Testing Embedded Software using Simulated Hardware

Jakob Engblom[1], Guillaume Girard[1], Bengt Werner[1]

1: Virtutech AB, Norrtullsgatan 15, 11327 Stockholm, Sweden

Abstract: This paper presents an approach to testing software-intense embedded systems using simulations of the target hardware instead of actual target hardware. Simulation can be used as an alternative to the actual target hardware for a significant portion of the testing effort, saving developers time and money, as well as increasing test coverage and providing better debugging facilities. We cover the technical issues involved in creating simulated test systems, as well as the business aspects and benefits.

Keywords: Verification, Validation, Simulation, Business models

1. Introduction

Simulation as a tool for testing and debugging software has a long history going back to the very first electronic computers [1]. Simulating a system has always carried the advantage of increased insight and flexibility, at a cost in execution speed and timing fidelity visavi the real machine. However, until recently, use of simulation technology for large-scale embedded systems software development and testing has been fairly limited.

Hardware designers for processors, supporting chip sets, systems-on-chip, and servers have always made use of simulation in order to model hardware early. Simulation is used for performance evaluation, to test various ideas for implementation, and to validate that a system works as intended [2][3].

Providers of software development systems (especially for 8-bit and 16-bit processors) have always provided instruction-set simulators (ISS) for the target systems. However, such solutions have been limited to only simulating the processor and not the surrounding hardware, making them suitable for simple initial software test but not for running operating systems or software that interacts with hardware.

Initial firmware bring-up and ports of operating system codes to new embedded and other computers is quite often performed using simulation tools, as the real hardware is typically not available early enough [4][5].

Some software work can be performed on the developer workstation using API-level simulations of the embedded operating system [6]. For final verification of functionality, it is necessary to use the actual binary that would be used in the real system, which is not possible in an API-level simulation.

Overall, however, for the volume work of embedded software development, embedded developers have relied on development boards and instances of the real target hardware boards[1].

We introduce a simulation tool that can replace the use of hardware to a large extent, by providing a simulation model that is faithful enough that all software for the target can run on it, and fast enough that it can be used in daily work. This paper describes this tool, Virtutech Simics, and how it is applied to a number of tasks in embedded software testing and development, with a focus on testing.

2. Simics

Our simulation tool, Virtutech Simics (see www.virtutech.com), is capable of simulating large computer-based systems and complete networks. The simulation approach used is *full-system simulation*, where the processor, memories, peripheral devices, and environment of a target computer system are all simulated in such detail that the target software cannot tell the difference from a real target system. The full-system model runs the same binary software as would run on the real target, including device drivers and firmware, as illustrated in Figure 1.

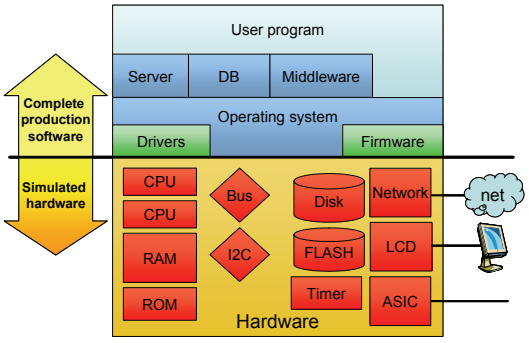


Figure 1: The concept of full-system simulation

At its core, Simics is an event-driven simulator where processors are treated specially for performance reasons. All I/O and other peripheral devices on a

[1] In this paper, we use the term “board” to mean any complete computer unit used in an embedded system. This can be a rack-mounted card, a stand-alone box, or some other package.

board (and thus all interconnects between boards in a system) are simulated in a *transaction-based* style. This means that accesses to devices are handled as a synchronous unit, rather than simulating the actual bus traffic required to perform the request in the real hardware. Networks are simulated on a packet level, where entire packets are sent and received as units. This simplification is key to gaining simulation speed.

Simics is based on very fast instruction-set simulators (ISS), which are bit-accurate in the results of all instructions, including supervisor-level operations, floating-point operations, and model- and hardware-specific registers. The use of an ISS provides the user with total virtualization of the target system; For example, Simics can execute operating systems and applications for PowerPC or MIPS targets on an x86 PC or a SPARC workstation, as illustrated in Figure 2. Simics handles endianness and word length differences, and can simulate a 64-bit little-endian system on a 32-bit big-endian host.

