# SIMICS: A COMMERCIALLY PROVEN FULL-SYSTEM SIMULATION FRAMEWORK

**Jakob Engblom and Dan Ekblom**

*Virtutech AB*
*Norrtullsgatan 15*
*113 29 Stockholm*
*Sweden*
*email:* jakob.engblom@virtutech.com *,* dan.ekblom@virtutech.com

## INTRODUCTION

This paper presents Virtutech Simics, a commercial off-the-shelf full-system simulation framework used to enable *virtualized software development* for cross-development targets on a regular PC or workstation. Simics provides *virtual hardware* that runs the same binaries as the physical hardware. The goal of virtualized software development is to enhance the software development process by reducing the reliance on physical target hardware and still test and develop as much as possible in the actual target environment.

From a technology point of view, Simics enables virtualized software development by being a *full-system simulator [1]*. In full-system simulation, you combine a fast instruction-set simulator of your target with models of all components in the physical hardware. Thus, the software experiences a virtual computer system that is functionally identical to the physical system, capable of running the same binaries, including device drivers, operating systems, and applications.

Simics is designed for fast execution of code on the virtual system, typically reaching several hundred millions of simulated instructions per second, allowing full workloads to be executed. Simulation speed can be accelerated even more when a system is idle or sleeping, potentially allowing days of simulated time to be completed in just a few minutes of wall clock time. The key tradeoff to enabling high-speed execution in simulation is that Simics abstracts and approximates the timing of the physical hardware. Thus, Simics is a suitable platform for testing functional aspects of space-bound software, which in our experience makes up 80-90% of the development, validation, and test work for real-time embedded systems. Especially for future space missions with large software loads and many fast on-board processors, Simics offers a feasible path for high-performance system simulation for software development. Currently, Simics exceeds the speed of physical LEON 2 processor implementations.

Simics offers a convenient software test and debug environment, with features like instant stop of execution, scripted test cases, full determinism, checkpoint and restart, as well as reverse debugging and reverse execution. As a simulator, Simics offers complete control over the simulated computer systems, enabling precise injection of faults and scripting of tests. Faults can be injected both inside devices, memories, and processors, and on the buses and networks connecting them. Simics includes a full Python language interpreter, enabling powerful scripting.

Virtutech has a large library of simulated components available for users to use to construct system modeling, which reduces the time to build a model of a particular system. Examples include the SPARC V8/Leon2, PowerPC 750, and other processors. There are also models of common system components such as serial ports, SDRAM memories, FLASH memories, and system controllers. Various buses and networks are available, including MIL-STD 1553, ARINC 429, and Ethernet [15]. Simulated buses and networks can be connected to real-world networks for mixed simulations involving both virtual and physical nodes.

Models of new hardware components are developed by Simics users or by Virtutech in a very efficient programming language called *DML, Device Modeling Language*. Users can extend Simics arbitrarily, using a well-tested and powerful general simulation API. User extensions can be both new hardware models and new simulation system capabilities like inspection and fault-injection modules. Thanks to DML and the large library of components available from Virtutech, Simics offers the ability to quickly build models of hardware. Models can even be built in advance of physical hardware availability, enabling software development to start before prototype hardware is available.

Simics fully supports simulation setups containing multiple processors, multiple boards, or even multiple space vehicles, all within a single Simics process running on a single workstation. Different types and speeds of processors can be freely mixed if desired, with no limitations. It is possible to simulate both a central computer unit and the payload computers of a satellite system within a single simulation, if that is desired.

Simics is a proven, stable, and efficient simulator framework, which has seen extensive use in both industry and academia. It has been commercially available since 1998, and it is currently in use at many commercial customers, including several companies in the US space, defense, and civilian aviation markets. The paper will present how Simics works, how it is used, and how it has been applied in the real world.
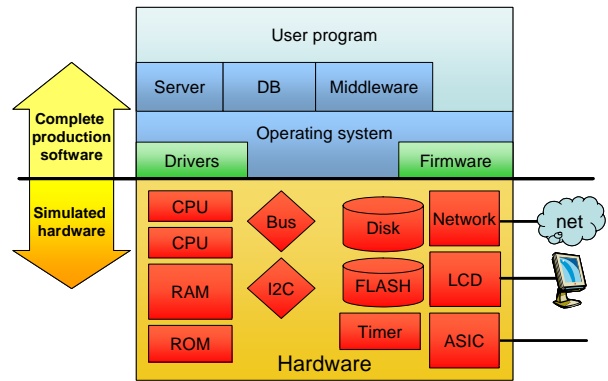


Fig. 1. Simics simulates all hardware components of a system, allowing the unmodified software to run just like on the physical hardware.

