Wind River Simics for Multi-core Systems Development

Executive Summary
The hardware shift to multi-core processors and multiprocessor systems calls for new software and systems development tools to help developers transform their code into parallel applications and gain performance increases. Developers now have to know how to create software and architect systems that can use parallel hardware efficiently. Virtualized systems development is a development methodology where the actual hardware of a system is augmented by a Wind River Simics virtual platform, a simulation model of the hardware running on a workstation or a PC. The virtual platform can run the same binary software as the physical hardware, fast enough to be used as an alternative and augment physical hardware for software development.

Virtual Platforms
A virtual platform provides benefits such as freedom from physical constraints, arbitrary configurability, check-pointing and restarting at any point, superior convenience and stability, access to the target long before prototype hardware, and the ability to test faults and boundary cases with complete control and precision.

Virtual platforms can reintroduce control and repeatability in the software debug and systems analysis process by providing a level of indirection between hardware and software. Traditional debugging techniques and debugging tools do not work very well on an inherently nondeterministic system such as a hardware multiprocessor or multi-core processor.

Future systems will look much like the example in Figure 1: software stacks running on small (or large) shared-memory Network (Ethernet, PCIe, RapidIO, etc.) with Local Memory in each Node.

Figure 1: Future systems template
nodes, communicating with other nodes over various types of interconnects, and building a system-level software abstraction on top of the multi-core and distributed hardware.

Note that a software structure such as that in Figure 1 can be used within a single multi-core hardware device. The trends toward virtualization and hypervisors in multi-core hardware make it possible to have several isolated groups of cores run their own shared-memory abstraction, looking at other groups of cores as remote network nodes, even if they physically exist within the same silicon package.

Virtualized Systems Development

Doing multi-core hardware right is not easy, but it is certainly easier than doing multi-core software right. The move to multi-core hardware that began in earnest in 2004 finally pushed parallel software into the mainstream, and the software development world is still catching up. There are three main problems:

- Ensuring existing software keeps working (without taking advantage of multi-core)
- Paralleling existing software to get the performance and power consumption benefits of multi-core parallel execution
- Creating new software that is parallel from the beginning

Virtualized systems development with Wind River Simics offers an approach to creating and porting multi-core applications that has many compelling advantages compared with simply attempting to use multi-core hardware directly for debugging and testing. Simics provides several crucial features for testing and debugging parallel code, and especially multi-core implementations of parallel systems:

- Repeatability: Re-running a software test case in Simics will give precisely the same execution, even when simulating with multiple processor cores and multiple processes on each core. Therefore, the worst problem in multiprocessor debugging is removed: the difficulty of reproducing failing conditions. It makes debugging a multiprocessor as easy as debugging a single program on a single processor. If an error is found in the simulator, it can be reproduced many times over.
- Multi-core debugging: Simics can attach a debugger to each processor core in a multiprocessor machine simultaneously. There is no limitation in accessing the state of the machine due to packaging.
- OS-aware debug: Using the OS awareness offered in Wind River Simics Analyzer, Simics provides the ability to debug individual processes and threads as well as operating-system code. Software processes and threads are tracked as they run on different cores in the system.
- System insight: In Simics Analyzer, there is a single synchronized global view of the system execution, across all cores, processors, and boards in the system.
- Global stop and step: When single-stepping code on a specific processor in a multiprocessor with Simics, everything else in the system also proceeds stepwise. This is not possible on real hardware that provides only limited-skid breakpoints on a single multi-core chip.

- Test scale-out: With Simics, the number of processor cores in a system becomes a simulation parameter, not a fixed value given by the hardware. This makes it possible to test how software reacts to having more (or fewer) cores available and how the software scales.
- Reversibility: The reversible execution and debugging features of Simics work perfectly with multiprocessors and multi-core processors. Simics reverses the entire system, including all the cores, in synchronization. This provides a powerful capability for tracking down hard-to-find bugs such as lock conflicts, deadlock, and priority starvation, which are all more common in a multi-core environment.
- Efficient error provocation: Simics provides several mechanisms to help provoke errors in multiprocessor systems. First, you can vary the speed of processors in the system, which is a good way to stress parallel cores. Second, you can make data writes take very long to propagate between processors. This provokes errors in parallel codes much more efficiently than on real hardware. Third, you can script and inject targeted system behavior at any point in time. Note that real hardware will sometimes provoke errors by virtue of its inherent randomness, but that same randomness prevents replication and thus diagnosis of the problem.

Exposing Bugs in Software

A truly concurrent multiprocessor environment presents a number of new types of potential software errors as well as making classic concurrency problems worse. In particular, existing software that works fine on a single-core multitasking setup often breaks on multi-core devices. Figure 2 shows a simple example of how moving from a single-core to a dual-core processor makes a system much more likely to exhibit a parallel programming problem. The same program is tested across a range of processor core speeds as well as a single-core and a dual-core configuration.
The program is multi-threaded and has a race condition problem. Note that on a single-core processor, the problem only manifests occasionally. And for high clock frequencies, the problem manifests very rarely because the operating system executes more instructions in a program between each process/thread switch, and the chance of being in a critical section exactly when a process switch occurs is lower; the program has longer sequential uninterrupted execution slices for each thread.

