The capabilities described above (inspecting device variable values and directing the state machine by choosing the next transition via the GUI) allow the model writer to use the state space explorer as a debugging aid, examining the effect of (a chain of) events on the device.

Counterexample Generation

The primary challenge in exploring the state space of a hardware device model is its huge size, which would quickly make visualization incomprehensible and state management cumbersome. The GUI explorer utility in the synthesis tool chain employs numerous techniques, built on a foundation of formal methods and symbolic execution to address this issue. These techniques include:

- aggregating states with the same properties with respect to the DML mode code into a set of states and displaying the entire set as one node
- symbolic representation of the model code, which allows abstracting the model variables (which can have a massive number of values)

into Boolean predicates that distinguish specific paths through the code

- showing only relevant subset of actions and parameter values when adding a state transition
- automatically "running" (tracing out a path in the device state machine) till a specified "way-point condition" (a predicate expressed over device model variables) is true

One of the most useful capabilities from a model developer's perspective is the tool's ability to generate counterexamples. The normal operating mode is to develop a successful strategy for the driver, but when the model is buggy such that it is impossible to generate a successful driver strategy, the tool generates a counterexample, that is, a set of actions on the state machine demonstrating how the driver can be prevented from moving the state machine into a desired goal state. This is possible since the tool is built on top of a formal representation of the model.

Providing counterexamples is very useful to a model developer as they can be presented with a specific sequence of actions on the device model that would lead the model into an undesirable state.

Scenario Replication

"Visual tool allows for scenario replication by supporting save and restore feature"

Device programming sequences typically involve massaging of OS input parameters, a long series of register reads/writes, and require specific environment conditions (such as network connectivity for a successful packet transmission) to hold. In order to assist the tool user in efficiently exploring the device state space and quickly repeating long repetitive action sequences, the GUI allows saving traces of action sequences, also known as state transitions, from any given state. In any subsequent run of the tool, as long as the model remains unmodified, the same scenario can be replicated by bringing the model to the same start state and then loading the trace saved.

This capability can be very useful for software-hardware co-development allowing device-driver and device-model developers to work together closely. The driver developer can initiate some OS-based scenario and capture its effect on the device model internals for the model developer to replicate. Typically such errors (for example, race conditions, synchronization errors, or deadlocks) involve very specific interactions of the software, device, environment, and OS actions, making it hard for model developers to replicate the exact error conditions being encountered in a complete system. While Simics does allow easily simulating the complete system state to replicate errors, the model developer would still need to instrument the DML model code with appropriate debug logic (typically log messages, to determine the root cause of the problem). The distinction is similar to classic software debugging done by adding code to print debug messages versus using a debugger to find problems.

The combination of the capabilities to explore model state space, counterexample generation and scenario replication allows a DML model writer to quickly narrow the search for bugs in DML device models as they are directly able to examine the device-internal state in the discovered failure paths.

Prototype Device Drivers

We have successfully synthesized device drivers for multiple nontrivial devices using DML device models. We used some existing models and developed some from scratch. For all the drivers the synthesized code was limited to driver code that handles device specific operations like initialization, configuration, and data transfer. We embedded this synthesized driver code in manually developed wrappers for code that involves OS and bus resource allocation and any data transformation. Resource allocation includes allocating IRQ lines, setting up DMA descriptor rings, creating mappings for memory-mapped device regions, and so on. Data transformations performed by drivers include preprocessing data buffers sent to the device, such as, for example, changing their alignment or padding, and postprocessing data received from the device, such as extracting checksum from a network packet. While many of these operations can in principle be formalized and synthesized using the game-based approach, we believe that a different formalism is needed to automate synthesis of this functionality. We successfully synthesized low-level drivers for the following devices:

- Legacy IDE Controller –Linux driver from manually developed DML model from device datasheet
- W5100 Embedded Ethernet Controller – Native firmware driver from manually developed DML model from device datasheet
- Intel PRO/1000 Ethernet Controller – Linux driver from manually developed DML model from device datasheet
- UART NS16450 – Linux driver from existing DML device model
- SD Host controller – EFI driver from existing DML model

"The synthesis tool has been used to successfully generate device drivers for several non-trivial devices"

SD Host Controller Case Study

This section describes the steps involved in synthesizing a UEFI SD Host controller driver from scratch. This case study is considered in detail here because it is based on using a preexisting device model. As such, it is the most representative of the intended use of this technology.

Input Specifications

Driver synthesis requires three input specifications for the device. This section describes the steps involved in acquiring/developing three input specifications.

Device Specification

We used an existing SD host controller DML device model from Simics team as our device specification. As we began to examine the model to determine where the device-class related annotations should be placed, we noticed that unlike the other DML models we had worked with, this model did not account for in-flight data transfer times. All data transfers to or from the card model happened instantaneously. Our past experience led

us to believe that we would not be able to successfully synthesize a driver from a model in this condition. The problem is that the instantaneous completion leads the synthesis algorithm to assume that any operation started in cycle x will be complete in cycle $(x + 1)$, eliminating the need to poll status registers for an indication of completion, and so on. Therefore, our first step became a rewrite step.

We rewrote the model to account for the in-flight times and validated the changes using a stock Linux image with the Linux SD Host driver, running on the Simics Framework. We submitted patches for these changes to Simics.

We then began the task of annotating the model with Device Class events and attempting synthesis. As this model was the most complex model we had tried to date, we immediately ran into problems. The complexity of the model resulted in an output TSL file with 6.8 Kb of state space (global variables), another 12.3 Kb for temporary variables, and 45 separate transitions. This extreme size resulted in tool-chain execution times in excess of 4 hours. As we were still trying to determine the correct locations for annotations, the extreme execution time was a significant hindrance to forward progress.

Since the model is a full model, it contains transfer modes and registers that would not be used in our project. In an attempt to reduce the overall size and complexity, we tweaked the model to hide the unused transfer modes and registers. This reduced model has 2.5 Kb of global variable space, 1.5 Kb of temporary variable space, and 14 separate transitions. This reduced tool-chain execution time to tens of minutes.

We also had to make a few changes to the model for TSL compatibility issues. These changes included rewriting arrayed register definitions without arrays, statement adjustments to allow width conversions, and elimination of arithmetic operations.

Class Specification

We needed to define this specification from scratch as it does not exist today. Normally we expect it to be published with the device industry standard specification. This specification defines all the interfaces supported by the SD Host controller device that are expected to be supported by all the drivers. We started with SD host controller standard specification[6] and defined the class interfaces. This is defined as a Word document. The class interfaces are the points of synchronization between OS and device specifications. We will use these interfaces to annotate both the OS and device specifications.

OS Specification

We chose to synthesize the SD host controller driver for UEFI (Unified Extended Firmware Interface). We used UEFI documentation[5] to define this specification. The SD host controller driver is the lowest level driver in the layered driver stack. The OS specification for this driver was motivated by the interfaces expected by the media layer driver.

"The SD host controller DML model was annotated to work with the tool"

UEFI defines a stylized model of system booting that includes interfaces between several different executable entities, including UEFI drivers, as shown in Figure 5.

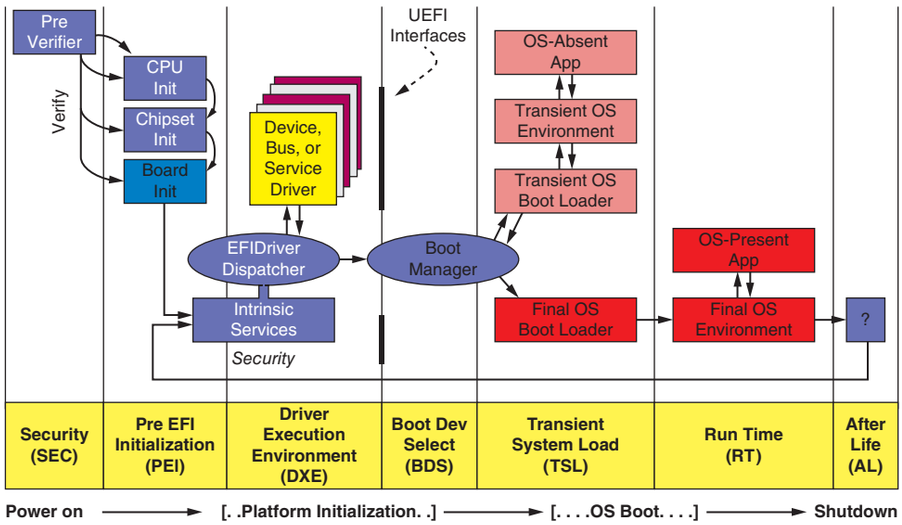

Figure 5: UEFI boot sequence
(Source: Intel Corporation, 2013)

"Strong assurance guarantees needed for firmware along with the extensive specifications available in UEFI make EFI drivers an ideal target for synthesis"

These interfaces are codified by the main UEFI specification and expose abstractions such as block device access, such as the EFI_BLOCK_IO_PROTOCOL. The generic services in the EFI_BLOCK_IO_PROTOCOL, such as ReadBlocks(), WriteBlocks(), and Reset(), need to be refined to an implementation that meets the requirements of the underlying hardware controllers. Today the requirements of the UEFI specification and its associated driver model, along with the semantics of the hardware, are all managed by the developer as part of the code creation process. This process is error fraught, and most developers typically take an existing driver source and "port" it to the requirements of the new hardware. As such, there is no guarantee of correctness, with flawed "existing sources" being evolved via this porting process.

Instead, with the driver synthesis, a single instance of an OS specification for a class of devices can be married to a specific device specification, such as the DML for the hardware, to derive the source. This removes the errant human interpretation of the UEFI specification and the hardware host controller interface definition.

This is an important issue in that the UEFI firmware on the system board is considered hardware by many end users of the platform. And with the trust guarantees around platforms based upon UEFI Secure Boot[7], assurance considerations, such as correctness of the implementation, gain even more importance as all of the UEFI drivers and components are in the same trusted computing base.

Specification Synchronization

We used the class specification as synchronization between the OS and device specification. This involved using the class interfaces in the OS specification at the synchronization points. Finding the correct synchronization points involved studying the DML device model. Finding the correct place to annotate the device model depends on the way the model is written. It was a fairly simple process to annotate the SD host controller and EFI OS specifications.

Integration

Once we had the three inputs ready, it was an iterative process to input them through our tool chain to synthesize the driver. We did not synthesize the configuration interfaces for this device, but synthesized the main function to send a command to the card. At the end of this step we were able to synthesize the device driver strategy for this driver.

Code Generation

Code generation proved much mode tedious than anticipated. At the time of writing, our synthesis tool does not support fully automatic code generation. Instead, it allows the user to interactively construct driver source code by selecting one of several possible actions proposed by the winning strategy in each state. Ongoing research on this problem is focusing on techniques for fully automatic code generation as well as on improved methods for interactive user-guided code generation (see the section "User-Guided Synthesis").

"The synthesized code generated by the tool was tested in the Simics simulator with an Intel® Core™ i7 based platform model"

Testing and Validation

We used the Simics simulator of a target platform based on the Intel® Core™ i7 processor for testing this EFI driver. This model does not contain an integrated SD host controller so our first step involved adding our SDHCI device model to the platform. We created a Python wrapper to instantiate our SDHCI model and Simics MMC Card model and integrated the wrapper into platform model startup script. The startup script was modified to connect the host controller to the platform model through an unused South Bridge PCI bus slot.

With the platform model extended, the next step was to validate the extended model. This was done using the Linux image supplied with the platform model. We booted the image in Simics and recompiled the kernel to create a loadable Linux SDHCI driver. We updated the Linux image to retain the new driver modules. We were then able to load the SDHCI driver and validate our SDHCI-MMC card model combination using Linux file-system commands targeted to the MMC card.

Our next step was to establish an EFI baseline image. To achieve this goal, we built an EFI image with an existing SD host controller driver and tested that simulation environment. We then integrated our driver with the EFI code base, replacing the existing driver. We needed to develop some wrapper code to integrate in EFI environment. We then built and tested this driver on the Simics simulator and successfully brought up the SD host controller and performed read/write operations to the SD card.

"User guided synthesis allows a driver writer to have fine grained control over the driver synthesis process"

User-Guided Synthesis

Our initial approach with this project was complete automatic synthesis, where once the specifications are available, a push-button approach will result in a driver. In practice we realized that users want much more control over the structure of the driver code. In addition, in some cases synthesis gets stuck, and having users provide some simple hints can make the job of the synthesis tool much easier. Given these findings we decided to make a shift toward user-guided synthesis, as illustrated in Figure 6.