## VIRTUTECH SIMICS PHILOSOPHY AND TECHNOLOGY

The goal of a Simics virtual system is to make the software believe it is running on the physical hardware. To achieve this, the model has to represent the properties and behavior of the hardware at the level where the software talks to the hardware. At this *hardware-software boundary*, the devices in the system show up as registers at certain locations in the processor address space, and processors execute code as specified by the instruction-set architecture (ISA) of the processor. Device drivers in the software running on the processor(s) read and write the addresses exposed by the devices to make the devices perform, just like on real hardware.

As schematically illustrated in Figure 1, the Simics model of the target system contains the same components as the physical hardware. Typically, a simple computer system will include a processor, some FLASH, ROM, and RAM memory, and basic devices needed to run an operating system like timers, serial ports, and some high-speed communications channel. The code running on Simics is the production code, produced using the same build variants and compilation tools as code destined for the physical target hardware. There is no need to use multiple different builds to support software development in a virtual environment.

All the devices and all interconnections in a system are implemented in a transaction-based style, typically known as *transaction-level modeling (TLM)*. This means that accesses to devices are handled as synchronous units that complete immediately, rather than simulating the actual details of the bus traffic (on the processor and peripheral buses) required to perform the request in the real hardware. Time delays for operations that take significant time in the hardware system are modeled by delaying the completion of an operation for some period of time. Simics has an internal event queue system which allows posting events at an arbitrary point in time in order to model such delayed completions.

Using this particular level of abstraction is a key factor in creating a simulator fast enough to be used for virtualized software development. More detailed models of the hardware implementation and timing would slow the execution down so much that it would not be useful for large workloads. Creating a simulation layer at higher levels of the software stack by emulating the behavior of an operating-system or middleware application-programming interface (API) has several drawbacks as well. Such an API simulation will not use the real target compiler, will not run the real target operating system and its process scheduler, and also requires a separate build chain and build variant. Such variant builds cost time and money to create and maintain [11]. There is also a significant cost to maintaining an API-level simulator in that it has to be updated every time the real system changes its API or behavior.

The Simics hardware model is accurate at the hardware/software boundary, so that the object code that runs on the physical hardware will run on the model (creating what is fundamentally a piece of virtual hardware). All operations performed by the software will have the same result, bit-by-bit, on both the virtual and the physical hardware. The main exception to this rule is reading timing-sensitive and system-profiling registers which are not modeled in detail, as discussed below. It is also possible to introduce deliberate differences between the hardware and the model, such as a register bit that is zero on the physical hardware but forced to one in the model.

Modeling a system at the hardware-software boundary has several practical advantages:

- This level is often the best documented and most stable layer in the system, since it is where hardware vendors interface to software authors.

- The processor instruction-set architecture is quite stable over time, making it easy to reuse existing processor models for new processors implementing the same instruction-set architecture (ISA). This makes it possible to create heavily optimized processor models, which is central to achieving high execution speed in the virtualized environment.

- It is the natural integration point between hardware and software engineers, which corresponds to how companies and the market place for semiconductor components are usually structured.

- Device simulation performance is optimal because device functionality is implemented with the minimal amount of state change needed.

- Device modeling times are minimized since only the device functionality actually used by the software needs to be implemented.

- Device models can often be constructed with only the details provided by a programmer's reference manual, reducing the need for communications between software and hardware teams.

- Low-level hardware implementation details such as those defined in VHDL, Verilog, or SystemC are not visible, making modeling much simpler and faster. It also makes the models execute faster.

**Simulation Timing**

In a Simics model used for virtualized software development, the functional results of instructions and device accesses are identical to a real machine (which is necessary in order to run real binaries), but the timing might be different. As stated above, this simplification of timing is necessary in order to achieve a sufficiently high speed of execution in the virtual environment.