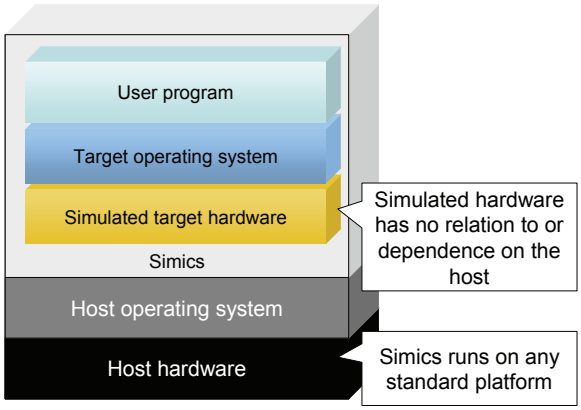


Figure 2: Simics provides full virtualization

Simics does not require any special hardware or particular operating system; it is a pure software application that can run on regular Solaris, Linux, and Windows workstations and servers. This also makes the virtual (simulated) hardware future-proof. As illustrated in Figure 3, the same simulated hardware will continue to be available as host machines change and evolve, as long as Simics is available for the new host.

The behaviour of the simulated hardware remains the same over time, allowing for maintenance and development of code for a platform for extended time frames. Unlike real hardware, the simulated hardware will not break or become unavailable.

Simics is designed to be a fast simulator, and can currently achieve speeds exceeding 2000 MIPS when running single-processor workloads on top of full simulated systems with a real operating system (measured when simulating a single PowerPC 750 processor running on a 2GHz Opteron PC). Such high execution speed allows for real workloads to be run in simulation, including operating systems, network stacks, and complex applications. This is a crucial enabler for using simulation in testing large real-world codes and real-world test suites, in a completely virtual environment [7].

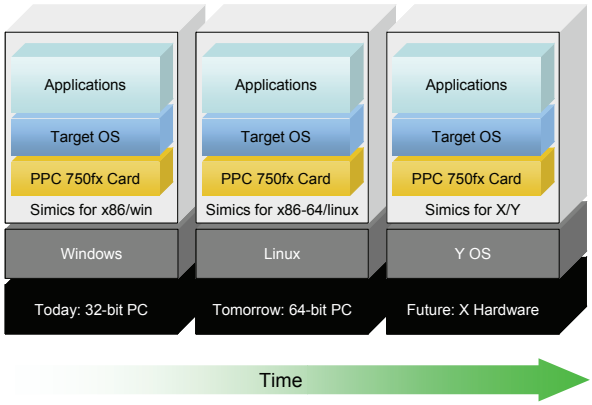


Figure 3: Simics provides insulation to host change

2.1. Using Simics in Software Development

The Simics framework has been specifically designed to facilitate software development and test on simulated machines, and has a number of handy features that aid in performing, automating, and diagnosing tests.

Simulation in Simics is guaranteed to be *deterministic* – from the same initial state, the simulation will simulate the exact same execution path every time the simulation is run. If variation is desired (in order to test multiple code paths that depend on target timing, for example), it can be provided by adjusting simulation parameters for a particular run.

Simics supports *checkpointing*, where the complete state of a system is written to disk. This checkpoint can later be restarted at the precise instant where it was saved. Together with determinism, this allows for executions to be repeated any number of times, using any workstation running Simics.

When simulating multiple processors and/or machines (for example, in a distributed system), Simics provides *global synchronization and stop*. If one part of one machine is stopped, the entire simulation is stopped. This makes it feasible to single-step interrupt handlers and perform deterministic debugging and analysis of multi-processor and distributed systems [8].

Scripting in Simics is very powerful, with a full Python language interpreter as well as an extensible command-line interface. Scripts can react to output from the simulated machine and to events inside the simulated machine (breakpoints, exceptions, device accesses, etc) and provide scripted intelligent input to the simulation.

All target state is accessible without probe effects, including information which is hidden and hard to get to on real hardware, such as TLB tags and the

contents of supervisor-level registers. This ability to observe also provides the ability to trace and log everything that happens in the system.

Tracing can be used to *profile* and perform *code coverage* analysis without having to instrument the target code. Even very detailed code coverage analyses like decision and condition coverage can be implemented transparently to the code being executed; it is all handled by looking at the execution trace and noticing which instructions are executed (and which are not).

All target state can be manipulated. If the state can be observed it can also be changed. For example, for fault injection, transient and permanent faults can be easily simulated [9][10].