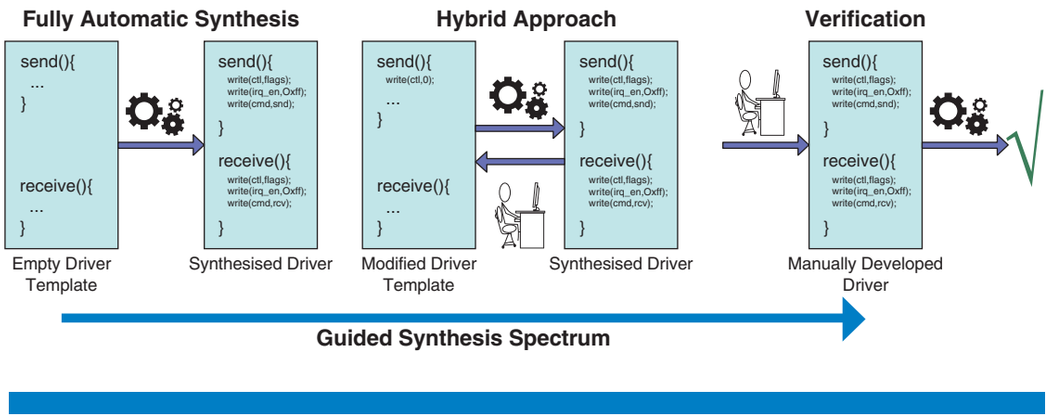

Figure 6: Guided synthesis spectrum (Source: NICTA, 2013)

To this end we plan on using driver templates that specify the driver structure. The user can add additional constraints on the synthesized driver by defining a device-specific driver template that can include some hints, or anything that is specific to a device. We plan on supporting a complete spectrum from fully automatic synthesis, where the device-specific template is empty, to the other extreme, where the user manually writes the complete driver in device-specific template and our tool can then act as a verifier to verify the driver against input specifications. We think the sweet spot is somewhere in the middle, where the user specifies some code structure and constraints in the device-specific template and generates more usable and readable code (see Figure 7).

We are also working on an interactive code generation GUI that gives user the flexibility to add any code at code generation steps. Any code manually added this way using the code generation GUI is saved by adding it back to the template and will automatically be available at the next iteration. Using a combination of templates and code generation GUI, our tool chain will provide user control over generated code at all stages of synthesis. Even though the user gets complete control, our tool chain will validate that the user has added correct code. Any errors caused by the user will result in synthesis failure and not an incorrect driver.

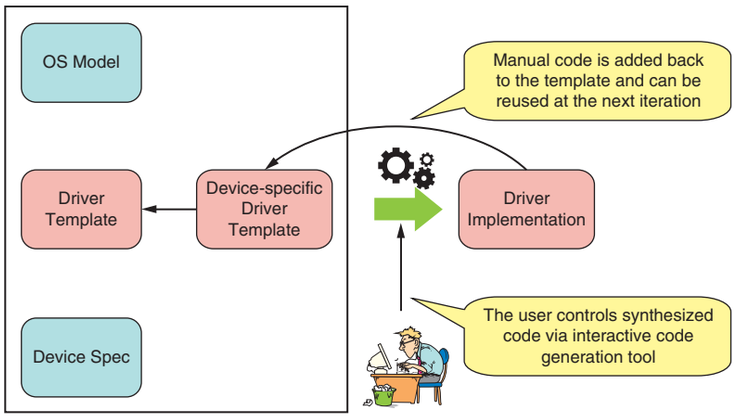

Figure 7: User-Guided synthesis with templates (Source: NICTA, 2013)

Future and Summary

Using existing device models for driver synthesis is a great start, but in practice we realized that we had to modify and annotate the models extensively in order to make them suitable for synthesis. In the future we hope to work with model writers to lay down requirements for writing device models with synthesis in mind, so as to reduce manual intervention to annotate or modify the models.

"Synthesis with user guidance has a potential to achieve the holy grail of fine grained user control with formal guarantees of correctness for generated device drivers"

Complete References

[1] A. Chou, J.-F. Yang, B. Chelf, S. Hallem, and D. Engler. An empirical study of operating systems errors. In *18th SOSP*, pages 73–88, Lake Louise, Alta, Canada, Oct 2001.

[2] A. Ganapathi, V. Ganapathi, and D. Patterson. Windows XP kernel crash analysis. In *20th LISA*, pages 101–111, Washington, DC, USA, 2006.

[3] N. Piterman and A. Pnueli, "Synthesis of reactive designs," in *Proceedings of Verification, Model Checking, and Abstract Implementation (VMCAI)*, 2006.

[4] R. E. Bryant, "Graph-based algorithms for Boolean function manipulation," *IEEE Transactions on Computers,* Vols. C-35, no. 8, pp. 667–691, August 1986.

[5] Unified EFI Forum, [Online]. Available: www.uefi.org/home.

[6] SD Association, "SD Host Controller Standard Specification Version 4.00," SD Association, 2012.

[7] Magnus Nystrom, Martin Nicholes, Vincent Zimmer, "UEFI Networking and Pre-OS Security," in *Intel Technology Journal - UEFI Today: Boostrapping the Continuum,* Volume 15, Issue 1, pp. 80–101, October 2011

Author Biographies

Mona Vij is a researcher in Intel Labs. She has been a security and operating systems researcher for over 20 years. She has a Masters in Computer Science from the University of Delhi, India and a Bachelor of Science in Mathematics from St Stephen's College, Delhi.

John Keys is a Staff Engineer in Intel Labs. He has been developing low-level software for over 25 years, for both PCs and embedded platforms. He has experience with a wide range of hardware devices, CPUs, operating systems, processor architectures, and platforms from bare-metal to PC to satellites and tunnel boring machines. He has made significant contributions to the development of PCMCIA and USB technologies and standards. Through this leading edge work, he also became an expert in "hacking" an existing platform to add new capabilities, beginning with plug-and-play support for MS-DOS3.2. John has been with Intel for 14 years in a variety of positions. Prior to joining Intel, he was the VP of Software for MCCI in Ithaca, NY.

Arun Raghunath is a Senior Software Engineer in Intel Labs. He has a Masters in Computer Science from University of Southern California, and a Bachelors in Computer Science & Engineering from Pune University, India.

He has been a Systems software researcher at Intel for the last 14 years. He has authored 5 conference papers, 1 book chapter and holds 8 patents in the areas of High performance computer networking, Operating Systems, Compilers and multi-core parallelization.

Scott Hahn is a Principal Engineer in the Systems Architecture Lab within Intel Labs where he leads the Operating Systems Research team. His team primarily focuses on the interaction of system SW and HW. Their projects cover multiple areas including storage, scheduling, memory and device drivers. Scott has been with Intel since 1994 and joined Intel Labs in 2006. Prior to joining Intel Labs, he was an architect in the LAN Access Division (LAD) where he worked on a number of network technologies and was the lead architect of Intel's Active Management Technology (Intel® AMT). Scott also worked in Intel's Supercomputer Systems Division where he was responsible for developing Intel's IP over ATM solution for the world's first TeraFLOP super computer. Scott has published over 15 technical papers, holds 13 patents, and has received an Intel Achievement Award for his work on Intel® AMT.

Vincent Zimmer is a principal engineer in the Software and Services Group at Intel. He has been firmware developer for over 20 years. He has a Bachelor of Science in electrical engineering from Cornell University, Ithaca, NY, and a Master of Science in computer science and engineering from the University of

Washington, Seattle, WA. He has published three books, two book chapters, one IETF RFC, ten publications and over 270 US patents.

Leonid Ryzhyk is a Postdoctoral Fellow at the University of Toronto and Researcher at NICTA. He obtained a PhD in Computer Science from the University of New South Wales, Sydney, Australia in 2010. He received his Bachelor's and Master's degrees in Computer Science from the National Technical University of Ukraine in 2000 and 2002.

Adam Walker is a PhD student at the University of New South Wales, Sydney, Australia. He obtained his Bachelor's degree from the University of Auckland, New Zealand in 2008.

Alexander Legg is a PhD student at the University of New South Wales, working with NICTA in Sydney, Australia. He received a Bachelor of Information Technology (Hons) from the University of Sydney in 2011.

USING SIMICS IN EDUCATION

Contributor

Robert Guenzel
Technical Content Engineering,
Wind River

Wind River Education Services provides user training for a variety of topics, including Wind River operating systems and tools, as well as more general topics like networking. Training always includes hands-on labs, which can complicate logistics for training sessions. Shipping boards and configuring networks is time consuming and error prone. For that reason, education services are using Simics as an alternative to physical hardware to streamline training logistics and provide new ways to do training.

Introduction

"It is not about training people on Simics."

By just looking at the title of the article, it remains unclear what kind of education this article refers to. A more precise but woefully long title would be: "Using Simics* for educating people on various embedded system topics, such as debugging tools, operating systems, device driver and application development, networking, and security." Apparently, Simics is not amongst the topics people are educated on. So this article is about using Simics as a training tool. It is not about training people on Simics. In fact, the training examples in question try to hide Simics as much as possible, because the students must not get the feeling that they are being trained on something they did not want to be trained on, or, which would be even worse, get the feeling that they need Simics in order to use the tools or software they are being trained on.

Using Simics in this way reduces Simics (more precisely, the target machine that it simulates) to a mere hardware replacement, thereby throwing away a lot of its unique features. To understand the reasons for this, a closer look at hands-on labs is required.

Training people on embedded software tools and techniques involves hands-on labs, within which the students can apply what they have learned within the lecture parts. These labs are essential for the overall learning effect because during the labs the taught concepts can settle in. To ensure a smooth lab execution, the complete lab setup needs to be provided to the customers, and hence executing such labs imposes the following challenges on the conductor of the training:

- Equipment has to be shipped—Shipping equipment costs time and money, and bears the risk of having the equipment not arriving on time or becoming defective.
- Reliability of equipment—Regular shipment and in-lab use of equipment eventually leaves it in a brittle state and leads to potential failures during use.

- Amount of equipment—There are literally hundreds of pieces of equipment per class (a laptop, evaluation board, Ethernet switch, hardware debugger, and cables per student). The overhead for making sure all pieces are there, in good shape, and recollected after training is significant.
- Installation—Creating networks, connecting the debuggers, probes, and so on at the customer site is very time consuming and error prone.
- Insight into the system—Real hardware does not allow full system time freeze and insight into the complete system state.
- Flexibility—The same training sessions need to be delivered on different target machines such as PowerPC, ARM, or x86.
- Scalability—The number of CPU cores or nodes in a network needs to be changeable on the fly to show effects of concepts introduced within lectures.

"…There are literally hundreds of pieces of equipment per class…"

All of the above points have a significant impact on training maintenance and training delivery—and at the end of the day training costs. With Simics, a lot of the above problems can be mitigated:

"With Simics, a lot of the above problems can be mitigated:"

- No need to ship equipment other than normal laptops.
- Broken laptops are easy to replace, because they are off the shelf.
- The required equipment is independent of the training that is delivered. It is always one laptop per student.
- Simics gives full insight into the system and allows full system time freeze.
- For a number of training sessions a change of the used target is simple. For example, migrating from the Simics PowerPC* Quick Start Platform to the Simics ARM* Quick Start Platform.
- System installation and bring-up is reduced to powering on the laptop and starting Simics with a well-tested Simics script.

The following sections show the application of Simics in a number of training examples, showing the real-world problems and how they were solved.

Device Driver Development Training

When teaching people on device driver development, the hands-on labs should present them with the most central tasks involved in device driver programming. This means the labs require a device that is

- simple enough to understand within the duration of the lab
- complex enough to warrant some device driver activity
- available in various platforms such as ARM, PPC, and x86

The only real-world device that fulfills most of the above requirements is a 16x50 UART. However, this device is about 20 years old and hence is not at all exciting. Needless to say, there is hardly any need for developing a device driver for it, because every OS already has one. In order to make the lab motivating and interesting, something else is needed.

"In order to make the lab motivating and interesting, something else is needed."

Another option is to design an artificial device and program an FPGA accordingly. This, however, would then require having boards with an FPGA for each of our target architectures (ARM, PPC, and x86). Leaving aside the question of whether all targeted operating systems would run on such boards, it is still highly likely that these boards would be only vaguely similar and hence would lead to a big development and maintenance effort for the training labs. Additionally, not knowing what the next customer would want to be trained on would require purchasing many more boards than would ever be needed. Furthermore, the FPGA-based boards could simply break, and reprogramming the FPGAs would require shipping even more equipment.

A third option is to use Simics. Simics also allows designing an artificial device. Having it included into one of the Quick Start Platforms (PowerPC or ARM) would automatically make it available in the other, because both platforms are—apart from the processor used—identical. With the way Simics models memory-mapped busses, it is also not difficult to put the device into an x86 platform.

The device created has the following specification:

- The device has only 32-bit registers.
- The device is an LED controller driving 32 LEDs with a value in its pattern control register (PTN).
- The device has a main control register (MCR).
- The device has a pattern life time (PLT) register.
- The device is enabled with MCR[0] = 1. With MCR[0] = 0, no pattern is driven.
- The device has two operation modes. Mode 0, "Memory mapped I/O only" and Mode 1, "DMA". The mode is chosen by setting MCR[1].
- The device can drive an IRQ to the CPU.
- Once PLT! = 0 the device decrements PLT every cycle and raises an IRQ once PLT = 0. Decrementing PLT is suspended while the IRQ is raised. PLT is immutable from the outside once decrementing starts.
- Writing the MCR clears the IRQ.
- When entering Mode 1 or when in Mode 1 and after clearing the IRQ, the device shall read the values of PNT and PLT from the addresses MCR[31:2]<<2, and (MCR[31:2]<<2) + 4 respectively.

"The specification is easy to comprehend, but still offers IRQs, DMA, and normal register I/O."

The specification is easy to comprehend, but still offers IRQs, DMA, and normal register I/O. The idea here is that one can create some kind of moving pattern by updating PNT and PLT in response to IRQs, as depicted in Figure 1.