When using a dual-core setup, every single execution triggers the bug. If this bug is less aggressive, it would manifest once a year on a single-core device, making it just a rare glitch. Once the system is moved to multi-core, it manifests once a week, making it a real problem. With a virtual platform, such issues can be smoked out early by testing software over a range of configurations, including those not yet available in hardware.

### Debugging Parallel Software

A primary benefit of a virtual platform is that it provides superior debug and analysis features compared to physical hardware. Anyone who has ever developed code for an embedded board will appreciate the convenience of a virtual environment. You get a system that is not randomly flaky, much better control over the target, faster communication, and conveniences such as unlimited numbers of breakpoints. If the target freezes completely, you can stop it and check what happened. You can change system parameters such as core counts, clock speeds, and memory size as well as network setups, with complete freedom and ease.

To give you an idea of how this works in practice, the following is a real-world example of debugging a multi-core bug with Simics.

#### Real-World Example

A Simics virtual platform was used to test an operating system port to a dual-core processor platform. In one test, the clock frequency of the target system was changed from 800MHz to 833MHz, and suddenly the system froze early in the boot process. The system was completely unresponsive, with no input or output.

By varying the clock frequency, it was established that the problem only occurred between 829.9MHz and 833.3MHz. It was not seen earlier because the clock frequency was set to 800MHz. Because of the repeatability of Simics, the bug was trivial to reproduce. Each time the virtual platform was booted with a bad clock frequency, the same crash happened at the same time. Unlike the physical hardware that would have only produced an unresponsive brick, the virtual platform made it possible to examine the state of the processor, memory, and software at the point where it was frozen.

To home in on the problem, reverse execution and interrupt tracing on the serial port, the interrupt controller, and the processor cores were used. This made it possible to pin down the exact cycle and instruction where the problem occurred and the sequence of events leading up to it. The stack back-traces were done at the critical point to determine the locations in the operating system where the freeze occurred.

It was determined that the problem was an interrupt service routine attempting to lock a kernel spinlock before re-enabling interrupts. When it froze, the lock was already taken when the service routine was entered, and with no interrupts enabled there was no way for any other code to run to release it.

The bug was only found because the virtual platform ran the complete real software stack, including interrupt handlers and hardware drivers. It was triggered by changing the system configuration, demonstrating the value of configurability of a virtual platform, and the power of configuration changes to provoke errors in complex code. Because of repeatability, bug reproduction was trivial. The ability to trace and inspect any part of the state was crucial in understanding what happened and in which order. Reverse execution allows for backing across the freeze to inspect the path the system took to get there.

![Figure 3: Repeatability and reverse execution](image)

### Virtual Platform Debugging Benefits

A key benefit of a virtual platform is repeatability and reversibility, as shown in Figure 3. The key problem in finding and fixing software bugs in parallel software is the lack of determinism in the execution of the software system. Every run of a program exhibits a different order of events in the program and even very small changes to the system state or timing result in very different program execution. This complicates debugging greatly because the very act of debugging a parallel program makes timing-sensitive bugs such as race conditions disappear or appear in a different place.

A virtual platform provides determinism and repeatability. The simulator has explicit control over the execution of instructions and propagation of information between processors and can therefore impose a repeatable behavior on the software running on a multi-core processor. Note that this property, usually known as determinism, does not mean...
that the behavior of a software program is always identical. It just means that when running the same software from the same initial state with the same sequence of asynchronous inputs, the same execution sequence is seen.

If anything is changed, a different behavior is seen. Figure 4 shows an example of this, where the same intentionally buggy program is run several times on two different simulated multi-core machines. Each run gets a different result because it is run from different initial states. The simulator can go back and reproduce each run precisely, which is not possible on physical hardware.

Another benefit of a virtual platform for multi-core debugging is that the simulator can stop the execution of the entire system at any point in time. This means that it is possible to single-step code where processors communicate with each other without changing the behavior of the code and that code running on other processors cannot swamp a stopped processor with data to process.

It is also possible to build extra statistics into the virtual platform, to assess fairness and accessibility and catch starvation of different parts of the system.

Testing Scalability

Virtual platforms are good for testing the scalability of systems and software designs. On physical hardware, you are necessarily limited to the core counts and configurations shipping today. On a virtual platform, it is easy to do what-if analysis and test software on arbitrarily wide systems, for
example, to check whether and how a software stack created for a dual-core platform scales up (or even keeps working) as the system moves to triple-core, quad-core, dozen-core, hundred-core, and beyond in future hardware generations. It is also possible to insert delays into the system and model contention in critical places, to see whether the combined hardware-software system is sensitive to certain system configurations or setups.

