Executive Summary

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

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

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

This paper details how virtual platforms built with Wind River Simics can improve software development—and how it is done. An overview describes what makes a virtual platform powerful as a debug and test tool; then the discussion covers how a virtual platform affects overall product life cycle and project planning.

The Power of a Virtual Platform

A virtual platform is best described as a functional model of a physical hardware setup. It is used as the target for software development. A virtual platform can represent a basic board with a processor and memory, or it can be a complete system made up of network-connected boards, chassis, and racks.
The accuracy and fidelity of the model is such that the target software is unable to distinguish the virtual platform from physical hardware; it runs the same binaries and behaves exactly like physical hardware. When these high-fidelity virtual platforms are combined with a feature-rich simulation environment, developers can define, develop, deploy, and integrate target-specific firmware, operating system kernel, device driver code, application, and communication stacks, even while the hardware design and production progresses in parallel.

The use of virtual platforms provides many benefits to the product life cycle, resulting from traits of the virtual platform, or from the development techniques that can be employed on those virtual platforms.

Fundamentally, a virtual platform turns inflexible hardware into flexible and accessible hardware. The virtual platform is also “just software” and can be managed like software: checked in to revision control, moved over the Internet, and installed on every developer’s machine.

In addition, the virtual platform makes it possible to access and control all parts of the target system, making it an open box that does not hide things like hardware tends to do. The virtual platform is isolated from the outside world and can do tricks with time that is not possible in the real world. It can observe all inputs and outputs to the target system, providing perfect repeatability. This provides the fundamental support for many unique development and debug features.

Simics Features for Software Developers

In the 1999 movie The Matrix the main character, Neo, found that in a virtual world he could do things that would otherwise be impossible. One memorable scene shows him dodging bullets by bending back and forth at hyper-speed. Similarly, virtual platforms enable the impossible. In a virtual platform, developers can monitor every bit and every register within the hardware, freeze execution of the whole system, inject any hardware error, take system checkpoints, precisely repeat every operation from run to run, automate every step of the system’s operation, and even run the system in reverse.

Full System Stop

Unlike physical hardware, virtual platforms and virtual systems can be completely and synchronously stopped. With virtual platforms, there is no “skid” as the hardware tries to stop all processors in the system. All processor cores stop with a single command, not from separately issued debugger commands as used in physical systems. With virtual platforms, the stop includes not just processor cores but also peripheral devices and even data in flight on networks and buses. Even a system consisting of many different boards and processors can be synchronously stopped.

An important result of full-system stop is that the target system perceives no disturbance or timing variations from the debugging; it runs exactly like it would when not being debugged. Code that is normally very timing sensitive such as interrupt handlers, hardware drivers, and multi-threaded programs can be single-stepped without any effect on the target state. Since all processors are stopped, you will not get time-outs as one thread is debugged while other threads wait for its result.

Figure 1 shows a screenshot of Wind River Simics single-stepping inside an interrupt handler on a Power Architecture board. Note that it is stepping on one of the four processors in the target systems and that everything else is standing still while it steps on CPU 0 on the “tuna” machine.

In the frozen state, developers have full access and visibility into all hardware and software variables and state. They can investigate the state at their leisure, set breakpoints, and then resume normal execution, all as if the platform never stopped.

Repeatability

How many times have engineers or users of a software product hit a bug only to be unable to repeat (and thus correct) it? The primary reason for this lies in the inherently chaotic behavior and variable timing of physical hardware, especially when concurrent processes such as interrupts, multiple cores, or multiple software threads are involved. On physical hardware, repeating a bug may take weeks or months as developers work to isolate the precise set of parameters that will force the system into a faulting state and look for that lucky execution when things go just right to make things go wrong in the right way.

Virtual platforms, on the other hand, are fully repeatable. If a bug or strange system behavior is observed once, it can be repeated any number of times with ease. How? Simply load a...
pre-bug system checkpoint and then run the system forward. The repeatability guaranteed by Simics ensures that the system will execute the same way (including concurrent behaviors) and that the same issue will be hit at exactly the same time as the first time.

Simics is designed to provide repeatable execution behavior across host machines, locations, and time. Simics target system semantics are defined in a host-independent way, which is key to enabling repeatability. Simics will repeat the same execution when starting from the same target state and providing the same asynchronous inputs at the same times as the original execution.