The resulting I/O definitions of the DML device are shown in Code 1.

```
dml 1.2;
device myDevice;
parameter desc = "LED Controller";
parameter documentation = "LED Controller with integrated timer";
```

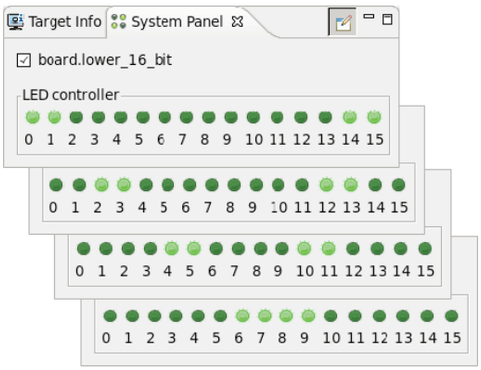

Figure 1: A running light implemented using the training device.
(Source: Wind River, 2013)

```
import “utility.dml”;
import “simics/devs/signal.dml”;
import “simics/devs/memory-space.dml”;

connect phys_mem {
    parameter documentation = “Connect memory space for”
                              +“DMA operations.”;
    parameter configuration   = “optional”;
    interface memory_space;
}
connect irq_dev {
    parameter documentation = “Connect IRQ receiver.”;
    parameter configuration   = “optional”;
    interface signal;
}
connect led[32] {
    parameter documentation = “Connect LEDs.”;
    parameter configuration   = “required”;
    interface signal;
}
bank regs {
  parameter register_size = 4;
  register MCR @ 0 x 00 is (read_write) “Main Control Reg”{
    field DMAADR [31:2] “DMA source address”;
    field MODE   [1]   “Operation mode of device”;
    field ENABLE [0]   “Enable/Disable device”;
}
  register PTN @ 0 x 04 is (read_write) “Pattern Reg”;
  register PLT @ 0 x 08 is (read_write) “Life Time Reg”;
}
import “myDevice_impl.dml”;
```

Code 1: The DML code for I/O definitions of the training device.
Source: Wind River, 2013

The DML code shown in Code 1 is a straightforward implementation of the specification. It has one bank with the specified registers, an array of 32 connect objects for the LEDs, and another two connect objects for the IRQ receiver (usually the PIC of the system) and the memory space for DMA operations, respectively.

The DML file "myDevice_impl.dml" then implements a number of convenience functions for the connect objects (as shown in Code 2), the side effects of the register accesses (like switching the LEDs on and off), and the event required to handle the pattern life time (as shown in Code 3).

```
connect irq_dev {
  method set(){
     $irq_state=true;
     if ($this.obj)
        $irq_dev.signal.signal_raise();
  }
   method reset(){
      $irq_state=false;
      if ($this.obj)
        $irq_dev.signal.signal_lower();
  }
}
```

Code 2: Convenience functions in the IRQ connect object.
Source: Wind River, 2013

```
event lifetime_event {
      parameter timebase = "cycles";
      parameter desc = "lifetime expiry event";
      method event(void *param) {
                  inline $irq_dev.set();
                  $regs.PLT=0;
   }
}

bank regs {
  register PLT {
    method write(val){
     if ($this==0 && val!=0){
          $this=val;
          if ($irq_state==false && $MCR.ENABLE==1){
             inline $lifetime_event.post(val,NULL);
          }
```

```
    }
  }
  [...] //part omitted
}
```

Code 3: Use of an event to handle the pattern life time.
Source: Wind River, 2013

The DML code is simple (myDevice_impl.dml has around 150 lines of code) and can be maintained without exhaustive DML knowledge. Integrating the device into the Quick Start Platform has been done at the scripting level as shown in Code 4, because this allows us to have a single script for both targets. Also note that the script handles the chosen OS, so there is only one script for four different training variants.

"…the script handles the chosen OS, so there is only one script for four different training variants."

```
if not defined target_arch { $target_arch = ppc }
if not defined target_os   { $target_os   = vxworks }
if not defined target_image { $target_image = someKernel}

$vxworks_image = $target_image
$kernel_image = $target_image

add-directory "%script%"
run-command-file ("%simics%/targets/qsp-"+$target_arch+"/qsp-"+$target_
os+".simics")


new-qsp-led-panel name = $system.upper_16_bit
new-qsp-led-panel name = $system.lower_16_bit


@device=pre_conf_object(simenv.system+'.dut','myDevice')
@device.led = [None]*32
@device.queue = SIM_get_object(simenv.system).cpu[0]
@device.phys_mem= SIM_get_object(simenv.system).phys_mem
@device.irq_dev = [conf.board.pic, "irqs[80]"]
@tmp1=SIM_get_object(simenv.system+'.lower_16_bit')
@tmp2=SIM_get_object(simenv.system+'.upper_16_bit')
@for i in range(16):
  exec('device.led[%d]=tmp1.led%d'%(i,15-i))
  exec('device.led[%d]=tmp2.led%d'%(i+16,15-i))
@SIM_add_configuration([device],None)
$system.phys_mem.add-map base = 0xe00a0000 length = 12 device = $system.
dut:regs align-size=4
```

Code 4: The Simics script to insert the device into the QSP.
Source: Wind River, 2013

The nice thing here is that the above code works with minor changes for an x86-based target as well. The difference is that the phys_mem memory space of an x86 target is not a direct child of the top level component. That means, for all targets, the same code base can be used for the device and the Simics machine script. Only the above code needs to be maintained in order to enable device driver labs for VxWorks* and Linux* on all three target architectures. So for this training example, the advantage of using Simics lies on the training developers' and maintainers' side.

"The students are mostly unaware of Simics…"

The students are mostly unaware of Simics, although the training can be extended to show how Simics can help in device driver development and debugging, such as, for example, using the device register view (see Figure 2) to inspect the current device state.

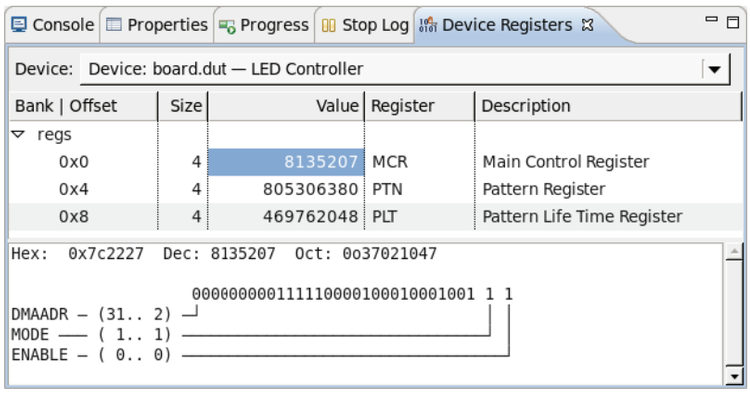

Figure 2: The simics 4.8 register view showing the registers of the training device
(Source: Wind River, 2013)

Networking Training

Networking labs have quite different requirements from the device driver development labs. Here, the actual used target machines are of secondary interest; what is more important are:

- Flexible network topologies
- Heterogeneity of nodes. That is, mixed endiannesses, different operating systems.
- Inspection on each node of the network
- Fault injection like packet drop or packet corruption

"Networking training sessions are also the best example of why shipping real equipment is not a viable option."

Networking training sessions are also the best example of why shipping real equipment is not a viable option. Consider a simple network that consists of only four leaf nodes organized into two subnets that are connected over a router (Figure 3).

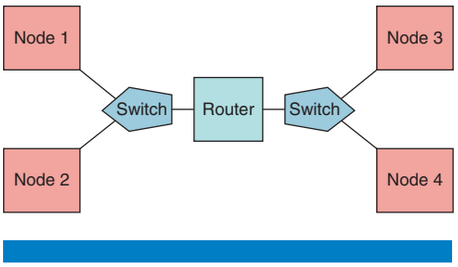

Figure 3: A simple setup for a networking lab (Source: Wind River, 2013)

In this case the setup requires five machines, two switches or hubs, and six Ethernet cables. This renders per-student lab setups impossible (imagine a class of 10 or more students). A per-class setup is feasible, but still subject to reliability issues. Taking the above requirements into account (especially the flexible topologies) using real hardware is no option.

"A per-class setup is feasible, but still subject to reliability issues."

An alternative to this is to use simulated networking like vxsimnet as offered by the VxWorks simulator. With it, creation of various topologies is simple, but the problem with vxsimnet is that it would restrict the labs to one OS and one endianness. Additionally, network analysis would be performed on host TAP interfaces. That means any training related to Frame Check Sequence (FCS) is not possible because FCSs are generally unavailable when sniffing a real network.

Simics' Python scripting interface allows implementing a configuration file format and a parser for it that can create the topologies needed. The configuration file has to be able to define the number of targets, the endianness of a target, how targets are grouped into subnets, and how subnets are to be connected.

"The configuration file has to be able to define the number of targets, the endianness of a target, how targets are grouped into subnets, and how subnets are to be connected."

Since the parser is written in a Python script, a generic script can be created that starts the parser with a given configuration file (Code 5).

```
if not defined wru_config {
  interrupt-script "No network config provided!" -error
}


if (file-exists ($wru_config+".py")) == FALSE {
  interrupt-script "Network config '"+$wru_config+"' not found!" -error
}
run-python-file ((sim->simics_base)+"/../wru_sysgen/createNetworkTopology.py")
```

Code 5: Content of the generic createNetwork.simics script.
Source: Wind River, 2013

With this script in the Simics project, various network topologies can be created by starting the generic script and setting the configuration file variable to the configuration file for the various labs (Figure 4).

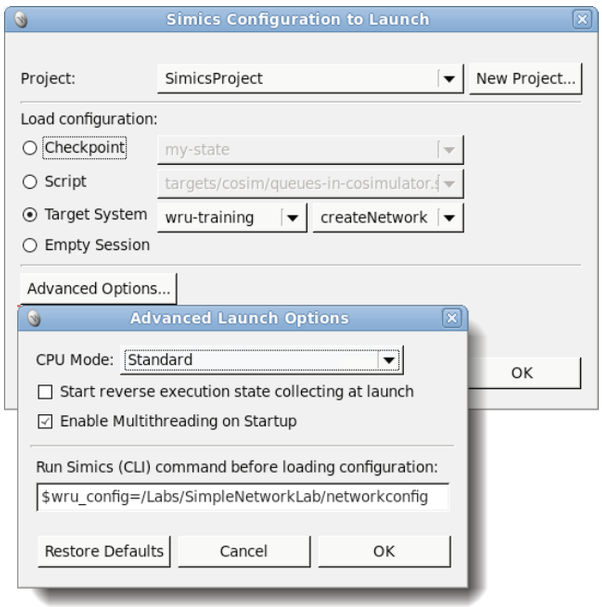

Figure 4: Launching simics using the topology generator and a configuration file
(Source: Wind River, 2013)

For example, the scripts shown in Code 6 and Code 7 generate the topologies shown in Figure 5 and Figure 6, respectively. The script in Code 6 resembles the topology shown in Figure 3, while the script in Code 7 shows a more complex topology as well as connected Wiresharks[1] and a real network connection. Note that to keep the example code short, targets and routers that use the same architecture and OS do actually use the same set of binaries. In actual labs, the routers have different kernels or root file systems in order to properly execute their router roles.

```
from wru_sysgen import *
pathToQSPARM='/home/wruser/simics-4.8/simics-qsp-arm-4.8.1/targets/qsp-arm/images/'
pathToQSPPPC='/home/wruser/simics-4.8/simics-qsp-ppc-4.8.1/targets/qsp-ppc/images/'

#vxWorksPPC, vxWorksARM, LinuxPPC, and LinuxARM are defined in wru_sysgen
```

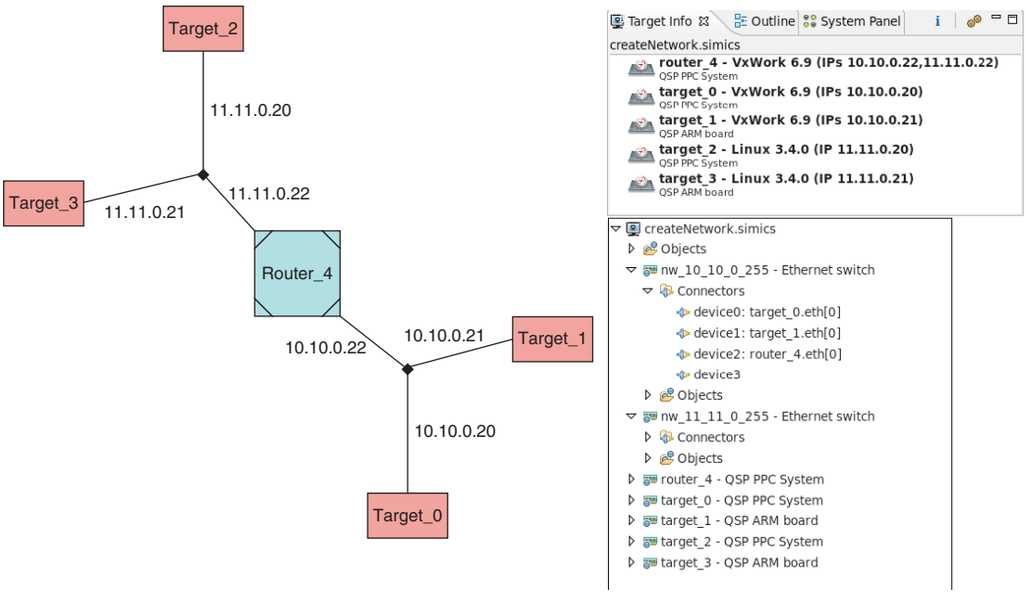

Figure 5: The generated network topology for the script shown in Code 6 as a graph, and as shown by the target info and system editor views.
(Source: Wind River, 2013)

```
#and point to the corresponding Simics machine scripts.
#### Setup arguments for targets (Script, and binaries required)
linuxFileList=['uImage','qsp.dtb','rootfs.ext2']
for arch in ['PPC','ARM']:
  exec('vxWorks%sArgs=[vxWorks%s,pathToQSP%s+"vxWorks"]'%((ar
ch,)*3))
  exec('Linux%sArgs=[Linux%s]+[pathToQSP%s+i for i in
linuxFileList]'%((arch,)*3))
routerArgs=vxWorksPPCArgs

#define targets
T1 = target(*vxWorksPPCArgs)
T2 = target(*vxWorksARMArgs)
T3 = target(*LinuxPPCArgs)
T4 = target(*LinuxARMArgs)

#define networks and connected targets
NW1 = network('10.10.0.20')
NW1.connect(T1)
NW1.connect(T2)
NW2 = network('11.11.0.20')
NW2.connect(T3)
```

```
NW2.connect(T4)

#define a router
R1 = router(*routerArgs)

#plug networks into router
R1.plug(NW1)
R1.plug(NW2)
```

