TRANSPORTING BUGS WITH CHECKPOINTS

Jakob Engblom

Wind River, Stockholm, Sweden. jakob.engblom@windriver.com

ABSTRACT

Virtual platform checkpointing is a technology by which the entire state of the software and hardware of an executing system is captured. Checkpoints can be used for many different purposes during software (and hardware) development. Here, we focus on how they are used to transport bugs between bug reporters and developers. Compared to describing the system state and steps needed to reproduce a bug using natural language in a bug tracking system, checkpoints provide an executable and complete description of the system state. Together with tools that let us reproduce the final steps to provoke a bug, we get a system that can precisely and efficiently transport bugs from a bug reporter to the responsible developers.

Index Terms— *Debugging, virtual platform, checkpointing, bug reporting*

1. INTRODUCTION

One of the hardest problems in debugging is to correctly and reliably reproduce a bug found by someone else. Typically, test departments and other users of software create long and brittle "instructions to reproduce the error" in bug tracking systems. Accompanying the bug report is a list of relevant facts such as the version of the software, the OS version on the machine, the nature of the hardware, any attached external hardware, and anything else deemed relevant.

More often than not, the developer has to iterate a series of questions with the reporter to get more information about the system where the bug manifest itself and the precise steps taken to cause the bug to trigger. Such iterations can take days for a globally distributed development effort. The questions are often hard for the reporter to answer in the way the developer wants. In some cases, developers can do remote logins to the failing system in order to get their hands on the precise failing setup, but more often than not this is not possible due to security restrictions or the fact that the failing system is no longer available or has been used for some other important test. For embedded systems the problem gets compounded by the availability of the precise hardware needed to run the software and reproduce a bug.

A crucial aspect of bug reporting is the reproduction of the issue. If the developer fails to reproduce the bug in a report, it is usually impossible to fix.

This paper describes our experience in using virtual platform checkpointing to transport bugs from a reporter to a developer in order to achieve both reliable bug reproduction and ease of investigation.

We assume that both the bug reporter and the developer have access to a virtual platform of the target system, and using it for software development and test. The reporter runs the software on the virtual platform and hits a bug. By using a checkpoint of the combined hardware and software state, the reporter provides the complete target system state to the developer, as illustrated in Figure 1 (the person with the "D" shirt is the developer, and the "R" shirt is the reporter). With the checkpoint R, the developer can reliably reproduce the issue and investigate the state of the failing system. This is easy in principle but requires some thought in practice.

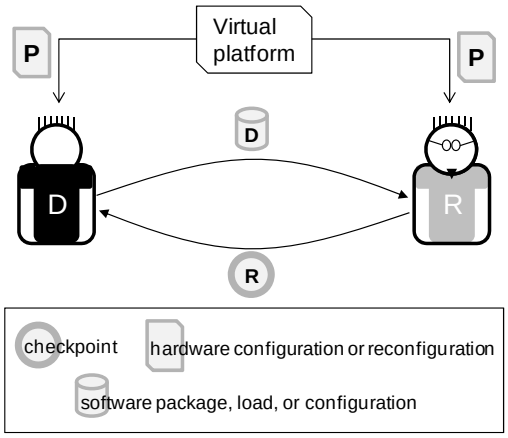

Figure 1 Basic Concept

Note that in all figures in this paper, the meaning of the "disk icon" and "board icon" encapsulates any kind of information added when working with software and hardware. For software, this would mean creating a binary, loading it onto a target, configuring it, and simply running it. For hardware, it would both mean the initial hardware setup of the virtual platform, as well as any changes and configurations done to the virtual hardware.

2. TECHNOLOGY BACKGROUND

2.1. Virtual Platforms

The basic requirement for the proposed approach is that there is a virtual platform (VP) available for the target system. Virtual platforms are currently becoming a natural

part of the development process for embedded systems, thanks to the flexibility they bring by decoupling hardware and software development. In many cases, VPs offer debug and development environments superior to physical hardware systems. Among the more interesting VP features are checkpointing and determinism, which form the basis for the work presented in this paper.