Note that repeatability does not exclude variation in the execution on the target. When you run a fresh compile of a changed program, the behavior might be different from the previous compile. Using a virtual platform does not change this. Since the initial state (which includes the program code) is different, the execution will be different. The program will not repeat the same execution it had when a previous build was run on the virtual platform. Each time is different.

Another example is running a multi-threaded program many times in succession within the same Simics simulation run; each time it runs, the precise pattern of thread starts, thread switches, lock and mutex operations, thread communications, and other concurrent properties of the program will be different, which might give different results in each run. The key with Simics is that you can repeat each run after the fact with perfect precision—not that you can predict what will happen.

Reverse Execution

“If I only knew then what I know today.” Hindsight is 20/20; a problem is far easier to identify and correct after it has happened.

Reverse execution complements system repeatability by allowing developers to run their systems backward beginning at a point after the problem occurs. Debugging in reverse is straightforward because even while the system proceeds backward to a point before the problem occurred, it continues to stop at every software and hardware breakpoint along the way. This capability alone has saved developers months of effort on difficult-to-find system bugs. Reverse execution makes it possible to reverse out of system crashes to debug them immediately rather than trying to stop just before the system goes bad and corrupts all state.

Figure 2 shows the effect of reverse execution and repeatability on debugging. On physical hardware, each run is different. In Simics, the second run reliably repeats the same bug, and with reverse execution you can diagnose and debug it.

Advanced and Unlimited Breakpoints

A virtual platform is virtually unlimited in terms of the breakpoints that can be supported. Unlike hardware, the virtual platform can plant any number of breakpoints in the code, without modifying the code in any way. Memory-access breakpoints can cover megabytes or gigabytes of memory without problem and be controlled down to individual locations.

In addition, virtual platforms allow breaking on normally inaccessible events such as the following:

- Interrupts from hardware devices
- Exceptions resulting from software behavior
- Control-register accesses in a processor
- Device accesses from software
- Network packets being sent
- Output on serial consoles and graphics devices
- A certain number of instructions or a certain amount of time passing in the virtual target
- Log messages from models in the simulation (see the section “Cross-Layer Debug” for more details)
- Task switches and system calls (with OS awareness)

Source Code Debug

Source code debug might seem like a mundane feature, but a virtual platform lets you apply it to any code, not just user-level applications. You can debug a boot ROM, low-level firmware, device drivers, operating system kernels, and other code that is usually hard to get a good grip on in a debugger. Using OS awareness, it is also easy to debug individual user-level tasks. Figure 5 shows an example of Simics source-code debugging in action.

A virtual platform lets you run a stripped minimalized binary on the target while loading full debug information into the debugger (the difference in size can be a factor of 10 or more). There is no need to reduce the amount of debug information generated during the compilation. For really tricky setups, the virtual platform provides facilities to manually specify memory remappings and to tell the debugger where code and data is located in memory. Source-code information and symbols can also be used to write automated debug scripts that know about the target software and not just about the lowest-level hardware.
Cross-Layer Debug
Virtual platforms see everything that is going on in the target system and can trace interactions at any level in the hardware and software stack, making certain types of bugs much easier to resolve.

For example, it is easy to trace the interaction between software and hardware in a system. This makes it easy to resolve a conflict between software and hardware teams over where the root cause of a nonfunctioning system is to be found, reducing the typical rounds of discussion over where a problem is located. With a trace, just compare the operations performed to the specification of the hardware and see if the hardware is misbehaving or if the software is programming it in the wrong way. The virtual platform provides a common point of reference for the hardware and software designers to discuss the issues.

Virtual platform models can also issue warnings about suspicious activity that is not bad enough to force an exception or error in the target system. Quite often, errors in code are masked by the behavior of the hardware. For example, accessing nonexistent programming registers in a device is typically harmless because the hardware just ignores accesses it does not understand. A virtual platform, on the other hand, can and will warn about this, making the code cleaner and avoiding crashes when the next-generation hardware starts to use the previously reserved registers or bit fields for something unexpected.

Single Build
Because the virtual platform runs the same software binaries that will be run by the physical system, there is no need for a special “host-compiled,” “simulation,” or “development” build. With a virtual platform, developers can always use the production build for their software. This reduces the number of build setups for a project, lowering both the maintenance costs for the build environment and the risk of errors from inconsistent builds. The virtual platform also makes it possible to include binary-only third-party software in the build and testing from the start. In a host-compiled simulation, such software cannot be integrated and has to be replaced with stubs or simulations.