Code 6: A simple topology configuration script
Source: Wind River, 2013

```
T1 = target(*vxWorksPPCArgs) #target_0
T2 = target(*vxWorksARMArgs) #target_1
T3 = target(*LinuxARMArgs) #target_2
T4 = target(*LinuxPPCArgs) #target_3

NW1 = network('10.10.0.20')
NW1.connect(T1,macAddress='f6:9b:54:32:42:42')
NW1.connect(T2)

NW2 = network('11.10.0.20')
NW2.connect(T3,macAddress='f6:9b:54:32:42:43')
NW2.connect(T4,macAddress='f6:9b:54:32:42:44')

NW3 = network('12.10.0.20')
NW3.connect_to_tap('TAP')

R1 = router(*routerArgs) #router_4
R2 = router(*routerArgs)
R3 = router(*routerArgs)
R4 = router(*routerArgs)

R1.plug(NW1)
R1.plug(R2, '13.10.0.20')
R1.plug(R3, '14.10.0.20')

R4.plug(NW2)
R4.plug(R2, '15.10.0.20')
R4.plug(R3, '16.10.0.20')
R4.plug(NW3)

wireshark('13.10.0.20','255.255.255.0')
wireshark('11.10.0.20','255.255.255.0')
```

Code 7: A more complex topology configuration script (argument setup omitted for spatial constraints and because it is identical to the simple script)
Source: Wind River, 2013

Simulating a network with a dozen nodes on a single simulation host sounds like a big challenge for the host, but in fact, once all targets have booted, most nodes are idle, meaning the idle nodes only add minimally to the overall simulation effort. For example, on a two-core Intel® Core™ i7 CPU with 2.6 GHz and 2 GB of RAM with a Linux 64-bit host OS, booting all eight machines in the network shown in Figure 6 takes 15 seconds. Afterwards simulation is able to progress by 4 virtual seconds per wall clock second, because when sending packets through the system to

"Simulating a network with a dozen nodes on a single simulation host sounds like a big challenge for the host…"

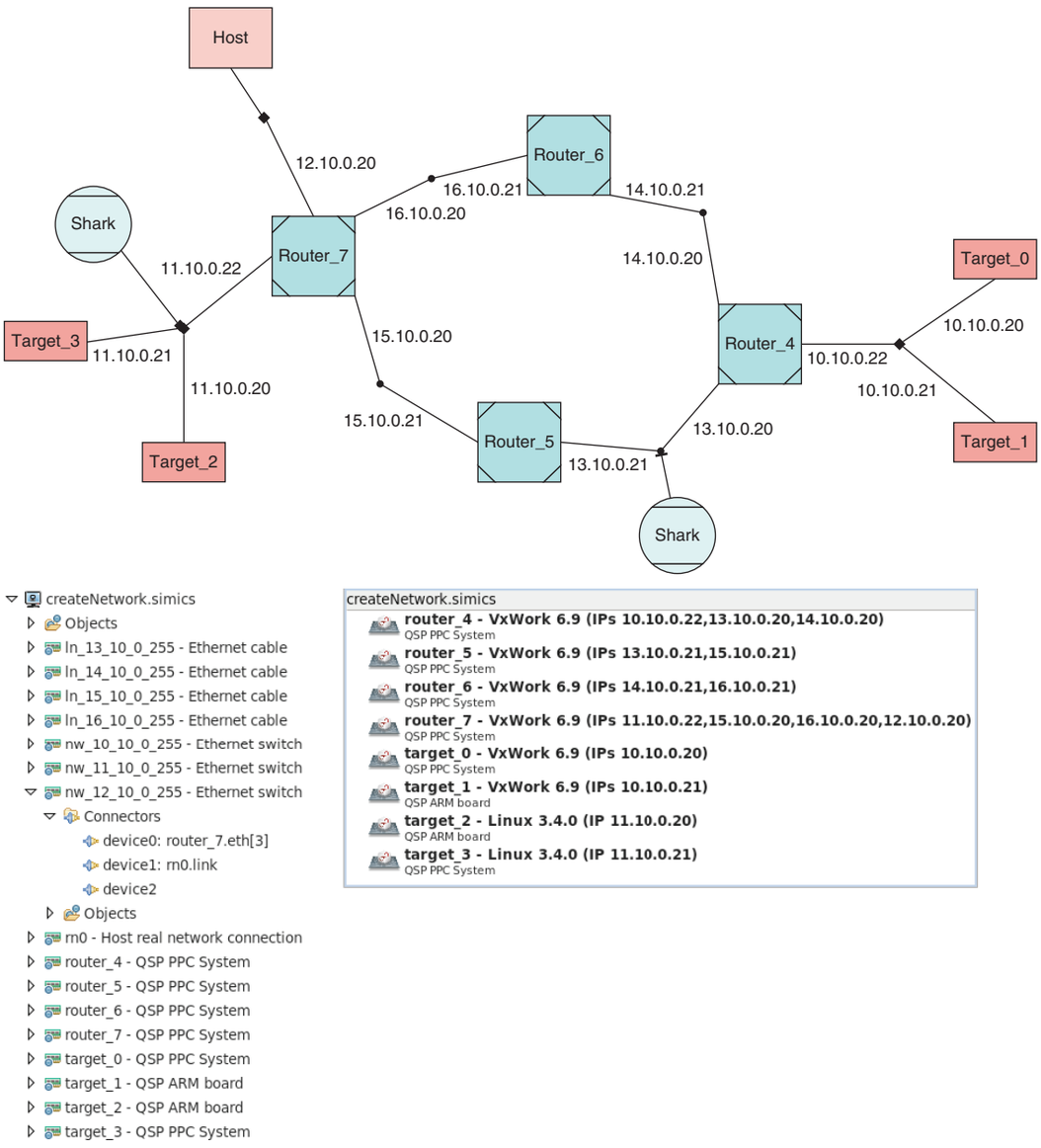

Figure 6: The generated network topology for the script shown in Code 7 as a graph, and as shown by the target info and system editor views.
(Source: Wind River, 2013)

observe the behavior of routing protocols or firewalls, most of the machines are idling.

Another noteworthy effect of using Simics is that it allows showing Ethernet Frame Check Sequences (FSCs). The problem with real hardware is that Software Ethernet sniffers (like Wireshark) are sitting on the software side of the Ethernet card. FCS is usually offloaded to hardware and hence cannot be captured, which means showing a "live" Ethernet frame in its entirety is not possible. With Simics, sniffing at the simulated hardware side of the network card is possible, and so the FCSs are observable. Figure 7 shows a packet captured on the hardware side and Figure 8 shows the same capture at the software side of the network card.

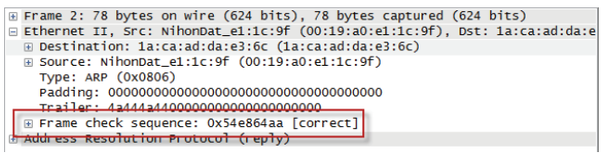

Figure 7: Ethernet frame captured on the virtual hardware side using simics
(Source: Wind River, 2013)

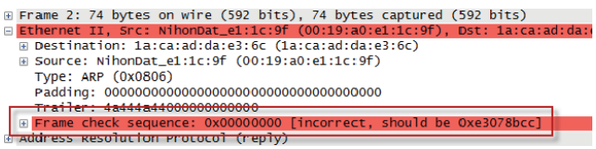

Figure 8: Same ethernet frame as shown in Figure 7 captured on the software side.
(Source: Wind River, 2013)

"…the capturing tool falsely interpreted the final four bytes of the frame (which actually belong to the trailer) as the FCS, which leads to a false check failure."

Note that the frame captured on the software side is four bytes shorter, because it lacks the real FCS. Also note that the capturing tool falsely interpreted the final four bytes of the frame (which actually belong to the trailer) as the FCS, which leads to a false check failure.

Combining the simple configuration format with the shown ability of sniffing on the virtual hardware side and with the ability of doing packet modification using Simics Ethernet probes, quick and reliable creation of any kind of network is feasible. Inspection, injection, or modification of traffic at any position in the network is also possible. This enables the construction of labs to show routing and communication protocols in action, their reaction to broken connections, and their reaction to packet loss or packet corruption in simple and complex network topologies.

In this training example, both the training developers and the students benefit from using Simics, because it simplifies the training development but also enables the use of complex and varying toplogies and inspection of complete packets everywhere, which improves the learning experience.

Multi-Core Training Examples

A third class of training examples with a different requirement set is training sessions for multi-core targets. These training sessions address topics embedded software developers face when developing applications for modern systems, like symmetric and asymmetric multi-processing, nonuniform memory layouts, and how to best exploit the number of available cores. A machine used for such a training session needs to fulfill the following requirements:

- It must provide multiple cores
- The number of active cores needs to be adjustable
- It must be able to operate in nonuniform memory architecture (NUMA) mode (that is, offer at least core local RAM)

Any modern PC nowadays is a multi-core machine, so that is no real issue, but problems arise when one wants to change the number of active cores. Limiting the number of active cores is feasible, but with real hardware, the upper limit is clearly defined by the number of available cores of the host machine. Some labs, however, require showing some laws and rules in action (like Amdahl's law) and for this, freely adjusting the number of cores is necessary in order to show how performance reaches some upper limit and a further increase of cores does not help at all.

"…problems arise when one wants to change the number of active cores."

With Simics, showing Amdahl's law can be done by implementing a simple application with a fixed sequential to parallelizable ratio. In the experiment, the parallelizable part spawns 150 tasks that all execute a fixed length for-loop, while the sequential part executed the same loop N*150 times for an N:1 sequential to parallelizable ratio. The more cores the system offers the faster the application will finish. Executing such an application on the QSP with 1 to 8 cores generated the expected result as depicted in Figure 9 (the used OS does not support more than 8 cores; from the point of view of the QSP, using up to 128 cores would have been feasible). This shows very good how quickly the speed up reaches saturation, even though there are 150 tasks that could work in parallel.

To support NUMA, the QSP only needs to be tweaked a little. An additional memory space was inserted between the CPU and the original phys_mem memory space into which another memory space can be inserted parallel to the original phys_mem into which whatever local memory (RAM, ROM, Flash) can be added (Figure 10).

This construct allows attaching distinct and independent timing models to the local memory spaces of the cpus (lmem) and the global memory space (phys_mem), such that the timing models do not need to do any kind of address analysis and can remain oblivious of the actual memory layout, so that their implementation is as simple as shown in Code 8.

"This construct allows attaching distinct and independent timing models to the local memory spaces of the cpus (lmem) and the global memory space…"

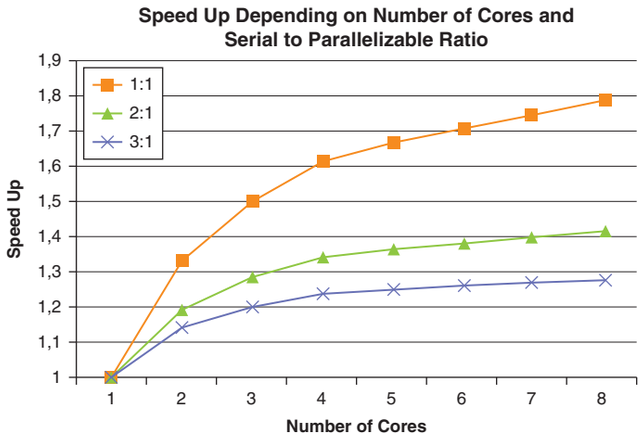

Figure 9: Amdahl's law as observed in a simics simulation. (Source: Wind River, 2013)