On the processor side, Simics does not attempt to model the precise cycle timing of code execution as dictated by the processor pipeline structure, cache hierarchies, and memory bus contention. Such detailed processor models are both hard to build and necessarily slow to execute since they contain much detail. The speed of a processor executing instructions in Simics is set by the processor clock frequency and the cycles-per-instruction (CPI) property of the processor. Typically, setting CPI to one, i.e., executing one instruction every cycle, works well. Setting CPI to other values like 2 or 3/2 can help Simics more closely approximate the average instruction execution speed of a physical processor. Experience shows that this works well for almost all workloads in almost all circumstances.

At the interconnection buses, the detailed arbitration and data transport mechanism of the memory and system buses (the AMBA bus in the case of the LEON2) are not modeled explicitly in Simics. The modeling abstraction used is rather a functional memory map, where read and write operations to an address are directly routed to the memory areas or devices mapped at that address. This can be performed in a very efficient manner, ensuring high simulation performance. All memory operations complete immediately, which is natural in a transaction-level model like that employed in Simics. It is possible to collect complete memory and device access traces with Simics for later analysis of bandwidth requirements, but Simics does not limit bus bandwidth during simulation.

For devices, most operations complete in a single transaction, effectively in zero time as seen by the processor and the software. For operations where this is not possible, timing is approximated by using fixed, configurable, or computed delays. A classic example is a device driver that writes to a device that will later trigger an interrupt. The driver expects to be able to run some more code after writing to the device, before the interrupt is triggered. Thus the device model must delay before generating the interrupt. Concretely, a network card would immediately deliver a packet to the network, but would wait until some microseconds later to raise the interrupt marking the completion of the operation. The time to delay the interrupt could be computed based on the size of the packet, or set as a configuration parameter to the model.

Such simplified timing has many advantages. First, the resulting models are very fast, which is necessary to be able to execute long software runs on large computer systems containing multiple processors and many millions of lines of code. Second, the models can be created quickly since no implementation details are needed, only the specification of the behavior. Third, the models can be reused wherever the same hardware/software interface is exposed, regardless of whether the implementation is actually the same. Fourth, a device model can be created in parallel to the physical hardware implementation and realization. The model and the hardware implementation are both built from the same functional specification but the model building does not need to know how the hardware design implements the specification. Thus, the two activities can be performed in parallel.

We should also note that building precise and valid timing models of complete hardware systems is very difficult. Many attempts have been made to create "cycle accurate" models of hardware, especially for processors. In most cases the results indicate that complete cycle-accuracy cannot in practice be achieved even for simple in-order embedded processors with no caches [2][4][6]. For more complex processors, it is even more difficult [3][5][7]. The reasons are many, including the fact that manuals are notoriously sketchy on timing details (in order to preserve implementation secrets), and that validating all the details involved is exceedingly difficult. Even computer architects themselves are turning to more approximate simulators aimed at estimating average throughput rather than precisely simulating the exact number of cycles required to execute small pieces of code [8]. Accurate models are only needed for studies of the processor architecture itself or first-level caches, not for caches beyond the first level or the memory system [9].

Going outside the processor, most components exhibit hard-to-model timing variations that may depend on analog factors in board design or skew between independent clock domains. Thus, absolutely accurate system models remain an elusive goal. In general, "cycle-accurate" simulation is a good tool for predicting and evaluating the relative performance of design alternatives when designing systems, but not a tool to be trusted for absolute performance measurements or validation of system-level real-time behavior.

In our experience, the preferred methodology is to iron out functional bugs related to communications between machines, logic errors in the code, and similar issues that are not dependent on the detailed timing of the hardware using a simulated environment. This covers 80 to 90 percent of all software problems. The remaining, timing-dependent, issues in the software have to be found and dealt with on the physical hardware corresponding to the actual flying system, since that is the only true representation of the timing.

**Execution Speed of Simics**

The level of abstraction chosen in Simics makes it possible to execute simulations very quickly. Virtutech provides fast processor simulators based on just-in-time compilation technology that make it possible to run target code and many hundreds of MIPS on a regular PC. The precise execution speed will vary with the nature of the workload, since input and output operations (I/O) take more time to simulate than simple arithmetic instructions. Figure 2 shows the execution speed of Simics simulating a LEON2 processor on a 2400 MHz AMD64 host machine, as published previously [15].