Using a target build from the start also ensures that issues related to the target environment are found as early as possible. Instead of being discovered in the integration stage, errors are found when a developer compiles and runs code the first time.

System Insight and Tracing
With physical hardware, the developer’s access to specific registers is limited to those that the system-on-chip (SoC) designer chose to make visible. This often includes most registers on the CPU and a select set of registers on the MMU and supporting I/O devices. Unfortunately, some OS kernel and driver software must access devices or registers that are simply not visible (or debuggable) on physical hardware. This issue gets compounded by the use of multiple layers of software such as hypervisors as well as in secure systems where hardware access for debug is naturally restricted.

On a virtual platform, every bit, byte, and register on the system can be both inspected and modified to provide faster development and higher code quality. Figure 3 shows how you can inspect the registers of a hardware device and the memory of a target from any perspective.

In addition, Simics allows you to trace anything you can observe. Tracing can be as simple as asking Simics to trace all memory accesses and instructions executed, or as complex as building your own customized trace modules to perform tracing very specific to your system. Simics can generate network traces in standard pcap format for online and offline analysis of network traffic.

Figure 3: Device registers and target memory
**Backdoor Access**

In addition to allowing inspection of all state in a system, a virtual platform allows you to change it. Using backdoors in the simulation, any state in the target system can be manipulated from the outside, without the need to run any code on the target or the target software noticing that changes are being made.

One example of how backdoor access can speed a common operation is loading an OS kernel directly to simulated memory. Compared to having a boot ROM download the kernel over a network from the development host, this is much faster. Another example is to replace flash programming of the target system with direct updates to the flash content, turning an operation that takes minutes or even hours in the real world into an instant process.

Backdoor access can be used to jump-start software development by putting the target in a useful state even without a boot loader or other setup code being run. For example a virtual platform script can set up translation lookaside buffers (TLBs) and break address registers (BARs) in the MMU in such a way that code can just run. No need for the target software to start with the tricky operation of configuring the MMU—that can be added later, once the basic features of the code have been proven.

Backdoor access can be used during debug sessions to patch target code and change the contents or memory, variables, or registers to see if a simple fix can solve a problem. It can be used to replace loops, clearing memory with a simple simulation operation that sets the contents of all uninitialized memory.

Backdoor access is also the key to fault injection, which is discussed in more detail later.

**Check-Pointing**

Simics checkpoints store the complete state of the virtual platform to the host computer disk. When the checkpoint is loaded into Simics, the result is the same target system state as when it was saved. The checkpoint includes the hardware setup (boards, networks, plug-in cards, and other configuration aspects), hardware state, and software state. It contains the contents of memories and disks, the state of processor registers, MMUs, peripheral devices, and network connections. It also stores some core Simics state, such as the current time and events queued for later execution, allowing the virtual platform to continue its execution seamlessly from a checkpoint.

With check-pointing, a repetitive target setup procedure can be done once and for all and saved. Subsequently, the fully configured system can be immediately resumed from that state, without any need to repeat the setup procedure. This gets more and more important as simulated systems increase in complexity and workload size. The simplest case is booting an operating system on a board; but in general, the system setup will also include loading target applications, starting servers, configuring networking software, and setting hardware configuration parameters. Large system boots can easily take hours of real-world time, as it involves the simulation of many billions of instructions across hundreds of processors.

Using check-pointing reduces the amount of time spent on routine work and increases the time spent doing real value-added work. Another benefit is that the state is consistent across multiple developers. Since they all use the same checkpoint, there is less risk of a manual mistake in the setup process that would affect test results and cause spurious bug reports and wasted work. One team member (or a platform team) can set up a standard platform with certain hardware and software and distribute it to all other developers very easily.

As shown in Figure 4, the target system often loads different software each time, starting from the same checkpoint. Using scripting, this process can be made automatic; and by using fast backdoor accesses to load software, a retest of a newly compiled piece of software on a target can be done in a matter of seconds.

Check-pointing allows developers to store interesting system setups that can be quickly activated for particular tests. Rather than configuring the hardware and software setup manually to get to a certain system configuration, a developer can simply browse a set of saved checkpoints and open the one that applies to the next test run. This saves time and broadens test coverage.