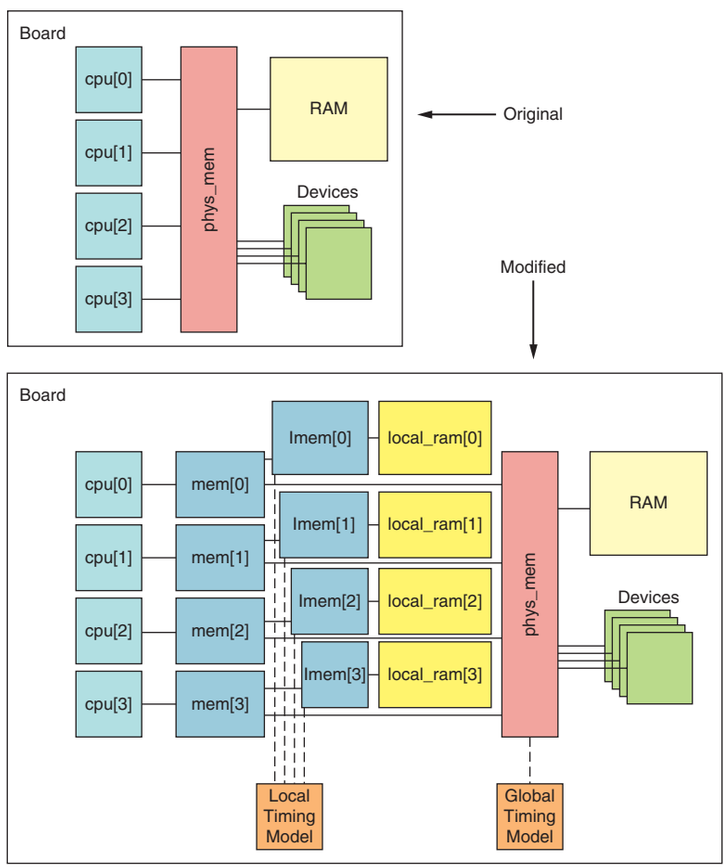

Figure 10: Original and modified memory space hierarchy in the Quick Start Platform to support NUMA setups (Source: Wind River, 2013)

```
import pyobj


class memorySpaceTiming (pyobj.ConfObject):

 def _initialize(self):
      pyobj.ConfObject._initialize(self)

 class accessDelay(pyobj.SimpleAttribute(
           None,
           type='i',
           attr=simics.Sim_Attr_Required)):
  """Delay for transactions"""
  pass

 class timing_model(pyobj.Interface):
  def operate(self,mem_space, map_list, mem_op):
    delayValue=self._up.accessDelay.val
    SIM_mem_op_ensure_future_visibility(mem_op)
    mem_op.reissue=0
    SIM_log_info(2, self._up.obj, 0,
     "Delaying mem op for %d cycles."%(delayValue))
    return delayValue
```

Code 8: Timing model implemenation.
Source: Wind River, 2013

The Simics script that modifies the memory space hierarchy is shown in Code 9. Note that the timing models are not attached by default, but only when calling the Python function enable_delays. This allows booting the OS much quicker. The timing models are only attached when executing the applications that make use of NUMA and that need to be analyzed.

"…timing models are only attached when executing the applications that make use of NUMA…"

```
add-directory "%script%"
run-command-file "%simics%/targets/qsp-ppc/qsp-vxworks.simics"

#load timing model definition
run-python-file "%script%/memspacetiming.py"


#A helper function to create a local ram
@def create_local_ram(prefix, index, size):
 img=SIM_create_object('image',
    prefix+'.local_ram_img[%d]'%(index),
    [['size',size]])
 ram=SIM_create_object('ram',
    prefix+'.local_ram[%d]'%(index),
    [['image',img]])
 return ram
```

```
#Create two timing models,
#one for the local memory access delays
@lDelayer=SIM_create_object('memorySpaceTiming',
        simenv.system+'.ldelayer',
        [['accessDelay',5]])

#The other for the global memory access delays
@gDelayer=SIM_create_object('memorySpaceTiming',
        simenv.system+'.gdelayer',
        [['accessDelay',25]])
@phys_mem=SIM_get_object(simenv.system+'.phys_mem')

#modify memory space hierarchy
@for i in range(simenv.cpu_cores):
 cpu=SIM_get_object(simenv.system+'.cpu[%d]'%(i))
 ram=create_local_ram(simenv.system, i, 1024*1024)
 mem=SIM_create_object('memory-space',
     simenv.system+'.mem[%d]'%(i),
     [['default_target',[phys_mem,0,0,phys_mem]]])
 lmem=SIM_create_object('memory-space',
     simenv.system+'.lmem[%d]'%(i),[])
 cpu.physical_memory=mem
 mem.map.append([0x40000000,
             lmem,
             0,
             0,
             ram.image.size,lmem])
 lmem.map.append([0x0, ram,0,0,ram.image.size])
 lmem.queue=cpu

#function to enable timing models
@def enable_delays():
 SIM_run_command('dstc-disable; dstc-enable')
 phys_mem.timing_model=gDelayer
 for i in range(simenv.cpu_cores):
  lmem=SIM_get_object(simenv.system+'.lmem[%d]'%(i))
  lmem.timing_model=lDelayer
```

Code 9: The DML code for the training device.
Source: Wind River, 2013

"Executing the instructions shows that the timing models are active…"

Figure 11 shows the setup in action. Initially, the timing models were not attached and hence steps and cycles are in sync (due to using a steps-per-cycle ratio of one to one). Afterwards the delays and logging output from the timing models were enabled, and then a check is performed that the next instruction will execute memory accesses (r13 points to global RAM, while r14 points to local RAM). Executing the instructions shows that the timing models are active, and that afterwards steps and cycles are out of sync, because the global

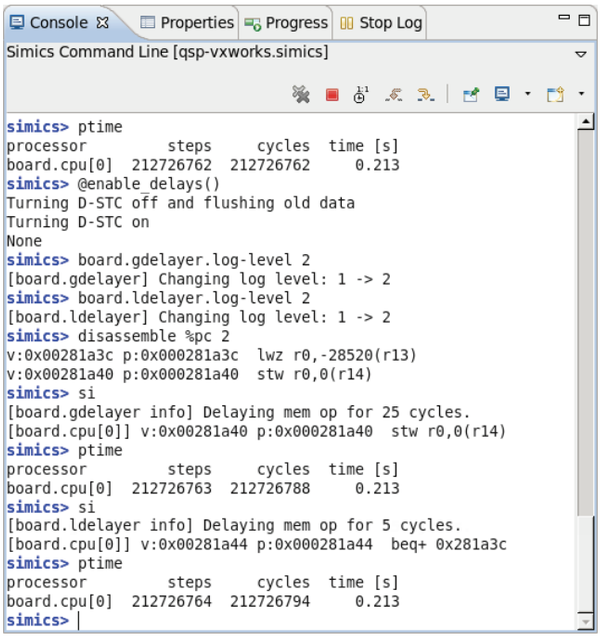

Figure 11: Main and local memory access with active timing models.
(Source: Wind River, 2013)

memory access lasted for 26 cycles (1 cycle execution + 25 cycles transaction delay), and the local access only for 6 cycles.

This enables the creation of labs that can show how NUMA can speed up applications, provided most of the data accesses of each core remain core local and only few accesses need to go to the global RAM. A good example for an algorithm that greatly benefits from NUMA is a parallel quick sort.

"This enables the creation of labs that can show how NUMA can speed up applications…"

Another important aspect to show in multi-core scenarios is synchronization mechanisms and problems. When using Simics, one has to be careful setting the CPU switch time right, in order to show that. If the time quantum of a processor is long enough to allow it to enter and exit the critical (but not properly synchronized) code sections, no problem will occur. One has to make sure that the quantum of a given CPU expires at least once right in the middle of the critical code section, such that another CPU will execute and enter its critical section as well. Figure 12 depicts that.

Due to this, labs that deal with synchronization need to be carefully designed. Nevertheless, choosing the right synchronization problem and CPU switch times makes it possible to show all major synchronization problems, ranging from simple mutex/semaphore-based synchronizations up to labs within perfectly valid uni-processor synchronizations (like using IRQ locks to safeguard critical sections) fail as soon as the number of available cores goes beyond one.

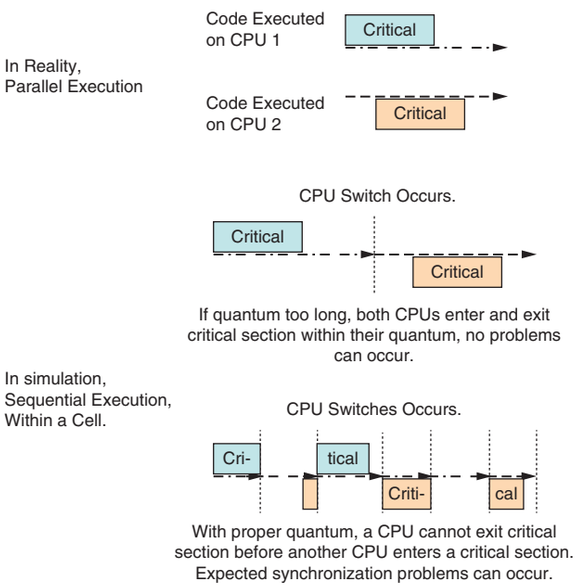

Figure 12: Effect of chosen quantum on observable synchronization problems.
(Source: Wind River, 2013)

There is, however, the drawback that, with Simics, showing cache-coherence protocols in action or cache coherency issues is not trivially possible, because, by default, Simics abstracts from caching for the sake of simulation speed. Then again, cache coherence is generally handled by hardware and is invisible to the software running on the CPUs. Since the training sessions focus on embedded software topics and tools, not being able to actually show cache coherence is no major concern. It should be noted, however, that Simics allows the implementation of cache models and their addition into the existing target system. However, this would be too much effort for the above-mentioned training example for showing something that is not of major interest to the students.

In summary, for multi-core training sessions, Simics allows a broad range of multi-core labs on an off-the-shelf laptop. Not only does this reduce the amount of equipment to ship to a single laptop per student, it also allows you to create labs that could not be done, or only in a limited way, with real hardware.

"…it also allows you to create labs that could not be done, or only in a limited way, with real hardware."

Reference

[1] Combs, Gerald, et al. 2013. Wireshark. www.wireshark.org

Author Biography

Robert Guenzel holds a PhD in computer science from the Technical University of Braunschweig. He joined Wind River in 2011. He is responsible for Simics training development and supports Simics integration into other training programs. Internet address: http://education.windriver.com

SIM-O/C: AN OBSERVABLE AND CONTROLLABLE TESTING FRAMEWORK FOR ELUSIVE FAULTS

Contributors

Tingting Yu
University of Nebraska-Lincoln

Witawas Srisa-an
University of Nebraska-Lincoln

Gregg Rothermel
University of Nebraska-Lincoln

As modern software systems become more complex, they increasingly contain classes of faults that are not effectively detected by existing testing techniques. For example, concurrency faults such as data races tend to be intermittent and require precise execution interleavings or specific events to occur. A common approach to test for data races involves repeatedly executing a program in the hope of exercising an execution interleaving that causes races. Unfortunately, occurrences of races do not always lead to erroneous outputs. As such, they often elude traditional testing approaches that rely on output-based oracles for fault detection. To effectively test for these elusive faults, engineers need to be able to observe the occurrences of faults by precisely controlling various execution events that can cause such faults, and properly monitoring outputs.

In this article, we introduce Sim-O/C, a framework that provides engineers with the observability and controllability necessary to detect elusive faults. The framework is based on the Simics* Virtual Platform but is equally applicable on other virtual platforms and full system simulators. The framework allows engineers to detect runtime violations as they occur. An engineer can choose to let violations propagate to determine whether they can result in failures. In addition, the framework allows engineers to control occurrences of nondeterministic events including thread or process scheduling, system calls, software signals, and hardware interrupts. We provide in-depth technical and implementation details about this framework and then illustrate how the framework can be used to detect intermittent concurrency faults that occur between applications and interrupt handlers.

Introduction

The basic characteristics of modern computer systems are that they utilize multiple CPUs, connect to a large array of peripheral devices, and sense their surroundings through various sensors and actuators. Such complexity makes software development challenging as developers must rely on concurrent programming and various combinations of interrupts, system calls, and polling to fully utilize processing power and to timely sense their environments. It is therefore not surprising that software running on these systems can suffer from classes of "elusive" faults including various forms of concurrency faults that are difficult to detect, isolate, and correct.

"Concurrency faults are difficult to detect because they occur intermittently. To combat these faults, engineers require observability and controllability."

Concurrency faults are difficult to detect because they occur intermittently. To combat these faults, engineers require observability and controllability. Observability is needed to detect when violations occur and assess their implications for correctness. Existing techniques for testing for concurrency

faults often reveal faults by producing observable incorrect outputs. However, the absence of observable incorrect outputs does not mean that there are no concurrency faults in the program. For example, when a thread or process wins a race but simply writes the same data value written by another thread or process, there is no observable incorrect output despite the presence of that race. However, the same race can reoccur later on in a case in which the overwritten value is not the same as the previous value. In this case, erroneous output can be observed. For engineers to effectively test for data races, they must be able to observe the actual races as they occur and not rely solely on the presence of incorrect outputs. This is a primary reason why there have been many instances of concurrency faults that remain dormant during testing and debugging periods and then appear during system deployment.[1][2]

"For engineers to effectively test for data races, they must be able to observe the actual races as they occur and not rely solely on the presence of incorrect outputs."

In addition, when testing for concurrency faults, controllability is needed to force a particular event under test (for example, a specific thread/process interleaving, an interrupt, or software signal) to occur at a precise point in execution so that engineers can observe that event's interaction with other system components. However, engineers often do not have control over execution interleavings. Thus, existing testing techniques often require engineers to repeatedly execute a program, in the hope that a particular execution interleaving that can reveal faults will occur. This approach, however, relies largely on chance. This problem becomes even more difficult in the context of an application's interaction with asynchronous external events. In the example we present in the next section, engineers need to be able to force an interrupt to occur at a particular location in a program. Unfortunately, existing approaches for randomly forcing interrupts are not powerful enough to support such a precise requirement. Even when random interrupts do expose faults, they may miss faults that can occur due to other execution interleavings.