Simics supports *source-level debugging* of software running on the simulated machine, including firmware and operating systems. Any code can be debugged, as the simulator has complete control over the state and execution.

2.2. Network Simulation

As noted above, Simics simulates not only individual machines but also networks of machines. Each machine in this instance runs a complete software stack, including operating-system drivers for the network devices. In network simulation, network traffic is sent as entire packets. It is possible to inspect (and modify or destroy) network packets as they travel across the network, as illustrated in Figure 4.

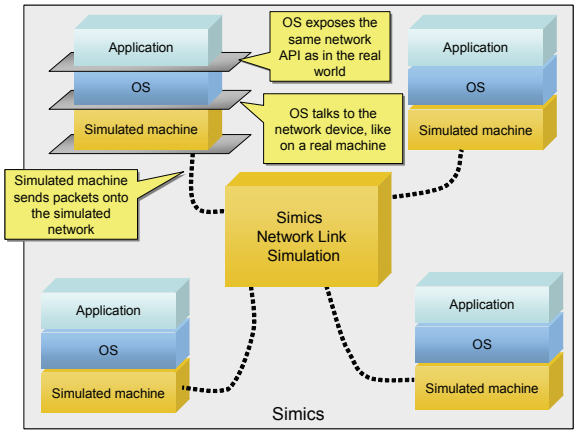


Figure 4: Network simulation with Simics

There are no limitations as to how machines can be combined in a simulated network: it is possible to combine many different machines of different types and speeds, and Simics simulates the relative execution speed of the different machines. For more information on Simics network simulation, we refer to [8].

2.3. Simulation Timing

In order to gain simulation speed, Simics simplifies the timing of the target system. In the basic model, processor instructions are assumed to have a fixed execution time, and device accesses provide simple time models for when transactions complete. The function is identical to a real machine (which is necessary in order to run real binaries), but the timing might be different.

This makes Simics suitable for testing the functional correctness of code, and coarse-grained timing. Simics is not intended to analyze or predict the precise cycle timing of processor pipelines or caches. Since building precise timing models of real hardware is very difficult, such detailed timing analysis has to be validated on the real target platform [11].

Simics provides *a single global virtual time*. All processors and device models are synchronized to this time base, across processors and machines in a simulated network.

The progress of this time in relation to the real-world time is variable. It is quite possible for a simulation of a slow system to be many times faster than the real world. Also, if a system is mostly idle, simulation can run very quickly. For example, we have run a network of 100 small sensor nodes[2] at five times real-world speed. In the tested case, the software tested had the sensor nodes spend about 99% of their time sleeping, making it possible for the total simulation to run faster than the real world, as very little processing was done in each node.

The simulated system might also run slower than a real system if the simulation is heavily loaded or contains many processors. It is obviously very hard to simulate ten high-speed processors as fast as real-time on a single host processor. In the most extreme case, simulation execution can be paused for some reason, in which case time does not advanced at all.

Real-time simulations where simulated boards are mixed with real-world systems are possible, as long as the simulated system is slow enough that the simulation will always run faster than real life. Then, the simulation can be stalled when it runs ahead of real-time, creating a simulation which runs at real-time speed [12].

2.4. System Model Creation

A key part of using Simics for a particular target system is creating the model of the target hardware. As noted above, Simics models the full system hardware, and thus more than just a processor ISS

[2] A ”Telos Mote” is a small wireless sensor node containing an 8Mhz 16-bit processor [14].

is needed in order to create a working simulated system.

First, any components already available for Simics can be reused. In processor arena, Simics has fast models available for all the most common embedded processor families, including PowerPC, MIPS, MIPS64, ARM, x86, and SPARC, along with system controllers, FLASH memories, IDE disks, I2C buses, network controllers, serial ports, timers, and other common components.

Second, any new components present in the system in question have to be modelled. This task can be performed by Virtutech, the Simics user, or by a third party. In order to speed and simplify the modelling of new systems, a domain-specific language called *DML* has been created for writing Simics device models. DML is many times more productive than using C or C++ for model construction.

2.5. Simics Uses

Simics is currently in use at many commercial customers and more than 1000 universities world-wide. Simics has been used for a large spectrum of activities, including computer architecture research & design, firmware development and test, operating system ports to new hardware, application development for distributed systems, and system testing for telecoms systems. Simics has been a commercial product since 1998. Version 3.0 shipped in October of 2005 [13], and development and enhancement of the product continues.