Check-pointing allows developers to save their work and pick it up again hours, days, weeks, or even years later. With

![Figure 4: Check-pointing saves the boot](image-url)
physical hardware, the system must normally be rebooted and manually coaxed close enough to what it was before. In many cases, the first step is just finding the hardware needed to reproduce the setup. With a virtual platform, this process becomes immediate—just like saving and opening documents in regular office applications.

OS Awareness
With OS awareness, the virtual platform supports working with OS-level abstractions such as processes, tasks, and threads. This allows debugging individual threads in isolation and seeing what is running where in the system. Breakpoints and stepping inside a thread will step through the execution of that thread, skipping times when the thread is calling into the operating system or is inactive due to OS scheduler decisions.

Figure 5 shows how OS awareness further enables Simics to plot the execution behavior of software tasks in a timeline. The target is heterogeneous, containing two boards, the upper one running Linux on an x86 target and the lower one running VxWorks on a Power Architecture target. The x86 system is running a program called “banner” that does not use much CPU power, while the Power target is running a compute-intensive program called “rule 30,” which is using most of the CPU.

With OS awareness, Simics users can stop the simulation on events such as tasks being swapped in or out by the OS scheduler or doing system calls. Instrumentation and tracing can be restricted to a single execution context, allowing developers to focus on a small part of the system. Scripts can use OS awareness to automate debug tasks and analyze the execution of a system with full knowledge of which tasks are running and the debug information for these tasks.

Code Coverage and Profiling
As discussed previously, a virtual platform can inspect and trace the behavior of a target system without any probe effect or intrusive behavior. This makes it possible to perform profiling and code coverage on actual shipping binaries. There is no need to include special directives in the compilation or patch the target binaries to provide hooks for coverage and profiling. A developer would typically load a stripped binary onto the target system and use the original binary with full debug information within the virtual platform to associate memory locations with code and variables.

Code coverage and profiling can be applied to any software, including boot firmware, hypervisors, and operating system kernels that would normally be inaccessible to analysis tools. With OS awareness, code coverage and profiling can be performed on individual user level tasks, excluding noise from the operating system.

Fault Injection
Virtual platforms can inject faults at any place in a system, at any point during a run. Thanks to scripting and the inherent control over time in the simulator, faults can be programmed to be completely repeatable, removing any random factors from fault testing. Virtual platform fault injection is completely nondestructive because the only thing affected is a simulation of the target. Breaking things has never been so easy or cheap before.

Faults can be applied in many ways and at many levels of a system. The following are some example faults that Simics has been used to inject:

- At the lowest level, memory contents and processor register values can be corrupted to simulate both transient and permanent faults and how the system software deals with such issues.
- At the system level, network links can be cut or entire boards disabled to test system-level redundancy and fault tolerance.
- If the target system contains error detection hardware, Simics can be used to force it to flag errors and test that software handles the errors accordingly.
- Overheating errors can be triggered by forcing temperature sensors to return very high values.
- Interrupts can be triggered at very high rates.

Figure 5: OS awareness and system debug
Scripting and Automation

Virtual platforms are excellent automation tools. Scripting can perform interaction with the system such as reading the serial console output and entering appropriate commands and text according to programmed instructions because virtual platforms control all inputs and outputs of the target system. Scripts can be used to load software, change the target state, inject faults, set up debug contexts and debug information, and set breakpoints, and anything else that can be done interactively.

This ability to automate the system to such a fine level of detail enables the detection and duplication of bugs, system corner case testing, integration and test, creation of advanced training scenarios, and sales demonstrations.

Simics scripting can be used to automate the execution of software test cases so that they can be run in pure batch mode with no user interaction. This makes it possible to run tests remotely and in parallel on servers and clusters. If errors are detected, checkpoints and traces can be passed back to developers for reproduction and analysis. Simics runs can be controlled from other software packages, such as grid software packages, allowing large-scale automation of target testing.

Virtual platforms make it easy to automate regression testing because there is complete control over the target system. Reboots, resets, and result checks are very reliable. System checkpoints can be used to avoid repeating lengthy processes such as system boots and software loads and to ensure that multiple tests are run from the same initial state.

Figure 6 shows an example script that runs a program on a target system and times the run in host and target time. The Simics command-line interpreter supports features such as variables, loops, conditional execution, and even parallel script branches. It can also call out to any Python library, and advanced scripts are often written directly in Python, offering the full power of an object-oriented programming language for scripting Simics.