Figure 5 shows an example of scaling beyond the physical limits of current hardware. There are three machines based on the Freescale MPC8641D SoC. The MPC8641D has two cores in its hardware form; but in Simics, two of the machines have configurations with three and eight cores, respectively. Note that the software stack was not quite designed for this because the /proc/interrupts output for the eight-core machine is not very pretty. This is an admittedly trivial example of how a virtual machine can reveal software issues by scaling systems beyond what is possible with physical hardware.

Figure 6 shows a different use of a virtual platform, this time to assess the scalability of a parallel code. In this test, a series of work packets are sent through a set of parallel worker nodes. Each worker node is a processor core running a local operating system instance and the processing code, using its own local memory. Such a setup is typical for signal processing and streaming data processing. Using Simics, the end-to-end execution time is measured across all the processor cores involved.

The virtual platform is configured with one to 13 processor cores for workers, with an appropriate software setup for each hardware configuration. A contention and delay model is applied to the shared memory used for communication between the cores, with a range of delays. The purpose is not to estimate the precise performance of the software on any particular target platform but to assess how well the software would scale with increased core counts under varying communications latencies. There are also two different communications variants used in the software: single-packet and quad-packet transfers to the worker nodes.

Figure 6 shows that using quad-packet transfers is necessary to obtain scalability because even with perfect memory, the performance of single-packet transfers starts to degrade at seven worker nodes. It also shows that with slow memory and single-packet transfers, adding worker nodes beyond five or six has no real benefit. Therefore, even a program that is computationally “conveniently concurrent” can stop scaling as communications overhead enters the picture. Simics provides the ability to investigate this long before the hardware becomes available so a plan can be made to adjust the systems design accordingly.

**Analyzing the Execution**

A virtual platform can inspect and trace the execution of a software stack without any probe effect. You do not have to instrument the program code or run a background sampling service on the target machine. Instead, Simics can be used to observe events in the target machine software (and hardware) without disturbing the target software execution. If a test case goes wrong, check-pointing and reverse execution make it possible to go back to its start, add analysis tools, and profile or trace the exact execution that caused an issue to appear. Perfect reproducibility is the natural way to use a virtual platform. Another advantage of a virtual platform is
perfect synchronization between the cores, and all traces can be time-stamped without worrying about hardware jitter or clocks being out of sync.

Tracing can be applied at a number of levels, from hardware-level tracing of memory operations, to operating-system events such as thread switches, to profiling the execution of a multiple-board multiprocessor distributed system to determine load balance.

Simics Analyzer provides real-time graphical displays of a target system hardware and software. Using OS awareness, Simics knows which threads and processes are running where in the target system. Simics Analyzer provides both a system overview with a hierarchical view of the target hardware and software setup as well as a timeline view, showing the history of the execution. The timeline view helps you understand when and where software threads, processors, and operating system instances are running.

Figure 7 shows a Linux boot on a dual-core MPC8641 processor. Most of the time is spent in the kernel, but it shows how various processes run and terminate during the boot process. Simics makes it possible to watch what happens as the operating system comes up, before any debug agent or other analysis tools would normally hook into the execution. The data is collected using the normal production kernel rather than having to use a modified debug kernel.

Figure 8 shows the execution of a multi-threaded compute program on an eight-core machine. The operating system kernel keeps most of the threads running on the same core for the duration of the run, but thread 863 changes core twice. The program uses almost all of the available processor time, and after it terminates, the kernel (light blue) is the only thing running on the machine.

The analyzer features extend to multiple machines and hypervisor-based software setups (Figure 8 shows a couple other machines at the bottom of the display). Simics
can tell which guest operating systems are running under a hypervisor, when they are active, and what processes are running under the guests. Breakpoints can be set in guest applications, stopping execution, regardless of which processor the guest application and its operating system are running on.

Simics can also be used to collect data for offline analysis. Figure 9 shows an example of profiling data processed in Microsoft Excel to gain additional insight. Two different ways to parallelize a packet-processing application running on a quad-core platform are compared. The processing of packets has three distinct steps. In the first alternative, each software thread performs all three steps on a packet. In the second alternative, a pipelined setup is used, where each step of the processing has a dedicated thread.

The graphs show that the symmetric setup has superior load balance and can make good use of all four processor cores. The pipelined alternative, in contrast, does not manage to spread the load as well because one stage of the processing pipeline is much heavier than the other two steps. In other testing, this variant of the code suffered packet drops long before the symmetric software setup. The symmetric setup scaled very well, running four times as fast on four cores as on one core, while the pipelined setup did not gain much at all from adding additional processing cores.

**Conclusion**

Building reliable and high-performance systems out of multi-core processors and parallel software is a tough problem for systems developers across the globe. Virtualized systems development offers a key tool to help make this simpler, faster, and less risky. More information and insight into the system workings can be obtained, debugging can be faster, and system and software architectures can be explored by using virtual platforms to augment physical hardware.