We have been using virtual platforms built using Wind River Simics (http://www.windriver.com/products/simics/). Simics has been used since the late 1990s to develop software for complex hardware-software systems [1]. Simics is capable of providing virtual platforms for anything from small single-processor embedded controllers up to complex multicore, multiboard, rack-based clusters [1], running target software quickly enough to be useful for large-scale software development [2].

For embedded systems, an important advantage of virtual platforms is that they are available anywhere, in any number, and any point in time. This makes it feasible for the reporter and developer to share the same hardware setup without shipping development hardware around the globe. Such common hardware ground certainly aids debugging.

To facilitate software development, there is no need for the virtual platform to be timing and pin-level accurate to the actual target – indeed, the only way to create a virtual platform which is fast enough to afford software development is to simplify the timing and use transaction-level modeling. Virtual platforms built in this style have been used for software development for at least 40 years [4].

It is important to note that a VP does not have to be a complete and perfect model of the hardware system to be useful for the task of testing and debugging software and transporting bugs. The VP needs to be complete enough to make the software happy, but that requirement is usually met by something short of a complete model. For example, using a virtual development board with a particular SoC plus memory and networking, running the same operating system as the actual target is often sufficient. Parts of the target system can be stubbed out or replaced with traffic generators [3].

2.2. Checkpointing

Checkpointing means storing the state of the virtual platform target to disk. The checkpoint can later be loaded into a fresh virtual platform session, resulting in the exact same target system state. Checkpoints include the contents of memories and disks, the state of processors, peripheral devices, and network connections in the virtual system. The checkpoint also needs to store some parts of the simulation kernel state, such as the current time and any event queued for later execution. Checkpointing has been implemented in various ways for a range of virtual platform systems, with the first successful implementations taking place in the mid-1990s [6][7][8][9][10].

For bug transportation, a crucial aspect of the checkpoint is that it contains the software state of the target system. When a checkpoint is taken, software can be loaded into memory, executing inside a processor, or residing on a disk. It does not matter, as all system state is captured in the checkpoint, affording the developer full insight into the software setup involved in the reporter's bug.

There is no need for the reporter to figure out the precise versions of the software they are using. When a developer has a checkpoint, it is possible to investigate any aspect of the software setup, without round-trips to the reporter. Examining the contents of the checkpoint can require the developer to run the virtual platform, potentially changing its state. However, by making checkpoints *read-only*, the checkpoint contents is preserved and the developer is able to go back to a pristine state any number of times [11].

For efficiency, checkpoints need to be *differential* so that each checkpoint only stores the changes to the system state since the previous checkpoint was taken. Without this optimization, checkpoints very quickly become unmanageably large. Another optimization is to only store the memory areas which are actually in use in a system. This means that even if a target system has many gigabytes of RAM, the checkpoint (and Simics memory usage on the host) might only be a few hundred megabytes if that is all the memory that is being used.

2.3. Checkpoint Portability

In order to actually *transport* bugs and not just reproduce them on the machine where they were generated (which is certainly useful), checkpoints have to be *portable* across hosts, across time, and across virtual platform software versions. In practice, the developer and the reporter can have very dissimilar systems, typically differing in the operating system version, size of memory, number of processor cores, and maybe even in word length and endianness. Simics checkpoints are designed to achieve such portability, but this is not necessarily the case for all checkpoint implementations [7][10]. See previous work for more details on the Simics checkpoint implementation [6].

One particular aspect of checkpoint portability and transportability is that they do not contain session state such as breakpoints or debug information. Neither do they contain the code for the hardware models used, that has to be provided separately to each user of a VP. However, the size of the code of a typical VP is far less than 100 MB, often less than 10 MB. Essentially, checkpoints only encode the target system state and nothing else [6][9].