Parallel Testing

A virtual platform makes it very simple to parallelize the execution of test cases. A large server, cluster of PCs, or company grid can be used to run many instances of a virtual platform. Each instance would typically start from a checkpoint and run through some script to run a certain piece of software in a certain way.

Parallel testing can be used to increase test coverage and decrease test run-times compared to physical hardware. For example, test suites that take days to run using a few hardware boards can be completed overnight using massively parallel runs on virtual platforms. This brings regression testing into the domain of a nightly build rather than a weekly task, which makes it much harder for bugs to sneak back in. The net result is higher-quality software developed in less time.

Flexibility and Scalability

With a virtual platform, it is simple to change the latency, speed, or size of memory. It’s easy to add another processor to the platform, add a new platform in to the system, or add new network interfaces. Developers can experiment with more configurations than with physical hardware, both to find configuration-related bugs and to explore performance scalability as hardware changes, long before physical hardware is changed.

For example, Simics has been reconfigured to test software and find bugs:

- Changing the clock speed of a processor from “quite low” to “very high” to reveal race conditions and timing assumption in parallel code
- Changing the number of processor cores in an SoC to test software scalability and find bugs related to the processor count
- Creating a target memory containing 264 bytes, to test that an operating system does not misbehave in a configuration where the most significant bit of an address is set
- Creating maxed-out network configurations to test self-coordinating networked system, master selection algorithms, and system boot behavior
- Making traffic generating nodes in a network much faster than normal, to provide data streams that easily overload even very powerful network equipment

Figure 6: Example Wind River Simics script
Access to Inaccessible Connections

On physical hardware, some of the connections from an SoC such as serial ports and Ethernet ports are often inaccessible due to board design issues (e.g., only a single serial port is exposed when the SoC has several). With Simics, such ports can be made accessible, providing superior target access. It is also possible to add extra hardware devices to the Simics model to enrich the set of available connections.

Being able to access more connections than physical hardware is very useful, in particular when setting up, debugging, and experimenting with hypervisor and asymmetric multiprocessing (AMP) targets. With a virtual platform, you can get direct connections to the secondary operating systems, which reduces the amount of code that has to work right to interact with them. For systems that are built on racks or are deeply embedded into equipment, Simics allows easy access to serial ports that are very difficult to get to on actual hardware, simplifying target access during software development.

The virtual platform makes it possible to communicate with a target earlier in the development cycle because there is no need to complete a board with serial ports and network ports before connecting to the target. Simics also provides a very simple serial device that lets software send out characters to consoles with only a few lines of driver code.

System Simulation Integration

Full system simulation does not have to end where the SoC, board, rack, or network ends. It can include simulations of the outside world (or even the physical world). A virtual platform that runs a control system can be connected to a model of the controlled hardware system. Such models are often expressed in tools such as MatLab or LabView but can also be written in C or Fortran. Other examples are models of Internet traffic or telecommunications networks used to test telecom or datacom systems.

With an integrated simulation, it is possible to test the actual code in the context of its electronic system (hardware, OS, and software) as well as the environment the electronic system is operating in. If software issues are found, the full power of debugging can be applied. A developer can thus find software issues that depend on the external environment as well as how the code works with itself and with other code.

With checkpoints and recorded interactions with external simulators, the software and its virtual platform can be taken back to the lab and debugged without having to set up the complete simulation environment.

Virtual Platforms Across the Product Life Cycle

Simics is a true life-cycle tool for electronic systems. Simics can be used from the very first preliminary design of a system on up to the end as a 30-year-old avionics system is being maintained and getting software upgrades. We have found that after an initial project, users of virtual platforms realize that many software development tasks can be reduced in time or be moved up in the project schedule. By taking these improvements into consideration when planning a new project, a very different project plan can emerge, one that truly compresses and improves the life cycle of an embedded project.

Hardware for Everyone

Since virtual platforms can be made accessible in any number without any hardware costs or delays, they remove the gating factor of hardware availability from software development. This lets software development ramp quickly, even for hardware that is not yet available, or hardware that is expensive, complex, or hard to set up.