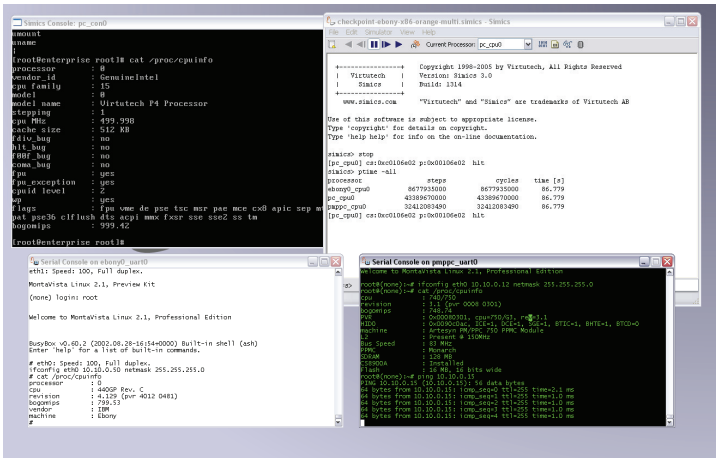


Figure 5: Screenshot of a Simics session

The technology has proven very flexible and scalable, being used for simulation of everything from single aerospace boards to networks of storage boxes to multiprocessor database servers and rack-based telecom systems with tens of processors.

As an example, Figure 5 shows a screenshot of Simics running a network containing one x86-based PC with Linux, one PowerPC 750-based embedded computer with Linux, and one PowerPC 440GP-based development card running Linux. Each target machine has its own text console available for interaction, and there is also the main Simics window from which the simulation is controlled.

In the remainder of this paper, we will primarily address the uses of Simics in testing of embedded software, with an eye towards general embedded software development.

3. Unit Testing

A straightforward use of Simics is to replace real target hardware for unit testing of programs. This is done either with or without deploying an operating system on the target machine, depending on the characteristics of the code and system being tested.

In the simplest case, a simplified target system is setup containing only a single processor ISS with memory (to store code and input data), and a special test output device. The program under test has to be able to run on a naked machine with no operating system, and to write its results to a “port” in memory.

Typically, the input data is loaded into the simulated machine together with the program, while the result is a stream of characters which is collected into a file on the host machine, and then analyzed in a separate post-processing phase. The proposed setup is illustrated in Figure 6.

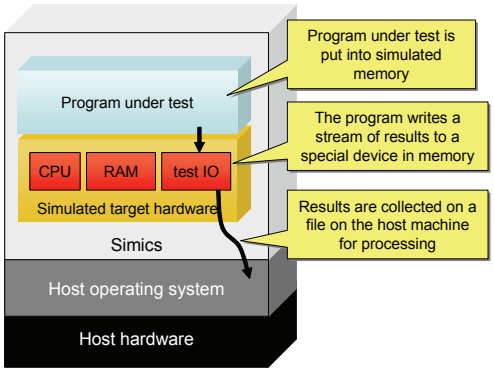


Figure 6: Setup for simple unit testing

In this setting, simulation is used as a replacement for development cards used to perform unit testing of binaries for a specific target architecture, without using any operating-system calls or particular input or output. The goal is to ascertain the correct function of a small unit of compiled target code, using some a test bench compiled into or with the program.

An alternative setup is to run an operating system on a complete simulated machine, mimicking the setup of a real development board. In this case, the simulated target will be connected using simulated networking to the driver application on the host, and interfaced and used just like a real development board. If the test cases use operating-system calls,

such a setup is obviously necessary. An illustration of this setup is given below in Figure 7.

3.1. Benefits of simulation

Using simulation for unit testing has several benefits compared to using real hardware.

First of all, loading and executing tests is more *convenient* than on real hardware. Code to be tested is put in place when the simulation starts by writing it directly to simulated memory. There is thus no need to download code over a serial line or network and running a monitor on the target.

The simulated hardware is also *perfectly controllable*; resetting the state of the target is achieved by simply restarting the simulation, without a need to physically touch any hardware. Test scripts can supervise the execution and use time-outs to kill off any tests that seem to have crashed and not produced any results.