2.4. Determinism

In a *deterministic* virtual platform system, the target system will execute in the exact same way each time a checkpoint is loaded. The checkpoint provides a fixed initial state for the simulation, and determinism guarantees that the execution is

the same each time the checkpoint is opened. Thus, the reporter and the developer will see the same execution (starting from the checkpoint), and reproduction of a bug becomes trivial. Without determinism, the value of VPs and checkpoints is greatly reduced.

3. PRACTICAL CHECKPOINTING USE

Working with checkpoints for the last decade has shown us that best practices can greatly enhance their effectiveness and efficiency.

The first issue is to bring down the size of the checkpoints which are transported. Even though networks are fast, moving gigabytes of data over long distances is a painful exercise. Bandwidth is an issue between different organizations and different geographical locations of the same company. Therefore, just blindly recording all the system state in a checkpoint might result in a blob of data that is so large that it is practically useless. Obviously, the data in a checkpoint should be compressed, but compression is in general not enough.

3.1. Shared Platform Software

Thankfully, the way products are developed helps us. Usually, there is some kind of basic platform software (often created by a platform team) which really does not need to be included in the checkpoint as it is already available to the reporter and developer.

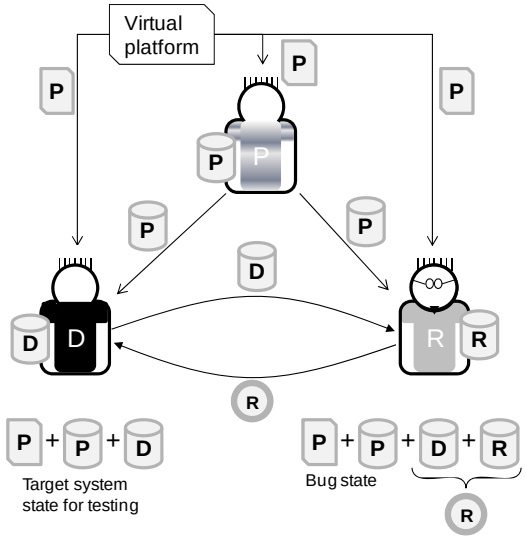

Figure 2 Workflow with a shared platform

The resulting workflow is shown in Figure 2, adding a lot of details compared to Figure 1. The platform team (with the "P" shirt) distributes the platform software set (the disk P) to the developer and the reporter. The reporter constructs the system to be tested by combining P with the software from the developer (disk D), and possibly some local configuration or input set (disk R). When a bug is found, only the difference from the platform software needs to be reported, and thus the checkpoint R which is sent to the developer only contains the changes made by R (including the loading of the developer's software onto the target).

In Simics, the sharing of common software is implemented by allowing checkpoints to depend on static read-only disk images. The current state of a disk or memory in the virtual platform is found by starting with an external file (the static disk image) and then adding the differences specified in the checkpoint. The user of the checkpoint tells Simics where to find the static images.

Note that Figure 2 also introduces the hardware configuration of the VP. In this case, all users use the same virtual hardware configuration (board P), but this is not always the case as discussed below.

3.2. Nightly Boot

In the previous scenario, the platform team provided software images to the other teams. This is a fairly inefficient way to work with a virtual platform, however. Each user has to spend their own time booting the VP, and runs the risk of not having quite the same configuration as other users. One way of solving this is to extend the concept of a "nightly build" to also encompass a "nightly boot". Essentially, the platform team provides a ready-to-use booted system checkpoint containing both the hardware and software configuration.