Additionally, the ease of access and scalability of virtual hardware makes it possible for an individual developer to test new code and changes in a complete, integrated system environment. This tends to catch obvious errors immediately, saving the latency and effort of sending the software to a specialized system integration and verification group. This also leaves the integration and verification teams free to concentrate on more difficult errors and integration work.

Bug Transportation

One of the most challenging problems in software debugging is to correctly and reliably reproduce a problem found by someone other than the software developer. Typically, test departments and software users have to type long and brittle “instructions to reproduce the error” in bug-tracking systems, along with a list of the version of the software that contains the bug and anything else in the software environment that seems relevant.
By using a virtual platform with check-pointing, any bug can be captured, communicated, and reproduced any number of times, at any location, on any machine.

As illustrated in Figure 7, when a bug occurs, the person hitting the bug saves a checkpoint (R) of the combined hardware and software state. This checkpoint is sent to the software developer. Opening the checkpoint using the same virtual platform, the developer is able to reproduce the bug as well as investigate the target system state for clues as to what went wrong. There is no need for the developer to get back to the reporter to gather more facts because everything is encapsulated within the checkpoint. Using the virtual platform ensures that the developer and the bug reporter are using the same hardware and software environment, regardless of how hard it is to get hold of the physical hardware needed and how difficult it would be to configure the software.

When Simics is integrated into test environments, it is possible to automate the creation of checkpoints for bug reports. When a test framework finds a failed test, it can order Simics to create a checkpoint of the target state and to save a recording of the input and have it automatically uploaded into a bug reporting system.

Bugs can also be distributed to many developers at once, allowing a team to work on the same bug simultaneously, speeding the time to resolution.

### Executable Specification

With virtual platforms, hardware and software teams are able to experiment with new system architectures by creating and collaborating around virtual platforms. A virtual platform can be used to run real software loads for the purpose of prototyping and design concept testing during the architecture or systems definition phase of a project. It can also be used to communicate the design between teams, forcing a more precise definition of behavior than written documentation.

When used in this manner, the virtual platform becomes a living executable specification that can be used by everyone: systems engineers, hardware and software teams, marketing teams, and salespeople. Like traditional requirements documents and specifications, the virtual platforms developed during the course of a project can be saved and archived for future reference.

The use of an executable specification encourages hardware and software teams to work together from the very start of a project. Now fundamental design problems can be detected and resolved, long before they migrate into physical designs. Traditional development approaches might only discover these issues during systems integration where they become expensive and problematic to correct.

With a virtual platform, the hardware team can get feedback on their hardware design decisions from the firmware and software teams at an early stage. Without this early feedback, software developers are often frustrated by difficulties in actually making use of hardware features due to poor programming interfaces, let alone able to use hardware to its full performance potential.

### Iterative Modeling

A virtual platform will often evolve incrementally, aiding software developers all the way from the model’s origins as a basic platform to the final full system model. When using virtual platforms, it is important to remember that a complete model of the system is no more important to specific software development tasks than a CD player is required to check the brakes on your car.
By focusing on what is known and what is needed first, a virtual platform can be iteratively developed in parallel to the hardware design. As shown in Figure 8, the virtual platform will become more and more complete as the hardware design becomes clearer. In this way, software porting can start long before a final design exists, let alone physical hardware prototypes.

The iterated hardware models should be delivered to the software team, allowing them to use the hardware very early and facilitating the use of the virtual platform as an executable specification.

Often the functionality or even existence of a particular hardware device is not required for a specific software task. In these cases, that device can be simply omitted, stubbed out, or replaced by off-the-shelf models that provide similar functionality. For example, boot code that initializes some device by writing register values can be developed, debugged, and tested on a model consisting of only the CPU, memory, and device registers directly used by the code. All functionality, beyond providing the read/write register for initialization, can be deferred until needed for OS porting and device driver development. An operating system kernel can be brought up using only a CPU core, memory, timers, interrupts, and a serial port.

**Agile Integration**

Once the basic operating system and software layers are up and running on a model, system integration testing can begin. This means that integration testing can start very early in a project, long before either hardware or a complete software stack exist. User applications can be developed for the first iterations of an operating system and integrated with other applications, legacy code, and third-party binaries on a virtual platform early in its development cycle.

By starting integration testing as early as possible, the integration and system testing teams can provide feedback and test results to the software developers early on, just like the software developers are able to provide feedback to the hardware designers. In this way, virtual platforms enable an agile iterative work process that encompasses hardware design, platform software development, application development, and system-level integration and test.