"In addition, when testing for concurrency faults, controllability is needed to force a particular event under test (for example, a specific thread/process interleaving, an interrupt, or software signal) to occur at a precise point in execution…"

It is worth noting that to date, there have been many techniques proposed for detecting thread-level concurrency faults with the aid of observability[3] and controllability[4]. However, these approaches have rarely been applied to scenarios in which concurrency faults occur due to asynchronous events (such as interrupts and software signals) and interleavings at the process-level. It is unclear whether these approaches can work in such scenarios for two reasons. First, controlling these events requires fine-grained execution control; that is, it must be possible to control execution at the machine code level or kernel level and not at the user-mode application level, which is the granularity at which many existing techniques operate. Second, occurrences of events like interrupts are highly dependent on hardware states; that is, interrupts can occur only when hardware components are in certain states. Existing techniques are often not cognizant of hardware states.[5]

In this article, we present Sim-O/C, a virtual machine based framework designed to tackle the challenges of testing for broad classes of elusive faults such as concurrency faults. Particularly, Sim-O/C can achieve the levels of observability and controllability needed to test systems by utilizing the virtual platform's abilities to interrupt execution without affecting the state of the

"…Sim-O/C is able to stop execution at a point of interest and force a traditionally nondeterministic event to occur. Sim-O/C then monitors the effects of the event on the system and determines whether any anomalies result."

"We have designed Sim-O/C to overcome deployment obstacles by implementing it on a commercial virtual platform, Simics.[6]"

virtualized system, to monitor function calls, variable values and system states, and to manipulate memory and buses directly to force events such as interrupts and traps to occur. As such, Sim-O/C is able to stop execution at a point of interest and force a traditionally nondeterministic event to occur. Sim-O/C then monitors the effects of the event on the system and determines whether any anomalies result.

Many existing approaches for detecting concurrency faults are not widely used because they require significant deployment effort. We have designed Sim-O/C to overcome deployment obstacles by implementing it on a commercial virtual platform, Simics.[6] We chose Simics for several reasons. First, similar to other full-system simulators, Simics provides functional and behavioral characteristics similar to those of target hardware systems, enabling software components to be developed, verified, and tested as if they are executing on the actual systems. Second, through a rich set of Simics APIs, software engineers have the ability to unobtrusively observe and control various system behaviors without needing the source code. Third, due to its powerful device modeling infrastructure, Simics already plays a critical role in hardware/software co-design; therefore, adding the proposed capabilities to Simics enables adoption without requiring undue effort. Thus, we envision that Sim-O/C will allow several aspects of product integration testing to be moved up to the co-design phase of system development. Fourth, licensing of Simics is free for academic institutions, making it a good platform for research.

The remainder of this article is organized as follows. The section "A Motivating Example" provides an example that is used to explain the operation of Sim-OC. This is followed by the section "Overview of Sim-O/C." The section "Implementation Details" describes the implementation of Sim-OC on Simics. The section "Evaluation" reports the results of an evaluation in which Sim-OC is used to detect concurrency faults due to interrupts and signals. "Further Discussion" presents the ramifications of our results. The "Conclusion" section discusses future work and concludes this article.

A Motivating Example

"This particular fault existed in the source code for over three years before being eradicated."

In early releases of version 2.6 of the Linux kernel, there is a particular data race that occurs between the *serial8250_start_tx* routine and the UART *serial8250_interrupt* interrupt service routine (ISR) in the UART driver program.[7] This particular fault existed in the source code for over three years before being eradicated. We provide the code fragments that illustrate the error in Code 1.

```
static int serial8250_startup(struct uart_port *port){
1.    up = (struct uart_8250_port *)port;
2.    ...
/* Do a quick test to see if we receive an
       * interrupt when we enable the TX irq. */
3.    serial_outp(up, UART_IER, UART_IER_THRI);
```

```
4.   lsr = serial_in(up, UART_LSR);
5.   iir = serial_in(up, UART_IIR);
6.   serial_outp(up, UART_IER, 0);
7.   if (lsr & UART_LSR_TEMT && iir & UART_IIR_NO_INT) {
8.         if (!(up->bugs & UART_BUG_TXEN)) {
9.               up->bugs |= UART_BUG_TXEN;
10.        }
11.  }
12.  ...
     /* Finally, enable interrupts.  */
13   up->ier = UART_IER_RLSI | UART_IER_RDI;
14   serial_outp(up, UART_IER, up->ier);
15   ...
16.}

static void serial8250_start_tx(...){
17.  serial_out(up, UART_IER, up->ier);
18.  ...
19.  if (up->bugs & UART_BUG_TXEN) {
20.        ...
21.        transmit_chars(up);
22.  }
23.}

static irqreturn_t serial8250_interrupt(...){
24.  ...
25.  transmit_chars(up);
26.  ...
27.}

static void transmit_chars(...){
28.  struct circ_buf *xmit = &up->port.info->xmit;
29.  ...
30.  xmit->tail = (xmit->tail + 1) & (UART_XMIT_SIZE - 1);
31.  ...
32.}
```

Code 1. Faulty code that can cause data races in the UART driver in Linux. Source: University of Nebraska-Lincoln, 2013

"…Faulty code that can cause data races in the UART driver in Linux."

Routine serial8250_startup is responsible for testing and initializing the UART port and assigning the ISR. This routine is called before the UART port is ready to transmit or receive data. Routine serial8250_start_tx is used to initialize data transmission, and is called when data is ready to transmit via the UART port. Routine serial8250_interrupt is the actual ISR, and is called to perform data transmission when: (1) the data is ready to be transmitted; and (2) the interrupt enable register (IER) is enabled by the serial8250_start_tx routine.

Under normal operating conditions, the ISR is always responsible for transmitting data. To ensure that an ISR is assigned correctly, serial8250_ startup issues an interrupt and monitors the response from the ISR (line 3–6 in Code 1). Several sources have shown that problems such as races with other processors on the system or intermittent port problems can cause the response from the ISR to get lost or cause a failure to correctly install the ISR, respectively. When that happens, the port is registered as "buggy" (line 9) and workaround code based on polling instead of using interrupts is used (lines 19–22). Unfortunately, the enabled interrupts (lines 13–14) are not disabled in the workaround code region so by the time the workaround code is executed, it is possible for both the ISR and the workaround code to transmit or receive data through the same serial port at the same time by calling the routine transmit_chars, and simultaneously read from or write to the shared variale xmit->tail (line 30). As such, a race in this illustration occurs when:

"…a race in this illustration occurs when:

the device driver program is preempted by the ISR after a shared memory access before it can proceed to the next instruction;

the ISR manipulates the content of this shared memory."

1. the device driver program is preempted by the ISR after a shared memory access before it can proceed to the next instruction;
2. the ISR manipulates the content of this shared memory.

Higashi et al.[8] introduce an approach to test for this fault by controlling invocations of interrupts. In that work, they used an ARM-based processor simulator and modified version of uCLinux with the same fault that could run on that simulator. Their modifications included porting the code from PPC to ARM and removal of irrelevant code to reduce the simulation time. Their methodology involves invoking an interrupt at every memory read and write operation. We recreated a similar testing system based on Sim-O/C with two additional optimization techniques beyond those used in the Higashi approach. In the first optimization technique, we apply static program analysis to detect the resources that can be affected by the UART driver and the ISR. With this optimization, we invoke interrupts only when these shared resources are accessed, resulting in a smaller number of interrupts. Second, we also check system states at runtime to ensure that it is possible to invoke interrupts when those resources are accessed, resulting in more realistic interrupt invocations. These two optimizations should significantly reduce the time required to conduct testing.

Overview of Sim-O/C

Currently, Sim-O/C is implemented for applications running on x86/Linux environments. The framework includes four major components, which interact with Simics as user-developed tools:

- A configuration repository stores initialization scripts containing information that includes execution breakpoints and variable locations that must be observed.
- An execution observer is an external module that can be attached to Simics. It monitors runtime information and then records it in a file, or directly sends it to online test oracles to predict whether a fault occurs.

- An execution controller is another external module that can be attached to Simics. It invokes callback functions when events of interest occur (such as interrupts and memory read/write operations).
- An oracle repository stores test oracle files in the form of property requirements.

To date, it is quite common for certain types of faults to be detected using code-level instrumentation. Such faults include thread-level concurrency faults, memory leaks, and buffer overflows. However, code-level instrumentation has been known to introduce probe effects. Furthermore, faults that occur across applications in a system can make code level instrumentation infeasible (for example, if only binary code is available or instrumentation tools are not compatible with development languages). Sim-O/C operates at the binary level so it is language independent and does not rely on external tools to perform instrumentation. It also minimizes probe effects as instrumentation is performed outside of virtual execution states. Controllability is achieved by allowing engineers to initiate the desired events at particular execution points during testing.

"Sim-O/C operates at the binary level so it is language independent and does not rely on external tools to perform instrumentation."

In addition, Sim-O/C is capable of detecting the following three classes of faults that cannot be easily detected without the support of virtual platforms:

- concurrency faults between applications and hardware interrupts, including both data races and deadlocks;
- concurrency faults caused by improper shared resource access among multiple processes;
- concurrency faults caused by incorrect arrivals of software signals.

As such, the ultimate benefit of Sim-O/C is its ability to test for classes of concurrency faults that cannot be effectively detected by existing approaches. In the future, we envision that Sim-O/C can be used to test for other elusive faults including temporal violations that occur due to interrupts, and improper resource usage. For example, Sim-O/C can be used to generate complex interrupt sequences to test for violations of expected worst-case interrupt latencies.

"…the ultimate benefit of Sim-O/C is its ability to test for classes of concurrency faults that cannot be effectively detected by existing approaches."

In the next section, we describe each component of Sim-O/C and illustrate how Sim-O/C can be used to test for data races that occur due to improper arrivals of hardware interrupts. This type of fault has been identified as one of the most "nasty" faults to test for in embedded software.[9] When we conduct testing for data races, the components under test include the main application, device drivers, and the ISRs that are associated with the device drivers. The focus of our illustration is testing for races that occur when the application coupled with the device drivers interact with an ISR.

Implementation Details

Figure 1 depicts the overall architecture of the Sim-O/C framework. There are four major components in the framework in addition to Simics itself. As stated earlier, Simics provides APIs that can be accessed via Python scripts; thus, all components except the test oracles are Python scripts.

In the example we are considering (from the section "A Motivating Example"), the program under test (PuT) includes an application that interacts with a serial port and the UART driver. We refer to the interrupt service routine for the tested UART port as the ISR.

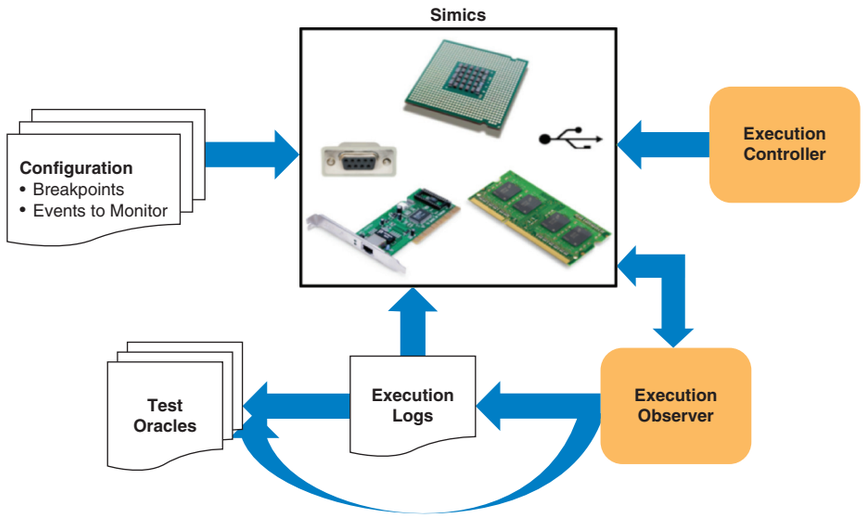

Figure 1: Overview of the SimTester architecture. (Source: University of Nebraska-Lincoln, 2013)

Test Configuration

The first component in our framework is the Configuration script, the content of which includes information such as locations at which to set execution breakpoints on the points of interests. We can identify such points through static or dynamic analysis (for example, we can set breakpoints at instructions that access shared resources).

"…the Configuration script sets breakpoints on the events of interest so that the Execution Observer and Execution Controller modules can use the monitored runtime information."

To test for data races in our example, the Configuration script sets breakpoints on the events of interest so that the Execution Observer and Execution Controller modules can use the monitored runtime information. In this example, the information generated by the Execution Observer includes:

- information on when functions of the PuT and ISR execute and when they return, and
- information on when SVs are accessed by the PuT and written by the ISR.

Since it is too expensive to monitor each variable access, we choose to monitor only variables shared between the PuT and the ISR. We use the precise shared variable detection algorithm proposed by Kahlon et al.[10], but we are interested only in shared variables that are read by the PuT and written by the ISR, or written by both the PuT and the ISR. We label each shared variable as a "definition" or "use" through our analysis.

One of our configuration tasks is to set breakpoints in Simics to detect when an ISR executes and returns, and when SVs are accessed. The *BasicConfig* algorithm (Algorithm 1) describes the steps for setting breakpoints.

```
procedure BasicConfig()

1:  begin

2:  for each shared variable

3:  set read/write breakpoint

4:  endfor

5:  set execution breakpoint on entry instruction of interrupt handler

6:  set execution breakpoint on interrupt return instruction iretd

7:  end
```

Algorithm 1

Execution Observer

The Execution Observer monitors and generates information that is used by the Execution Controller and Test Oracles. The generated information can either be recorded in a log file (to support offline analysis) or directly taken from the observer (to support online analysis). For offline analysis, the callback functions log events into a runtime trace. This runtime trace serves two purposes. First, it provides information for the Execution Controller. The Execution Controller can determine the locations at which events should occur in the following test executions. Second, Test Oracles can use the log file to determine whether a fault occurs. For online analysis, the Execution Observer notifies the Execution Controller as to when or where to cause an event to occur in the current run. As in the offline analysis, runtime information can be used directly by the oracle components for dynamic fault detection.