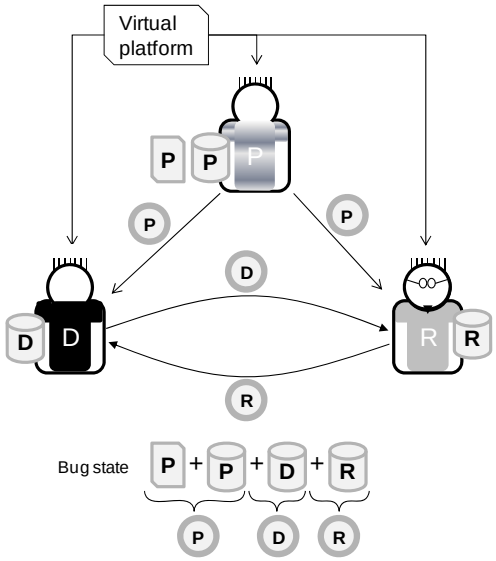

Figure 3 Nightly boot workflow

Figure 3 shows a nightly boot workflow. The target system is booted once by the platform team, and the resulting checkpoint P is distributed to the other teams. The platform team takes control of the hardware configuration to be used by the other teams. D and R receive the virtual platform hardware configuration as part of the checkpoint.

In practice, there can be many checkpoints P produced, each one for a certain hardware/software configuration. For example, in a rack-based system, there might be a set of

board configurations. Each configuration would target certain software development areas, such as control plane software, DSP software, or operations and management software, and only include the boards and software needed for each case. Configurations might also mix in boards from previous generations of hardware, in order to test backwards compatibility of new software.

The development team adds its software to the system configuration given by the checkpoint P, generating a new checkpoint D, which is then used by the reporter. The final checkpoint R builds on all the previous checkpoints – but it only contains the changes from checkpoint D, making it fairly compact. Most of the information would be contained in the checkpoint P.

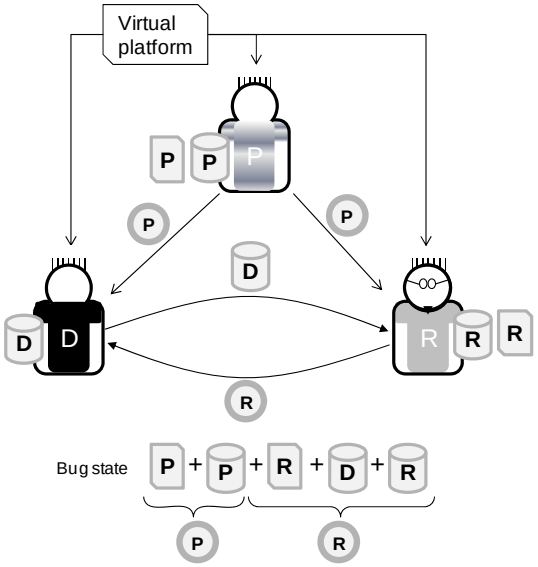

Figure 4 Nightly boot with reporter loading developer software

Another workflow variant is shown in Figure 4. Here, the reporter first modifies the hardware configuration of the virtual platform and then loads the developer's software in order to test it. This is a fairly common operation in testing. Bugs can be triggered by actions like reconfiguring the virtual platform network, injecting faults into the target system, or adding nodes to network. All such changes are captured in the checkpoint, making it much easier for the developer to reproduce and understand the bug.

Imagine the work needed to replicate such a test on physical hardware: you would go to the lab, find the appropriate pieces of hardware, and load the specified software. Then, after a precise amount of time, pull a network cable and insert it into a different port. Wait another precise amount of time. Power up a second board and connect it to the network. Type a particular sequence of commands on the new board, and on and on. Reproducing and describing such sequences of actions is not easy, but a checkpoint offers an easy way to capture their effects.

3.3. Checkpoint Chains and Checkpoint Merge

Differential checkpoints create chains of dependent checkpoints. For example, in Figure 3, in order to use checkpoint R, you also need to have the checkpoints D and P. This keeps each checkpoint small, but it also means that when a user creates several checkpoints as part of their work, you end up with a set of checkpoints that all need to be sent as part of the bug report. The situation is illustrated in Figure 5, along with the solution: checkpoint merging.