It is also quite possible for a simulation to run many times faster than the real world. In particular, if a simulated system goes into sleep mode or halt mode, Simics can look ahead in the event queue and instantly move time forward to the next interesting event. Thus, if a system is idling most of the time, simulation of very long executions can be run in a fraction of the time. For a system with short power-on periods and long sleep times (like a deep space probe on its journey out), this makes it possible to simulate many days of mission time in a much shorter real-world time. It also makes it feasible to simulate very large clusters or constellations containing many computers. In our experience, most distributed and parallel systems are idling quite a significant portion of their time. This is shown with great effect in the CPPemu cluster simulator, where Simics is used to provide a reasonable execution speed for a simulated telecom system containing 20 processors in a typical configuration – on a single PC
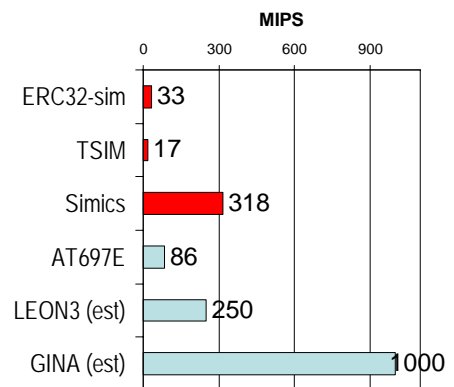


Fig. 2. Simulation speed for SPARC v7 and v8 simulators, compared to the execution speed of physical hardware, from [15]

[11]. Parallel simulation of many different test cases can also be used to reduce the overall execution time for a series of tests [12].

**Determinism**

Simics is designed from the ground up to be a deterministic simulation system. This means that no device or processor model in the system is allowed to have internally non-deterministic behavior, and that the behavior is the same across different host systems. A simulation run on a 32-bit PC host with Windows will give the same result as a simulation on 64-bit SPARC host with Solaris. Together with checkpointing, determinism is a powerful enabler for other features in the simulator like (reverse) debugging, and also helps in debugging Simics itself.

Each time a simulation in Simics is run from the same initial state, precisely the same final state and results will be computed, provided that all input and output to the simulator is controlled and identical. It is possible to record interactions with a user or other entity external to the simulator to ensure this property in all circumstances. In the case where the system being simulated is self-contained (for example, driving tests using test scripts within the simulator or testing a boot of an operating system), determinism is automatic even without recording.

In some cases, variation in the target machine is a desirable property. For example, testing often needs to examine the behavior of the system in a range of circumstances and states. In physical hardware setups, such variations are often introduced by re-running the same tests many times over, counting on randomness in the real world to provide variety. In Simics (and other deterministic simulators), variation has to be programmed in. Such variation is usually pseudo-random from a random seed, so that the same test cases can be reproduced by using the same random seed. Thus both coverage of a wide variety of circumstances and precise reproducibility of tests can be achieved. Failed tests are trivial to reproduce and subject to detailed analysis and debugging. Simics determinism ensures that no other variation than that programmed into the system will occur, removing uncertainties from the analysis.

**SOFTWARE DEVELOPMENT WITH SIMICS**

Simics has been specifically designed to facilitate software development and test on virtual hardware, i.e. virtualized software development. A number of important features and properties of Simics are derived from this design goal. As discussed above, Simics is guaranteed to be *deterministic*, which helps in reproducing and analyzing errors, as well as in reliably injecting faults and known variations in the target.

Simics supports *checkpointing*, where the complete state of a system is stored to disk. This checkpoint can later be loaded into Simics, putting the target system at the precise instant and state where the checkpoint was saved. Checkpointing is useful to instantaneously get a system to a booted state, for example, or to get back to the starting point for a unit test to try different input sequences. It makes possible to package a bug case and distribute it to multiple developers for simultaneous investigation.

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

*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 target machine and to events inside the target (breakpoints, exceptions, control-register writes, device accesses, etc). Scripts can be used to provide input to the target, or to perform smart logging of events or even triggering further breakpoints. Scripting is also used in fault-injection to program fault scenarios and perform the actual injection of faults.

*The complete target state is accessible, without probe effects*. Information which is hard to obtain on physical hardware, such as translation lookaside-buffer (TLB) tags in a memory-management unit (MMU), the contents of supervisor-level registers, and internal device state registers are all easy to observe in Simics. Observation is achieved without running code on the target or disturbing it any other way. Every state change in the target system can also be traced and logged. Fundamentally, Simics lets you peek under the hood of your system.

Tracing can be used to *profile* and perform *code coverage* analysis without having to instrument the target code. Even very detailed code coverage metrics 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.