Tests can be easily *automated and scripted*. A test is started by running a program or script on a host workstation, and the entire test runs under control from the simulator. A test engine can easily run a series of tests without user intervention. There is no need for manual intervention to reset the target hardware between tests or when a test crashes, thus freeing test engineer time for more productive work.

As test execution is performed by starting a regular software program (the simulator) on the host workstation, the execution of test suites containing many individual test cases can be easily *parallelized* across multiple host machines. As each test is run in isolation, we can expect perfect linear speedup in overall test suite execution time. The only overhead in such parallel simulation is the work involved in starting Simics on remote machines. This makes it possible to run large test suites faster than in real life, shortening test turn-around time.

Simics *determinism* makes it easy to run regressions tests. Any change in the output of an execution compared to previous executions of other versions of the same program can only be caused by differences in the tested program, not by hardware glitches or other random variations.

As pointed out above, Simics can also do *code coverage analysis* and *execution profiling* on the test cases, without instrumenting the code.

4. Function Testing

Broadening the scope from unit tests, the next step is typically to test a complete function consisting of several software units working in concert. Here, we assume that such testing is carried out for a single target board at a time (testing networked systems in a network configuration is addressed later).

For function testing, Simics is used with a full simulated machine running an operating system (and various supporting libraries or middleware systems, where such are employed), as illustrated above in Figure 1. Thus, the program under test is expected to be using OS and library API calls. Interrupt occurrences and OS scheduling will behave as on a real machine, allowing the test of tasks prioritization and execution modes. Memory management and usage can be tested, as well as the interaction between parallel tasks or threads in a system.

The testing can be driven by test bench code loaded with the program under test, or it can use external testing tools that communicate with the program. In both cases, testing is performed in the same way it would be on a real target board. The simulation does not necessitate changes to methodology or tool use for testing, it just changes the means used to execute the code.

For example, a networked target machine can be tested by connecting existing network testing tools to the simulated network, communicating with the simulated board just like with a real board. Both in the real and simulated cases, the testing tool is given a network address to communicate with (typically, an IP address + port number), and is thus oblivious to the nature of the target (whether it is real or simulated). The test program and simulated board with the program under test can both be run on the same host workstation, as illustrated in Figure 7.

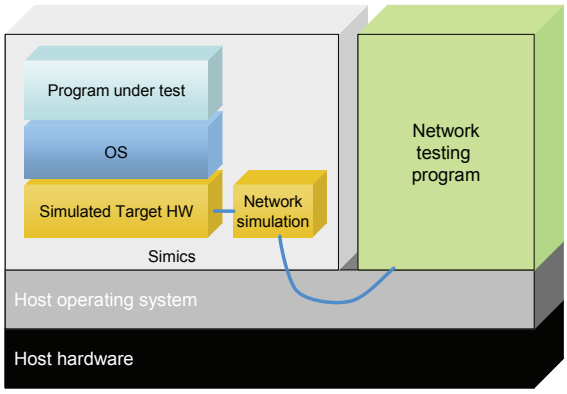


Figure 7: Network test on a single machine

Note some care regarding timing is needed when mixing real-world testers with simulated machines. Timeouts for determining when a crash has occurred in the program under test are usually set in terms of real-world time, and this is not always appropriate for a simulated system. As noted above in Section 2.3, the timing of the simulation is different from that of a real system. Thus, for best results, testing tools should use a time feed from the simulated system to determine time-out conditions based on the virtual time of the simulation [8].

4.1. Benefits of Simulation

All the benefits of simulation that apply to unit testing also apply to function testing. In addition, some new benefits apply.

Thanks to the easy configuration of a simulated system, *multiple versions of a run-time environment* can be used in testing. Different versions of the operating system and other supporting software can be canned as memory images or disk images, and instantaneously brought up for use.

Deterministic simulation offers a very powerful tool in a multithreaded environment. Rerunning a test case will result in the same sequence of interrupts, task switches, and inter-task communications every time a test is run. This greatly simplifies diagnosing errors found in testing, and in *communicating errors to the developers* from the test group. A failed test case will fail on the developer's workstation just like it failed in the test lab, making error reproduction trivial.

Testing can begin before the hardware is available. A common use of full-system simulation is to provide a development and software test environment for hardware in development, allowing for parallel software and hardware work for new generation systems; such use of Simics is common [2][7][15].

5. System Testing

Beyond the testing of individual units or functions, simulation can also be used to test the functionality and correctness of a complete system involving multiple processors and networked boards [2][3][7][8].