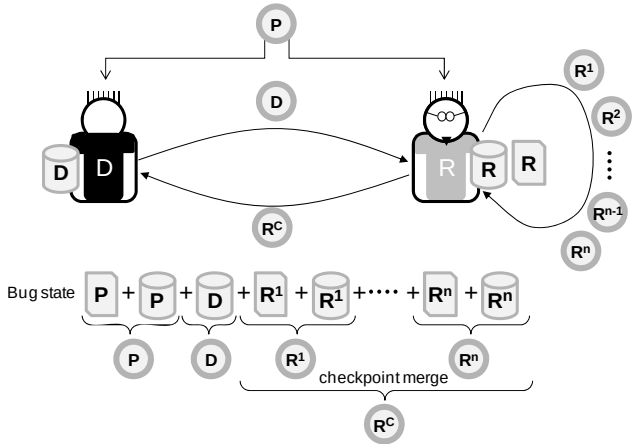

Figure 5 Checkpoint chains and checkpoint merge

In a checkpoint merge, a single checkpoint is created which summarizes all the changes in the checkpoint chain. Normally, the merged checkpoint is smaller than the set of checkpoints it replaces, as data items that have changed several times in the checkpoint chain are only represented by their last values.

In general, the new checkpoint still depends on shared checkpoints or static disk images (as is the case in Figure 5). It is possible to create a single "absolute" checkpoint which does not depend on any other checkpoint or disk image. This will be a large file, but there are cases where it is convenient, for example when passing a checkpoint to a group which does not have access to the right version of the platform software.

4. SYSTEM STIMULUS

The above discussion has ignored the issue of system input and how to reproduce the stimulus needed to cause a bug to occur. In the best case, the bug report checkpoint has captured the system state after the last relevant stimulus occurred. Simply running the virtual platform from the checkpoint will reproduce the bug. In general, however, the checkpoint has to be followed by some stimuli to the target in order to trigger the bug.

4.1. No Inputs

When the system stimulus is the hardware and software configuration itself, capturing the bug in a checkpoint is

trivial. Just start the system and let it run. The trick here is to capture the checkpoint as late as possible, in order to reduce the waiting time for the developer. To get to a good point in time, the reporter typically reruns the failing test case, stopping some time short of the time that the bug occurs.

One example of a no-input configuration is using a test program with compiled-in input data. Another example is changing the hardware and software configuration of the target and booting it (this works surprisingly often). On the software side, this might be updating an OS kernel, adding a new device driver, or changing target init scripts. On the hardware side, changing parameters like clock frequencies, core counts, or the types of boards used can reveal software bugs. It is a key strength of virtual platforms that they are able to vary their configuration to explore a larger space of possibilities than that offered by physical hardware. In particular, multicore software tends to be sensitive to hardware and software configurations [12].

4.2. Internal Inputs

The target system can be driven to a bug state by a stimulus sequence generated *within the virtual platform itself*. For example, traffic generators, fault injection frameworks, or scripted input can be used to test software [11][14]. Stimuli sources running inside the virtual platform system have the distinct advantage that they are perfectly reproducible: they work in the virtual time of the virtual platform, and each time they are run, they will produce the same input sequence with the exact same timing. When stimulus generators use pseudo-random number generators to generate variable data, they need to make sure that the state of the random generator is part of the checkpoint (and that the generators are considered part of the VP for checkpointing purposes).

Just like for the case of no input, only the checkpoint is needed to reproduce the bug.

4.3. External Inputs

Inputs can come from tools *external to the virtual platform*, including hardware network packet generators and software running on the same host machine but outside the virtual platform [14][15]. In general, we cannot assume that external tools will be able to regenerate a particular input sequence deterministically and with precise timing, since they depend on the nature and timing of the host machine.

The solution is to record the inputs from the external tools, and use these recordings to drive a deterministic replay of the stimulus sequence following the checkpoint. Recordings capture the contents of the inputs and the exact point in virtual time when they appear to the virtual platform.