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. Debugging can be performed using the Simics built-in debug facilities, or using an external debugger connected to Simics. In this way, Simics lets you reuse existing tools – if desired, Simics can appear just like a physical target system to your debugger, requiring very little change to the software tools setup.

Simics can do *reverse debugging*, where execution is practically reversed in time [10] – even for a multi-board, multi-processor system. Such abilities have been available for a time for development boards, using hardware trace recorders from vendors like GreenHills, IAR, and Lauterbach. With Simics, it can also be done for virtual systems, without the limitations of hardware recorders. You can set breakpoints not just on the next write to a variable, but also on the previous write. If a bad pointer has crashed your program, you can "un-crash" it and investigate the state and chain of events leading up to the crash. In simulation, reverse debugging is possible for multiprocessor systems and networked systems, including the effects of the software on the devices in the system

Simics has also proven to be a more convenient development environment than development cards. Each developer can have an arbitrary target machine or network of machines available at their desk and do not have to walk to the hardware lab to perform tests on hardware. Downloading a new version of code to a target is instantaneous, since it can simply be immediately put into the memory of the simulated machine. This can save significant time in edit-compile-test cycles. A Simics model is stable and reliable, unlike physical hardware (especially in prototype form), which is often flaky, making it hard to tell if a problem is in the software or the hardware.

**MODEL CREATION**

A key part of using Simics for a particular target system is creating the model of the target hardware. This process is typically quite fast, thanks to a number of factors. First, any components already available for Simics can be reused. Second, existing models of similar devices can be used to provide a good starting point for a new model. Third, DML enables any user, third-party consultant or Virtutech engineer to quickly model new devices.

Virtutech has a large library of simulation components available for users to construct system modeling, reducing the time to build a model of a particular system. Examples include the SPARC V8/Leon2, PowerPC 750, and other processors. There are also models of common system components such as serial ports, SDRAM memories, FLASH memories, and system controllers. Various buses and networks common in the space industry are available, including MIL-STD 1553, ARINC 429, and Ethernet [15]. Virtutech components can often be modified to quickly create models of devices in the same family.

Any new components present in the system to be virtualized have to be modeled. 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 has been designed specifically to support the efficient creation of device models for virtual hardware. As such, the language offers syntactical support and features particular to the needs of the functional device modeling domain. In DML, the interface of a device towards the software is described as a series of register banks, each of which can be mapped at a different location in the memory or I/O space of a processor. Inside each bank, registers are specified with their size and offset from the start of the bank. A register can be from one to eight bytes large, and may be divided into fields consisting of one to sixty-four bits. Fields can be specified in both big-endian and little-endian bit orders, allowing the specification of fields to follow the programmer's manual for a device regardless of the bit-ordering convention used by the device supplier.

Models can be created iteratively, starting with a stripped-down system or single part of a system, and adding devices and details over time to quickly get a useable virtual hardware system.

### SIMULATOR HISTORY AND CUSTOMER CASES

Simics is being used for software development in industries such as satellite systems, public aviation, defense, telecommunications, data communications, and servers. Simics has been a commercial product since 1998, and the current release is version 3.0. In the US space market, the Simics model of the BAE RAD750 radiation-hardened space computer system is being used by several vendors, targeting a variety of space missions. This section will give examples of how Simics has been used at various customers.

### Iridium Satellite

Simics is used by the Iridium Satellite Company to create a model of the Iridium satellite constellation. Since Iridium has no access to physical satellite hardware on the ground, they depend on simulation to provide a development platform for software to be uploaded to the satellites already in orbit. The model is used for unit testing of software, eliminating the need for hardware in this process. Iridium is also able to examine how both existing and new flight software operates on a satellite, and to debug software problems in a convenient desktop environment. Simics has been used to recreate performance issues for in-depth inspection and diagnosis. The simulation model was mainly created by Iridium engineers, based on models of standard components like processors and memories supplied by Virtutech [13].

Iridium demonstrates value of modeling an existing hardware system for software maintenance and new feature development.

### NASA Gamma-Ray Large Area Space Telescope

Virtutech Simics is being used by General Dynamics in their work on the NASA Gamma-ray Large Area Space Telescope (GLAST). Here, Simics is being used to train operators for the mission on a virtual space system. Using fault-injection in the simulated hardware makes it possible to practice operations when problems occur during the mission. The effect of actions taken by operations staff can also be tested in the simulator, allowing feedback loops between operator actions and mission software to be explored prior to launch.