In this case, full-system models are constructed for all nodes in a network, and several target board models connected using a simulated network, as discussed in Section 2.2. The machine models themselves are the same as used for function testing, but typically using multiple instances of a particular type of simulated machine, with identical or different software loads.

In system testing, the simulation is used to execute a complete real software load, including multiple programs running in parallel and software relying on communicating with other machines in a network. A complete distributed system is easily simulated, providing a realistic environment for the software on each node in the system.

For embedded systems controlling physical systems, mechanical or physical simulations of the environment can be interfaced to the Simics simulation of the computer system. Such models rely on the virtual time in Simics, computing the evolution of the environment in the same time-domain as Simics executes the software.

In order to enable the simulation of large systems, Simics can distribute the simulation of a network across multiple host processors or host machines. This lets large simulation take advantage of added CPU cycles and memory resources to handle really large simulated systems [8].

A recent example of system testing with Simics is the simulation of Iridium satellites. Using virtual satellites, software is developed and tested on the ground in an environment corresponding to what it would meet in an actual satellite in orbit [16].

5.1. Stimulus

When doing full-system testing, the problem of how to inject stimulus to the system becomes very important. The goal is to execute the software in use cases taken from the real world, and this is realized by providing relevant stimulus as input to the system.

The *system configuration* itself can be the stimulus. One example of this is a real-world case where Simics was used to test self-configuration of large networks by simulating several hundred network nodes connected in various ways. Booting up all the machines on the network tested that they correctly elected leaders, obtained network addresses, and established communication.

A *simulation of the physical environment* provides good test input for control software, as discussed above.

In network simulation, stimuli can be provided by *traffic generators* in the classic sense that provide a stream of packets from a given distribution. It is also possible to use *behavioural models* of network nodes, i.e. small programs describing the network behaviour of a node without all internal details. Existing network testers can also be interfaced with the simulated system. Figure 8 shows how these types of stimuli are combined with regular Simics simulations running complete software stacks.

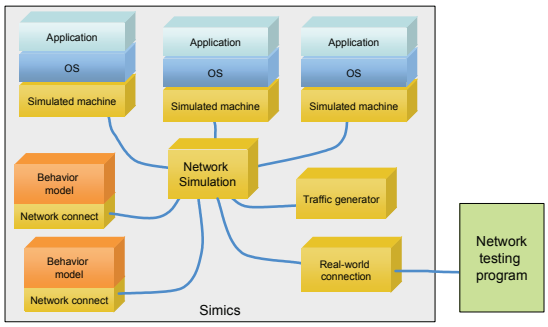


Figure 8: Network testing using a variety of simulation styles for different nodes

5.2. Benefits of Simulation

Using simulation, the *setup time for system-level tests can be greatly reduced*. Configuring a network using a set of scripts is much faster than plugging in cables and configuring network devices. Once a setup has been created, it can be stored and reused

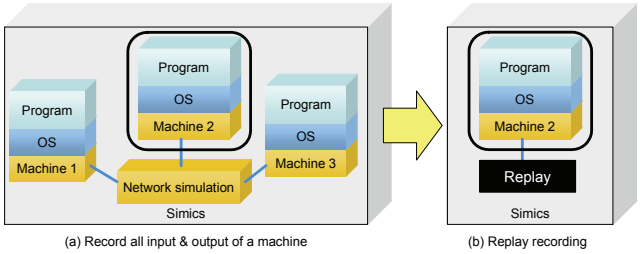
instantly. This saves significant amounts of setup and turn-around time.

The *test coverage will increase*, as more systems become available for testing thanks to the cost savings and flexibility of simulation (see also Section 7 below). Network tests are typically difficult and expensive to perform on real hardware, as each test requires multiple development or prototype boards and long system setup times. Also, network testing usually requires booking time in a special lab, which also makes turn-around times longer.

With simulation, hardware availability is no longer a problem, system setup is much faster, and tests can be performed on any workstation, making it much easier and cheaper to perform full-network tests.

Just like for function testing, deterministic simulation *makes it easier to communicate between testing and development groups*.

Since simulation provides full control over all data exchange and input and output of a system, it is also possible to record and replay input and output (network traffic, user interaction, other external events) in a simulation. This makes it possible to perform *isolated error diagnosis* on a single machine in a multi-machine setup, by recording the whole system and replaying the external events for a single machine in the system.