"The Execution Observer monitors and generates information that is used by the Execution Controller and Test Oracles."

An important feature of the Execution Observer is that it can ignore irrelevant events. Typically, engineers are interested only in the program under test and not other programs running in a system. When using the raw callback information, the Execution Observer cannot distinguish between different execution units such as processes or threads (for example, a shared memory address specified in the *BasicConfig* that can be accessed by other programs not under test.) In most cases, we can overcome this ambiguity by using a *process tracker* provided by Simics. A process tracker can distinguish each user-space process; as such, engineers are able to focus on the program under test.

"An important feature of the Execution Observer is that it can ignore irrelevant events."

The process tracker is not operational when a program executes in kernel mode, so the process tracker alone cannot help us test for faults occurring between applications and kernel components such as the UART device driver mentioned earlier. To overcome this problem, we created an algorithm

(Algorithm 2) to isolate the PuT and ISR from other applications in the system or other ISR invocations. Using a static analysis tool, we identify all function names in the PuT and their entry addresses. By parsing the symbol tables, we can identify the entry address to the ISR. Furthermore, we monitor the function return instruction (ret in X86) to determine whether a function or the ISR has returned, and we monitor the interrupt return instruction (iretd in X86) to determine whether the PuT has recovered from the interrupt context.

"This mechanism allows us to ignore those shared variables that might be accessed by a different ISR or a different program on the same ISR."

At runtime, we keep a call stack named func_list. When a function or an ISR from PuT is invoked, its <address, frame pointer, stack pointer> is added to func_list (line 3). When a shared variable is accessed, we compare the current frame pointer *ebp* with the frame pointer on top of func_list (line 24). This mechanism allows us to ignore those shared variables that might be accessed by a different ISR or a different program on the same ISR. If a ret instruction is encountered, by comparing the current stack pointer *esp* with the stack pointer on top of func_list (line 9), we can determine whether the current function or the ISR has returned.

A function is popped from func_list if its ret instruction is reached. Program counter pc is recorded twice to determine whether the PuT is actually preempted between a shared variable access and its following instruction. The first recording time occurs when a shared variable is accessed (line 26) in a non-interrupt service routine context, and the second time occurs after an interrupt returns (line 20). An interrupt return instruction iretd is recorded to indicate termination of an interrupt context. Note that the mere presence of an iretd does not imply that an interrupt will jump back to the PuT, because more than one device can issue interrupts and call iretd instructions. To overcome this problem, an iretd is logged only when its frame pointer is equal to the frame pointer when a shared variable is accessed in the PuT (line 18).

```
procedure RaceObserver()

require: procedure BasicConfig()

1: switch (breakpoint)
2:    case function addr:
3:          func list.push({func addr, ebp, esp})
4:          if func addr == ISR entry
5:                log "ISR entry"
6:                is_ISR = true
7:          endif
8:    case ret:
9:          if esp == func_list.top[index esp]
10:               if func list.top[index f unc] == ISR entry
11:                     is_ISR = false
12:                     log "ISR exit"
13:               endif
14:               func list.pop()
15:         endif
```

```
16:  case iretd:
17:       if ebp == ebp_switch
18:            log “iretd”
19:            /*log program counter in next instruction*/
20:            next pc()
21:       endif
22:  case SV:
23:       /*check if SV is accessed by the PuT*/
24:       if ebp == func list.top[index ebp]
25:            if is_ISR == false
26:                 log “PuT”, SV ,SVaccess, pc
27:                 /*save Reg[ebp] content*/
28:                 ebp_switch = ebp
29:            else /*interrupt handler context*/
30:                 if SVaccess == write
31:                           log “ISR”, SV , SVaccess
32:                 endif
33:            endif
34:       endif
35: end
```

Algorithm 2

In summary, events logged for testing race conditions include: (1) read/write accesses to shared variables (an SV access by the PuT); (2) entry to the ISR; (3) a write to an SV by the ISR; (4) return from the ISR; and (5) context switches from the ISR to the PuT. Code 2 illustrates a sample of trace information recording these events for this example.

```
...
PuT, $xmit->tail$, read, pc1
ISR entry ISR, $xmit->tail$, write
ISR exit
IRETD
pc1+1
...
```

Code 2.

Source: University of Nebraska-Lincoln, 2013

Note that there is a race in the trace given in Code 2. By observing the program counter when an SV is accessed by the PuT and the interrupt recovery point, we can determine that an interrupt occurs right after xmit->tail is read by the PuT. Later on, this race can result in a program failure when the PuT reads the wrong value of xmit-tail.

“Note that there is a race in the trace given in Code 2.”

“…the Execution Controller. This module specifies certain events that should be invoked at particular points…”

Execution Controller

The third component in our framework is the Execution Controller. This module specifies certain events that should be invoked at particular points

during executions of the PuT. First, the Execution Controller can artificially create I/O interrupts simply by writing data to the I/O bus or memory locations that have been mapped to hardware devices. Simics allows us to issue an interrupt on a specific IRQ line from the simulator itself. The interrupt will happen before the subsequent instruction. Second, the Execution Controller can force the system to execute a particular exception handling routine by artificially creating that exception, such as by configuring the system environment into an error state. Third, the Execution Controller can force the system to execute a particular path by specifically setting a desired branch condition in a hardware register or a memory location. The logical address of a global variable can be obtained by parsing the symbol table. Thus engineers are able to artificially inject values into the memory address to set the value of this variable. Note that in order to write to program variables, the logical virtual address must be converted to a physical address. As for the hardware register, the *cpu.iface.int_register.write* API is used to manually set the content of a register. Finally, the Execution Controller can cause an interrupt and use the interrupt handler to control kernel-level scheduling. This is an important feature of the Execution Controller when events need to be precisely controlled at the kernel level. For example, to test for concurrency faults among multiple user processes, Sim-O/C first observes the correct execution of two processes and identifies system calls that can potentially race with each other (for example, two processes that both write to a file using the *write* system call). Next, Sim-O/C controls the scheduling of the two processes and attempts to switch the order of the system calls in the original execution. If such a switch causes a failure, a concurrency fault is detected.

"…the Execution Controller can cause an interrupt and use the interrupt handler to control kernel-level scheduling. This is an important feature of the Execution Controller when events need to be precisely controlled at the kernel level."

Note that existing approaches would likely use functions such as *yield* or *sleep* to achieve some form of execution control in user-level applications. However, such approaches are not precise, as these functions do not guarantee when the suspended process will be executed again. In addition, they do not work for kernel-level programs such as device drivers. Instead, we use the Device Modeling Language (DML)[11] to achieve more precise execution control. Our approach creates a dummy device for the platform running on Simics, and installs its associated IRQ line in the ISA bus. In this way, whenever a process is ready to be stopped or resumed, the interrupt of this device is invoked. The associated interrupt handler serves as an event handler. The handler first checks the state of the process (suspending, active, and so on) using the *task_struct* data structure. Next, the handler controls the process scheduling using process scheduling APIs (such as *wake_up_process()*). This type of controllability can also be used to control software signals.

In the example that we are considering, when engineers enable the controller module *RaceController*, a controlled interrupt is invoked immediately after a shared variable access by the PuT.

Because our framework forces interrupts to occur, we would like to distinguish between interrupts that occur naturally as part of program execution and forced

interrupts issued by the Execution Controller. We refer to the former interrupts as *self-generated*, and to the latter as *controlled*.

It is not realistic to invoke an interrupt at an arbitrary program point. The interrupt enable register and possibly other control registers must be set to enable interrupts. In the example we are considering, before invoking an interrupt, the interrupt enable register IER of the UART must be set while the interrupt identification register IIR must be cleared. Even if interrupts are enabled, they can be temporarily disabled. The following routine is the routine in *RaceController* used to determine whether it is possible to issue an interrupt.

```
procedure ISR enabled(int p)
/*p is the pin number for a certain interrupt*/
1: begin
2: if eflags[9] != 0 and ioapic.redirection[p] == 0
   and ioapic.pin raised[p] == LOW
3:    return true
4: else
5:    return false
6: endif
7: end
```

There are two general steps that our system takes prior to invoking a controlled interrupt. First, the Execution Controller checks the status of the local interrupt and global interrupt bits to see if interrupts are enabled. In an X86 architecture, the global interrupt bit is the ninth bit of the eflags register (line 2 of ISR enabled). When this bit is set to 1, the global interrupt is disabled, otherwise it is enabled. For local interrupts, Simics x86 targets use the Advanced Programmable Interrupt Controller (APIC) as their interrupt controllers. As such, our system checks whether the bit controlling the UART device is masked or not. Our system also checks whether a self-generated interrupt is about to be issued by examining the current pin status. If this is true, the controlled interrupt will not be invoked.

Note that in the foregoing discussion we have assumed that interrupts are not nested, but our algorithms do also support nested interrupts. Also note that in our discussion we have considered only the presence of a single ISR, but our algorithm can be generically applied to handle multiple ISRs.

Test Oracle

The fourth component in our framework is the Test Oracle. Test oracles can be "online" or "offline". Online test oracles detect violations at runtime and issue error messages then. For example, Sim-O/C can be used to dynamically monitor for the occurrences of deadlocks by maintaining a resource allocation graph. Each time a shared resource is accessed by the PuT, Sim-O/C updates the graph and checks whether there is a cycle in the graph. Offline test oracles analyze runtime log files after program execution and report them then. Each log file is compared against an oracle specification to detect a particular type of

"Test oracles can be "online" or "offline". Online test oracles detect violations at runtime and issue error messages then."

"Offline test oracles analyze runtime log files after program execution and report them then."

anomalous execution behavior. For example, in the example discussed earlier, the oracle can specify in which condition data races occur. The runtime trace (Code 2) is used to compare against the specification.

Test Driver

In addition to the components illustrated in Figure 1, a test driver is also needed to automate the testing process. Typically, engineers conduct testing by running test programs on a system. A test driver is a program that automates the process of running test programs in a suite, managing the runtime violation detection and analyzing generated log files.

"A test driver is a program that automates the process of running test programs in a suite, managing the runtime violation detection and analyzing generated log files."

For example, to test for data races we need to provide two components to the test script: the test input and conditions governing when to invoke interrupts from within the system. In this case, a test case is used as the test input for the PuT (which includes the application and any device driver running under non-interrupt service routine context that is called by the application). Note that test cases for the PuT can be generated based on various criteria. We discuss the criteria we use in the evaluation section. Next, we need to describe each interrupt condition (IC). We express an IC as a tuple: < loc, pin >. The first element, loc, specifies a code location at which to invoke an interrupt. The second element, pin, specifies an Interrupt Request (IRQ) line number at which to invoke the interrupt. This is needed because typically, an interrupt service routine can be associated with multiple IRQ lines. ICs are used only when the controllability module is enabled.

"Our system invokes only one controlled interrupt per test run. This is done to avoid fault masking effects…"

Our system invokes only one controlled interrupt per test run. This is done to avoid fault masking effects, which may occur in cases where multiple interrupts fire and cause a failure that would be evident in the presence of a single interrupt to be "masked" by the presence of the second.[14] Thus, our system needs to first check a flag to determine whether a controlled interrupt has already been invoked in the current run. If it has, the test system does not monitor any further events in this run. Once it has been determined that there has not been any invocation of a controlled interrupt in this run, the system then checks to see whether the last accessed SV has already been tested in prior runs. If it has not, the system enables the control register for UART and then calls the simple interrupt API.

Note that, given the above approach, with the controller module disabled, the PuT runs |tc| times during a testing process, where |tc| is the number of test cases.

However, with the controller module enabled, there can be multiple runs of each test case. This is because each test case execution may encounter different numbers of SVs. Therefore, the number of runs depends on the number of SVs that must be tested, and is given by the equation

$$|tc| \times (|int| + 1)$$

where |tc| is the number of test cases and |int| is the number of controlled interrupts issued. We also need to run the PuT one additional time for each test case to determine whether all SVs have been accessed.

Evaluation

To evaluate Sim-O/C for race detection related to interrupts, we applied it to the UART device driver on a preemptive kernel version of Fedora Core 2.6.15. The driver includes two files, serial core.c and 8250.c, containing 1896 and 1445 lines of non-comment code, respectively. The main application transmits character strings to and receives character strings from the console via the UART port. In this article we apply our testing process only to the UART driver; however, the same process is also applicable to other types of device drivers.

"To evaluate Sim-O/C for race detection related to interrupts, we applied it to the UART device driver on a preemptive kernel version of Fedora Core 2.6.15."

Our approach requires the use of existing test cases, so we generated test cases for the system based on a code-coverage-based test adequacy criterion. After SVs were identified (see the "Test Configuration" section), we generated a set of test cases that cover the feasible SVs (SVs for which there exists a possible execution of the program which executes them) in the PuT. This process produced 12 test cases.

To better assess the cost and effectiveness of our approach, we considered both the approach and two alternative baseline approaches. In the discussion that follows, we refer to our approach as the "conditional controllability approach," because it involves issuing controlled interrupts under certain conditions. The second approach that we considered, "no controllability," involves testing the program without any controlled interrupts; this is the approach that test engineers normally use. The third approach that we consider, "random controllability," involves issuing controlled interrupts at random program locations after shared variable accesses and without checking interrupt conditions.