The NASA GLAST case shows how simulation can be used not just for software and hardware development, but also for training purposes.

### Switchcore

Simics accelerates new product development at Switchcore, a Swedish company creating high-speed Ethernet switches. Simics is used to model the next generation Xpeedium3 switch chips so that development of the supporting software stack can start well ahead of availability of prototype hardware. Simics is used both internally at Switchcore and at Switchcore customers, instead of the more usual development card. By seeding important OEM customers with simulated hardware, Switchcore can reduce the time-to-market for products based on their chips [14].

Switchcore demonstrates how simulated hardware is used to shorten the total development time of a system, by providing software developers with an early start. It also provides an example of using simulation to provide development systems to external customers for a hardware company.


### SUMMARY

This paper has presented Virtutech Simics, a commercial off-the shelf full-system simulation platform that enables virtualized software development. Virtual hardware can replace and augment physical hardware for most parts of the embedded software development process, producing higher quality software with shorter development times. Simics is a proven tool with a long commercial history and several public customer cases demonstrating how Simics adds value to all parts of a development project, from accelerating software development before hardware exists to training operators for a mission to supporting maintenance of existing system.

## REFERENCES

[1] 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*, February, 2002

[2] P. Atanassov, R. Kirner, and P. Puschner, "Using Real Hardware to Create an Accurate Timing Model for Execution-Time Analysis", *IEEE Workshop on Real-Time Embedded Systems (WRTES 2001)*, December 2001.

[3] B. Black and J. P. Shen, "Calibration of Microprocessor Performance Models", *IEEE Computer*, pp 59-65, May 1998.

[4] J. Engblom, "On Hardware and Hardware Models for Embedded Real-Time Systems", *IEEE Workshop on Real-Time Embedded Systems (WRTES 2001)*, December 2001.

[5] J. Gibson, R. Kunz, D. Ofelt, M. Horowitz, J. Hennessy, and M. Heinrich, "FLASH vs. (Simulated) FLASH: Closing the Simulation Loop", *Proc. 9th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS00)*, November 2000.

[6] S. Montán. *Validation of Cycle-Accurate CPU Simulator against Actual Hardware*. Master's thesis, Department of Information Technology, Uppsala University, October 3, 2000. Also Technical Report 2001-007, http://www.it.uu.se/research/reports/2001-007/.

[7] "Meet the experts: The Mambo team on the IBM Full-System Simulator for the Cell Broadband Engine processor", IBM DeveloperWorks, November 22, 2005. http://www-128.ibm.com/developerworks/power/.

[8] J. D. Davis, C. Fu, J. Laudon, "The RASE (Rapid, Accurate Simulation Environment) for Chip Multiprocessors", *Workshop on Design, Architecture, and Simulation of Chip Multiprocessors* (dasCMP 2005), November 13, 2005.

[9] D. Wallin, H. Zeffer, M. Karlsson, and E. Hagersten, "VWasa: A Simulator Infrastructure with Adjustable Fidelity", *Proc. 17th International Conference on Parallel and Distributed Computing and Systems*, November 2005.

[10] C. Maxfield. Cool EDA Products light up 2005, *EETimes*, March 13, 2005. (http://www.eetimes.com/showArticle.jhtml?articleID=159403011)

[11] M. Bergqvist, J. Engblom, M. Patel, and L. Lundegård, "Some Experience from the Development of a Simulator for a Telecom Cluster (CPPemu)", *Proc. 10th IASTED Conference on Software Engineering and Applications, November*, 2006 (to appear).

[12] J. Engblom, G. Girard, and B. Werner, "Testing Embedded Software using Simulated Hardware", *Proc. ERTS 2006: Embedded Real-Time Software*, January 2006.

[13] F. Morring, "Iridium Tests Satellite Software on the Ground", *Aviation Week and Space Technology*, March 6, 2006.

[14] K. Morris, "Commercial Virtuality – Learning from Switchcore and Simics", *Embedded Technology Journal*, June 13, 2006.

[15] J. Engblom, M. C. W. Holm, "A Fully Virtual Multi-Node 1553 Bus Computer System", *Data Systems in Aerospace (DASIA)*, 2006.