(a) Record all input & output of a machine (b) Replay recording

Figure 9: Record and replay in a network

Figure 9 illustrates this idea. Machine 2 in the three-machine network has all its input and output recorded, later to be replayed using just a simulation of machine 2. This can also be done for all machines simultaneously, allowing for analysis of the networked behaviour of each machine in isolation following a single networked simulation run.

6. Fault Injection Testing

Simulation can also be used to test the behaviour of a system in the presence of faults [9][10].

Fault injection in simulation requires the creation of appropriate fault-injection modules inside the simulated system, modules that are attached to buses, networks, memories, devices, and processors, and perform the actual injection of faults.

These fault-injection modules have to be tailored for each component, but it is easy to reuse models for memory components, processors, networks, and similar common devices. In addition, it is necessary to establish a way to specify which faults occur when, so that fault injection campaigns can be specified and executed using a single point of control. Usually, a master fault injection module is used which reads a file specifying which faults are to be injected where and when.

Some faults can only be reasonably studied on real hardware, for example electrical effects of pulling cables and cards out of racks and radio disturbances caused by interference from electrical systems. Similarly, radiation effects on chips have to be tested in irradiation chambers.

However, once physical studies of fault behaviour have been performed, simulation can be used to replicate the effects on the computer system. Data from the physical experiments, such as the frequency and nature of transient and permanent faults, can be used to guide the faults to be tested in simulated fault injection campaigns.

Fault injection in simulation is typically used to check that fault detection and recovery mechanisms work as designed. This might mean testing voting mechanisms or redundant failover, or just that a system correctly logs errors. Fault injection can also be used to diagnose problems – it allows an engineer to test a hypothesis as to which hardware problem causes a particular software error.

6.1. Benefits of Simulation

Simulation has a range of advantages for fault injection studies compared to using real hardware.

Simulated fault injection is *non-destructive*. The system under test does not suffer permanent damage from being tested with faults, unlike physical experiments where hardware components are quite often damaged (intentionally and unintentionally) in testing.

Simulation offers *repeatability* of fault injection, as replaying the same fault injection script will inject the precise same faults and the precise same point in time. Achieving precise repetition with physical fault injection is very difficult.

Fault injection in simulation will allows *increased fault-injection coverage*, as faults are easy to program and introduce, and fault-injection campaigns are simple to execute; just run the simulator, no special lab needed.

Since faults are easy and precise to program and tailor, *corner-case testing* is enabled. With the help of simulation, it is thus possible to script and repeat known hard cases in the system, such as multiple board failures close in time and babbling idiot failures.

7. Business Aspects

Using simulation for testing in a real production environment requires considering the business aspects of simulation as a testing tool.

7.1. Costs

The costs involved in introducing simulation consist of four parts. First, the development of the simulation framework, which should not be needed as it can be bought off-the-shelf.

Second, the cost of developing the necessary hardware models to model the system or systems of interest. Depending on the nature of the system, many standard parts are typically available from the simulation vendor. Developing new standard parts of custom parts requires a one-time investment by the vendor or the customer.

Third, there is a licensing and continued support cost associated with the simulation tool.

Fourth, there is an initial need to train users on the simulation tool and introduce the simulation into the workflow. As we have already noted, simulation in testing can often slip into an existing workflow, using the same methodologies and tools as used with real hardware. Full-system simulation really does provide virtual hardware that can be used instead of real hardware.

7.2. Benefits

The benefits from introducing simulation are both direct costs savings, and indirect economic benefits from faster development and better quality products.

First, buying a single simulation license is usually cheaper than buying a single development card or custom product board. Since several boards can be simulated in a single simulation instance, there are obvious cost reductions from simulation. These cost savings can either translate to lower overall costs or more systems deployed at the same cost.

Simulation also provides increased flexibility. There is no fixed inventory of available boards. Any users can setup any board or combination of boards, without allocating physical boards. Any workstation can be used to run target code for any target system, greatly improving hardware availability for developers and test engineers. There is no need to ship hardware around to supply each developer with the particular hardware needed at a given moment.

Second, simulation makes it possible to develop software in parallel with the hardware, and to deploy more hardware earlier in the development cycle. This translates to a shorter time to market for new products, as the software teams can start working earlier than if they have to wait for hardware. Removing the dependence on hardware also means that more developers can work in parallel on the same project, as hardware availability is no longer a bottleneck.