Figure 6 shows an example of using a recording. We also use periodic checkpointing during the execution of a test. Taking checkpoints at a regular interval means that there is always a "fairly recent" checkpoint that we can go back to in order to report a bug. Regular checkpointing is particularly important for long test runs, where the virtual platform might run overnight or for several days. Obviously, you do not want to restart such a run in order to prepare a bug report.

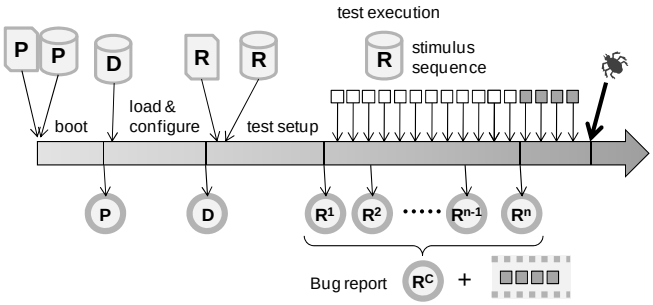

Figure 6 Periodic checkpointing and input recording

4.4. Spontaneous Input

A fairly special source of asynchronous input is the user of the virtual platform, typing in the serial consoles of the virtual target as well as giving commands to the virtual platform tool itself. More often than not, if a bug is found during interactive usage, the user has to reproduce the issue in a way similar to how you would on a physical machine. A virtual platform has the great advantage that scripting can be used to precisely build a replay of the input sequence.

Usually, the reporter will start the recreation with a checkpoint of the system state (since almost all users save a checkpoint at least after having booted the target), and create a script issuing target commands and waiting for target outputs. Building a script is often preferable to just recording inputs, since the resulting sequence of commands can be changed to try variants, and used with updated software for regression testing. Recorded input is tied to the particular system configuration where the recording was taken.

The bug report would consist of a checkpoint and a script that typically opens the checkpoint and then performs the necessary steps to trigger a bug [11].

Scripts can also be used to decorate and explain bugs. An ambitious reporter can create scripts that set breakpoints to catch when the target software reaches certain locations, wait for certain points in time, or react to target output, and annotate these events with additional information for the developer. The bug report thus becomes an executable interactive session, rather than a piece of static text.

5. RELATED WORK

Hardware trace debugging such as that offered by GreenHills Time Machine and the Lauterbach debuggers use special hardware trace units to capture a tape-like recording of the behavior of a target processor. However, they only look at the processor (no recording is made of peripheral devices), have a limited recording time, and offer

no way to save the absolute state of the target for someone else to process.

The *reverse debugging* of gdb 7.x and UndoDB also builds on a record-replay system. Such reverse debugging does not offer the ability to transport recordings between machines, and only apply (by design) to user-level processes. They are also only available for certain operating systems.

Regular *desktop/server virtual machine* systems like VmWare and VirtualBox offer the ability to snapshot the state of a virtual machine and move it to another host. VmWare offers deterministic replay of the target system, but only for a single-processor targets [16]. Thus, they provide only part of the solution presented in this paper. The *ReVirt* system offers checkpointing and replay of the execution of multiprocessor systems [17]. It assumes the use of paravirtual guests rather than unmodified software stacks, and the checkpoints and recordings do not appear to be designed for portability. ReVirt does not deal with multiple machines in networks or hardware changes over the course of an execution.

Hardware data recorders record the inputs to an embedded system, such as network packets and other data streams. They can be deployed in the field or used as part of product testing. The recordings are replayed in a development lab in order to reproduce issues found. The difference from our approach is that it only provides the inputs and not the target system state, nor does it provide for determinism in the (re)execution of the system. Note that the data streams from such recorders can be used with a virtual platform as a stimulus.

Simics's *reverse execution* technology allows a user to apply reverse debugging to an entire VP including the target OS. It is a very powerful debugging tool, and a complement to the use of checkpointing to transport bugs. In particular, reverse execution helps position the target in a good spot for a bug-report checkpoint to be created.