**Avoiding Show-Stopper Bugs**

The debugging facilities of virtual platforms reduce the time required to find and correct software bugs. The precise effect of this on a particular organization will have to be measured over time, but it will translate to a reduction in typical software development time. There are cases where average bug resolution times go down by a factor of four or more.

The debugging facilities have a huge impact on the really hard bugs—the ones that keep development stalled for weeks and have software engineers tearing their hair out. Those bugs tend to be intermittent and hard to reproduce on physical hardware systems. Since reproduction is trivial in a virtual platform, debugging can be much more productive and focus on the analysis of a problem, not on reproducing it.

We have seen bugs that take weeks or months to correctly diagnose on physical hardware found in a matter of hours or days using virtual platforms. This reduces the risk of major stalls in a schedule due to a really nasty bug.

**Long-Term Support**

Many products have requirements for very long maintenance times for software. In the aerospace and defense industry, having to support, maintain, and even extend the functionality of software in 20–30-year-old systems is not uncommon. This creates a need to access stable and available instances of older hardware setup.

Virtual platform models can be created for any hardware, even hardware designed decades ago. The virtual platform of the old hardware can run on any current machine, providing the old hardware in stable form without worrying about physical wear or the magic smoke escaping. Virtual platforms do not wear out over time but provide reliable access to old hardware for decades to come.

An interesting use for virtual platforms for old hardware is to simulate faults that have appeared in deployed systems. For example, if the target system is flying through space on a long mission, a virtual platform on earth can be used to model the faults that appear in the physical system during the mission. Software workarounds and patches for hardware issues can then be tested on the ground, in the simulated environment of the virtual platform, before being uploaded to the live system.
Demonstration Platform

Virtual platforms can be used to demonstrate new systems to potential customers. Rather than developers, marketing, or sales teams carrying heavy and brittle hardware around, they can use a laptop to show a system to customers. The system aspects demonstrated will vary with the product:

- A virtual platform can be used to seed potential customers with a model of a new SoC, allowing an ecosystem of tools and operating systems to be developed prior to the formal product launch.
- For a new processor architecture, virtual platforms can be used to test benchmarks and customer code on a cycle-accurate model of the processor, demonstrating the benefits of the new processor long before hardware becomes available.
- Operating systems and their features for various architectures can be demonstrated using a virtual board instead of a physical board. Each demonstrator can easily carry many different architectures, which is not really practical with physical boards.
- Software applications can be demonstrated without relying on physical boards. In projects where software deliveries have hard milestones, this can be a real lifesaver because hardware availability does not stop the demonstration of software.
- By using a virtual system containing many machines, complex network software such as security systems and hardware-accelerated packet processing can be demonstrated using a single laptop rather than a network of physical boards.

Scheduling Projects Involving New Hardware

For projects involving developing software for new hardware, the early and wide availability of virtual hardware compared to physical hardware enables a number of schedule improvements:

- It is possible to decouple the hardware and software development and delivery schedules. The scheduling of the software is only dependent on the finalization of a hardware design, not on its actual physical availability.
- The start of software development can be scheduled with greater certainty because the limiting factor of hardware availability is removed. This greatly reduces the ripple effect of hardware delays on the overall project.
- Software development can ramp up faster, again because it is not tied to hardware availability.
- Overall project risk is reduced because unexpected delays in hardware availability (e.g., a key chip or processor not becoming available from the vendor according to initial plans) do not impact software development.
- Once the hardware appears, bringing up the software on the physical hardware is a much shorter and smoother process. Most bugs will already have been found and fixed and what remains are often bugs related to detailed hardware timing and analog effects of printed circuit board (PCB) design. The difference in development time can be striking, reducing the time to a working system by several months.
- Hardware and software developers can cooperate around the virtual model and provide mutual feedback on the design earlier than otherwise possible. This tends to avoid hardware design mistakes that make life unnecessarily hard for the software developers.

Conclusion

Wind River Simics is a system and software development tool that improves engineering efficiency and reduces costs across the life cycle of a product. In this paper, we discussed how Simics features support the development, integration, test, and debug of software at all levels of the software stack, from the first lines of code written for a brand new project to the maintenance of decades-old software. In addition to improving the working environment of individual developers, Simics makes the overall product development process run smoother and faster.