Third, simulation brings a number of technical benefits for the actual execution of tests, as detailed above. These cumulative benefits lead to better product quality thanks to more and better testing and easier fault identification and correction. Shorter setup times and turn-around times, as well as more convenient execution environment makes test engineers more efficient.

Our experience is that the cost of introducing a simulation are usually far outweighed by the advantages it offers, especially for companies building and integrating complex embedded systems based on custom hardware and with large value deriving from the software running on these systems.

8. Summary

This article has presented the use of simulation in general (and the Simics simulation product in particular) for testing and developing embedded software.

Simulation does not replace all testing on real hardware, but it offers a very good complement to real hardware, with potential for great improvements in development and test process efficiency and overall cost. Simulation can be used within existing work flows and tool environments, making such benefits quite easy to realize.

We have reviewed a number of typical usages of simulation for embedded software testing, and for each case, how the simulation would be setup and the resulting benefits.

9. Acknowledgement

The authors would like to thank all our colleagues at Virtutech for having created a great tool and applied it to real-world problems.

10. References

[1] S. Gill: "*The Diagnosis of Mistakes in Programmes on the EDSAC*", Proceedings of the Royal Society of London. Series A, Mathematical and Physical Sciences, Vol. 206, No. 1087, May 1951.

[2] P. Magnusson, M. Christensson, J. Eskilson, D. Forsgren, G. Hållberg, J. Högberg, F. Larsson, A. Moestedt, B. Werner. "*Simics: A Full System Simulation Platform*", IEEE Computer, Feb 2002.

[3] A. Alameldeen, M. Martin, C. Mauer, K. Moore, M. Xu, D. Sorin, M. Hill and D. Wood. "*Simulating a $2M Commercial Server on a $2K PC*", IEEE Computer, Feb 2003.

[4] M. Stetter, J. von Buttlar, P. T. Chan, D. Decker, H. Elfering, P. M. Gioquindo, T. Hess, S.

Koerner, A. Kohler, H. Lindner, K. Petri, and M. Zee: "*IBM eServer z990 improvements in firmware simulation* ", IBM Journal of Research and Development, Vol. 48, No. 3/4, 2004

[5] J. Connell, B. Johnson: "*Early Hardware/Software Integration Using SystemC 2.0*", Proc. Embedded Systems Conference, San Francisco, USA, 2002.

[6] A. Möller, "*A Simulation Technology for CAN-based Systems*", CAN Newsletter, Dec 2004.

[7] P. Magnusson: "*The Virtual Test Lab*", IEEE Computer, May 2005.

[8] J. Engblom, D. Kågedal, A. Moestedt, and J. Runeson: "*Developing Embedded Networked Products using the Simics Full-System Simulator*", Proc. 16th IEEE International Symposium on Personal Indoor and Mobile Radio Communications (PIMRC 2005), Berlin, Germany, Sep 2005.

[9] Myhrman and Svärd: "*Studying Fault Injection in WCDMA Base Station Processors Using Simics Simulator*", MSc Thesis, Chalmers Institute of Technology, Department of Computer Science and Engineering, 2005.

[10] B. Bastien: "*A Technique for Performing Fault Injection in System Level Simulations for Dependability Assessment*", MSc Thesis, University of Virginia, School of Applied Science, Jan 2004.

[11] J. Engblom: "*On Hardware and Hardware Models for Embedded Real-Time Systems*", Proc. IEEE Workshop on Real-Time Embedded Systems (WRTES 2001), London, Dec 2001.

[12] J. Engblom and M. Nilsson. *Time Accurate Simulation: Making a PC Behave Like a 8-bit Embedded CPU*, Technical Report 2002-024, Dept. of Information Technology, Uppsala University, 2002.

[13] D. McGrath: "*Virtutech system-level simulator features Hindsight technology*", Electronic Engineering Times (www.eetimes.com), Oct 4, 2005.

[14] Moteiv Corporation, *Telos (Rev B) Datasheet*, May 2004.

[15] A. Ernst: "*New CEO John Lambert on Virtutech Present and Future*", Virtual Strategy Magazine, Oct 4, 2005

[16] J. K. Waters: "*Iridium Simulates Space Software with Simics*", Application Development Trends Magazine (www.adtmag.com), Oct 31, 2005.

11. Glossary

API Application-Programming Interface

DML Device Modelling Language

HW Hardware

ISS Instruction-Set Simulation

OS Operating System

SW Software