"To better assess the cost and effectiveness of our approach, we considered both the approach and two alternative baseline approaches."

We measured execution times for the foregoing approaches by embedding a timer in the Simics module. The reported times are the actual times spent by Simics to execute the program.

Testing for Race Conditions

We begin by considering the target program as given, and evaluate the effectiveness and efficiency of our race condition testing approach on that program. We summarize the results in Table 1.

	Conditional	None	Random
Test runs	352	6,000	288
Interrupts	96	16,500	1,044
Time (sec)	77.91	74.08	75.17
Race detected	Yes	No	No

Table 1: Summary of Results
(Source: University of Nebraska-Lincoln, 2013)

We first applied conditional controllability together with observability. Under this approach, across the 12 test cases utilized, 84 controlled interrupts were

"In the course of applying the approach, we detected a race in function uart_write_room of /linux/drivers/serial/serial_core.c, which we later determined had been corrected in subsequent versions of the system."

applied, and for each test case, one extra run was needed to determine whether all shared variables had been accessed. Thus, 96 test runs were required to finish testing the target program with an average execution time of 77.91 seconds per test run. Including self-generated interrupts, the number of interrupts generated for the target program was 352. In the course of applying the approach, we detected a race in function uart_write_room of /linux/drivers/serial/serial_core.c, which we later determined had been corrected in subsequent versions of the system. By running the system with observability turned off, we determined that this fault can be detected only with observability enabled; in other words, it is a fault that did not propagate to output on our particular test inputs, but may propagate under a different test input.

We next tested our target program with no controllability. In this case, the only interrupts that occur are self-generated interrupts. Because runs of each given test can conceivably differ, we ran each test on the program 500 times. The total number of interrupts observed was 16,500. Over the 6000 total test runs, average execution time was 74.08 seconds per test, only 3.83 seconds less than with controllability added. None of these test runs detected the race condition detected by our first approach, however, either with observability enabled or disabled.

Finally, we tested our target program using the random controllability approach. For each test case, we ran the target program three times more than the number of runs performed under the conditional controllability approach, on each run generating an interrupt at a randomly selected program location. The total number of test runs was 288 and the number of interrupts generated was 1044. In this case the average execution time per test case was 75.17 seconds, only 2.74 seconds less than with controllability added. Again, the race was not detected, either with observability enabled or disabled.

One important characteristic of our technique is that checking is performed before issuing a controlled interrupt. When a shared variable is accessed in the main program, the controllability module first checks to see whether it is possible to issue an interrupt, and if not, it proceeds to the next possible location. This approach can save test runs, but at the cost of checking. To quantify the tradeoffs involved, we also applied our conditional controllability approach without the checking step enabled. Recall that with checking enabled, 96 test runs were needed to issue controlled interrupts, with an average execution time of 77.91 seconds per test. With checking disabled, on the other hand, 1428 test runs were needed to issue controlled interrupts, with an average execution time of 75.66 seconds per test. Clearly, the checking approach saves time overall.

A second characteristic of our technique is that interrupts are issued only after shared memory accesses, and this can be much less expensive than issuing interrupts after each memory access, which is the approach used by Higashi et al.[8] For our target program, there are 94,941 data accesses made in the course of running the 12 test cases. If an interrupt were issued after each data access, we would need 82.6 days to finish testing the target program.

Fault Detection Effectiveness

While the results of the foregoing study are encouraging, the numbers of naturally occurring faults found in the target program was low, rendering comparisons of the fault detection effectiveness of our approach less meaningful. To further investigate fault detection effectiveness we followed a process often utilized in the software testing research community[12]; namely, the use of seeded faults.

"To further investigate fault detection effectiveness we followed a process often utilized in the software testing research community; namely, the use of seeded faults."

In this case, we injected 12 potential race condition faults into the target program. To do this, we removed statements corresponding to critical section protection (for example, spin lock, spin lock irq). Of the 12 potential faults thus created, further examination revealed that seven of them could not possibly be triggered on the system on its given hardware platform, so we removed those. This left us with five potentially detectable race condition faults.

Given the faults thus seeded, we ran our test cases on the faulty systems using conditional and random controllability, and in the case of race detection, with observability enabled and disabled. For the race condition detection approach, conditional controllability detected two of the five faults. One of these faults was detected both with and without observability. The same fault was also detected with random controllability, but only with observability enabled because in this case the fault does not propagate to output. This occurred because interrupts issued by conditional controllability visited more unprotected shared variables that can cause incorrect output than random controllability did, and these shared variables are not visited by random controllability.

The second fault revealed in our race detection trial was revealed not through observability, but rather, through output, for both conditional controllability and random controllability. The reason this occurred is because the fault was not actually caused by our defined race condition, but rather, by another type of violation known as an "atomicity violation," which we did not specify but can easily detect. In particular, in this case, a read-write SV pair in the main program is supposed to be atomic, but the ISR read this SV before it was updated in the main program. This outcome shows that, while our approach does not specifically target other types of faults, it may catch them as byproducts.

"…while our approach does not specifically target other types of faults, it may catch them as byproducts."

We also inspected the three potential race condition faults that were not detected by any technique. We determined that the reason for their omission was that the interrupt handler in each of the versions does not share variables with the main program, so they are not races. This does not mean that the code regions involved do not need to be protected, because other ISRs may share memory locations, or programmers may intentionally cause the regions to execute without interruption.

Further Discussion

Our observer module considers one type of definition of a race condition. In practice, testers can adopt different definitions because there is not a single

general definition for the class of race conditions that occur between an ISR and a PuT. According to our definition, we consider the case in which the PuT first reads from or writes to an SV, and the ISR modifies this SV during the next access to it. However, our definition does not capture the case in which the ISR reads from the SV. As noted above, for the four faulty versions on which the ISR does not modify the SV, we still found one fault with controllability enabled. This fault is related to an atomicity violation, as a code region in the main program is supposed to execute atomically, such as, for example, before a shared variable is updated in the main program, and an interrupt occurs and the wrong data is read by the ISR.

"Testers can extend our method by forcing interrupt handlers to execute different paths, which may increase the probability of revealing faults."

Our approach injects data into device ports and forces an interrupt handler to execute one path. The data we inject is the same as the test input given to the program. For example, if an application sends the string "hello" to the UART console passed by the UART transmitter buffer, a controllability module would inject "world" into the UART transmitter buffer to force an interrupt to occur after a certain access. It is also possible to have multiple paths by which shared variables can exist in interrupt handlers. Testers can extend our method by forcing interrupt handlers to execute different paths, which may increase the probability of revealing faults. However, no faults are left undetected due to missing shared variables or spin locks in the other paths of the ISR in our target program. It is also possible to force an interrupt handler to execute only the paths that have definition-use relationships with the main program. This may further reduce the number of controlled interrupts and test runs. To do this, the value schedule approach proposed by Chen et al.[13] could be adapted.

To force an interrupt to occur, our controllability module issues a new interrupt. However, races and deadlocks can occur relative to the interrupt generated by the target program itself. For example, suppose an interrupt is requested by device driver code, but is not immediately processed for some reason (for example, due to the occurrence of a device port delay). The interrupt handler associated with this interrupt may be executed later within a spin lock pair or after a shared variable access, and thus a race condition or a deadlock may occur. Our controllability module can be further extended to deal with such cases. For example, when an interrupt is triggered, the module can delay this interrupt by masking its interrupt enable register and issuing the interrupt after a certain event occurs (for example, when a shared variable is accessed or a spin lock is acquired). If there is no such event, the interrupt is issued on exiting the PuT.

In practice, when testing software components (such as device drivers and interrupt handlers), the first task that a test engineer must accomplish is to gain confidence that the software component is developed correctly. In our study, the analysis involves a test program, the interrupt handler that interacts with the device driver, and the device driver code. The key point here is that the tester focuses on a specific component and how it interacts with the rest of the components. If the focus changes to a different component, the same analysis can be applied to test the new component. As such, the proposed approach is

not designed to test the entire system at once. Instead, it is more suitable for component testing.

In our current work, the test generation process was performed manually, which is currently the norm in practice. Our study considers a test input to include input values and interrupt scheduling. However, there is no reason the approach could not also utilize input values created using existing test case generation approaches (such as dynamic symbolic execution[13].) A problem with such approaches by themselves is that they generate large numbers of test cases with no methodology for judging system correctness beyond looking for crashes. Our approach provides more powerful, automated oracles, and thus should ultimately facilitate the use of larger numbers of automatically generated test cases.

"Our approach provides more powerful, automated oracles, and thus should ultimately facilitate the use of larger numbers of automatically generated test cases."

"We have introduced Sim-O/C, a framework that provides test engineers with the ability to precisely control execution events and observe runtime context at critical code locations."

Conclusion

We have introduced Sim-O/C, a framework that provides test engineers with the ability to precisely control execution events and observe runtime context at critical code locations. The framework is built on a commercial virtual platform that is commonly used as part of the hardware/software co-design process.

The main benefit of using virtual platforms for testing is the ability to interrupt execution without affecting the states of the virtualized system. Moreover, precise process scheduling can be implemented by installing a dummy device, which is a significant benefit of using virtual platform. Furthermore, we can monitor function calls, variable values, and system states, as well as manipulate memory and buses directly to make typically nondeterministic execution events more deterministic.

References

[1] Apache Software Foundation. Datarace on org.apache. https://issues.apache.org/bugzilla/show_bug.cgi?id=37458.

[2] Launchpad11. Canonical Ltd. Launchpad: data-races-implementation sql crash. https://bugs.launchpad.net/f-4d-cb/+bug/516622

[3] Michael D. Bond, Katherine E. Coons, and Kathryn S. McKinley. 2010. "PACER: Proportional Detection of Data Races." In Proceedings of the 2010 ACM SIGPLAN Conference on Programming Language Design and Implementation, Toronto, Ontario, Canada, 2010, pp. 255–268.

[4] Koushik Sen. "Race Directed Random Testing of Concurrent Programs." In Proceedings of the 2008 ACM SIGPLAN Conference on Programming Language Design and Implementation, Tucson, AZ, 2008, pp. 11–21.

[5] John Regehr. "Random Testing of Interrupt-driven Software." In Proceedings of the 5th ACM International Conference on Embedded Software, Jersey City, NJ, 2005, pp. 290–298.

[6] Jakob Engblom, Daniel Aarno, and Bengt Werner. "Full-System Simulation from Embedded to High-Performance Systems." Processor and System-on-Chip Simulation, pages 25–45, 2010

[7] I. Jackson, "IRQ handling race and spurious IIR read in 8250.c." https://lkml.org/lkml/2009/3/12/379.

[8] Makoto Higashi, Tetsuo Yamamoto, Yasuhiro Hayase, Takashi Ishio, and Katsuro Inoue. "An Effective Method to Control Interrupt Handler for Data Race Detection." In Proceedings of the 5th Workshop on Automation of Software Test, Capetown, South Africa, 2010, pp. 79–86.

[9] EETIMES. "Five top causes of nasty bugs." http://www.eetimes.com/design/embedded/4008917/Five-top-causes-of-nasty-embedded-software-bugs

[10] Vineet Kahlon, Yu Yang, Sriram Sankaranarayanan, and Aarti Gupta. "Fast and Accurate Static Data-race Detection for Concurrent Programs." In Proceedings of the 19th International Conference on Computer Aided Verification, Berlin, Germany, 2007, pp. 226–239.

[11] Jakob Engblom. "Virtutech Device Modeling Language." Simics White-Paper, 2009

[12] J. H. Andrews, L. C. Briand, and Y. Labiche. "Is Mutation an Appropriate Tool for Testing Experiments?" In Proceedings of the 27th International Conference on Software Engineering, St. Louis, MO, 2005, pp. 402–411.

[13] Cristian Cadar, Daniel Dunbar, and Dawson Engler. "KLEE: Unassisted and Automatic Generation of High-coverage Tests for Complex Systems Programs." In Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation, Berkeley, CA, 2008, pp. 209–224.

[14] D.J. Agans. "Debugging: The Nine Indispensable Rules for Finding Even the Most Elusive Software and Hardware Problems." AMACOM, 2002.

Author Biographies

Tingting Yu received her MS in Computer Science from University of Nebraska-Lincoln in 2010, and a BE in Software Engineering from Sichuan University in 2008. She is currently pursuing her PhD at the University of

Nebraska-Lincoln. Her research interests include software testing with an emphasis on testing embedded systems and concurrent systems.

Witawas Srisa-an received his MS and PhD in Computer Science from Illinois Institute of Technology. He joined University of Nebraska-Lincoln (UNL) in 2002 and is currently an Associate Professor in the Department of Computer Science and Engineering. Prior to joining UNL, he was a researcher at Iowa State University. His research interests include programming languages, operating systems, virtual machines, and cyber security. His research projects have been supported by NSF, AFOSR, Microsoft, and DARPA.

Gregg Rothermel received his PhD in computer science from Clemson University, an MS in computer science from SUNY Albany, and a BA in philosophy from Reed College.

He is currently Professor and Jensen Chair of Software Engineering in the Department of Computer Science and Engineering at University of Nebraska-Lincoln. His research interests include software engineering and program analysis, with emphases on the application of program analysis techniques to problems in software maintenance and testing, and on empirical studies. Previous positions include Vice President, Quality Assurance and Quality Control, Palette Systems, Incorporated.