6. SUMMARY

This paper has presented our experience in using irtual platforms and checkpoints to transport bugs from a bug reporter to a developer. With checkpoints, both the state of the system where the bug was found and the steps needed to reproduce the bug can be encapsulated for transportation. In addition to the checkpoint, there are cases where you also need to move recorded inputs to the target system to complete a bug report. Such record and replay can also be automated using a virtual platform.

Organizations should take advantage of existing data shared between bug reporters and developers in order to minimize the need to move large amounts of data around. If an organization has a defined platform group, it can provide shared setups involving both hardware and software to both developers and bug reporters.

7. REFERENCES

[1] Peter S. Magnusson, Magnus Christensson, Jesper Eskilson, Daniel Forsgren, Gustav Hallberg, Johan Hogberg, Fredrik Larsson, Andreas Moestedt, Bengt Werner. "Simics: A Full System Simulation Platform", *IEEE Computer*, February 2002.

[2] Mikael Bergqvist, Jakob Engblom, Mikael Patel, and Lars Lundegard, "Some Experience from the Development of a Simulator for a Telecom Cluster (CPPemu)", *IASTED Conference on Software Engineering and Applications*, November, 2006.

[3] Jakob Engblom: "Simulating Embedded Hardware for Software Development", *Embedded Systems Conference Silicon Valley*, San Jose, USA, 17 April 2008.

[4] Kazuhiro Fuchi, Hozumi Tanaka, Yuriko Manago and Toshitsugu Yuba. "A program simulator by partial interpretation", *Symposium on Operating systems principles (SOSP)*, Princeton, New Jersey, October 1969.

[5] Stanley 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.

[6] Marius Monton, Jakob Engblom, and Mark Burton, "Checkpoint and Restore for SystemC Models", *Forum on Specification and Design Languages (FDL)*, Sophia Antipolis, France, 22-24 September 2009.

[7] Stefan Kraemer, Rainer Leupers, Dietmar Petras, and Thomas Philipp, "A Checkpoint/Restore Framework for SystemC-Based Virtual Platforms", *International Symposium on System-on-Chip (SoC)*, Tampere, Finland, 5-7 October 2009.

[8] Mendel Rosenblum and Mani Varadarajan, *SimOS: A Fast Operating System Simulation Environment*, Stanford University Technical Report CSL-TR-94-631, 1994.

[9] J. L. Peterson et al: "Application of full-system simulation in exploratory system design and development", *IBM Journal of Research and Development*, Vol 50, no 2/3, March/May 2006.

[10] Cadence. SystemC Save and Restore Part 2 Advanced Usage. http://www.cadence.com/Community/blogs/sd/archive/2009/03/09/systemc-save-and-restore-part2-advanced-usage.aspx

[11] "Finding an Intermittent Multicore Bug with Wind River Simics", Wind River whitepaper, May 2010.

[12] Jakob Engblom, "Debugging Real-Time Multiprocessor Systems", *Embedded Systems Conference Silicon Valley*, San Jose, USA, 3 April 2007.

[13] Jakob Engblom, Bengt Werner, and Guillaume Girard, "Testing Embedded Software using Simulated Hardware", *Embedded Real-Time Software (ERTS)*, Toulouse, France, January 2006.

[14] Ross Dickson, Jason Andrews, and Jakob Engblom, "Design Flow for Embedded System Device Driver Development and Verification", *Design Automation Conference (DAC)*, San Francisco, USA, 29 July 2009.

[15] E. Lewis, "Preparing to Replay Debug…", http://www.replaydebugging.com/2009/11/preparing-to-replay-debug.html, November 2009.

[16] George Dunlap, Dominic Lucchetti, Michael Fetterman, and Peter Chen, "Execution Replay of Multiprocessor Virtual Machines", *ACM/Usenix International Conference On Virtual Execution Environments (VEE)*, Seattle, USA, March 2008.