# SIMULATION-BASED TEMPORAL DEBUGGING OF LINUX

**Lars Albertsson, Peter S Magnusson**
Computer and Network Architectures Laboratory
Swedish Institute of Computer Science
Box 1263, SE-164 29 Kista, Sweden
lalle@sics.se, psm@virtutech.se

**Abstract**

We present a temporal debugger, capable of examining temporal behaviour of operating systems. The debugger is based on a simulator modelling an entire workstation at the instruction level. Unlike traditional debuggers, which need to interfere with program execution, a simulation-based debugger can operate without disturbing time flow of the simulated system. This allows non-intrusive and reproducible debugging of general-purpose operating systems, such as Linux.

We demonstrate the utility of the temporal debugger by analysing two time sensitive parts of Linux, scheduling and interrupt handling. We show how our tool allows a user to identify and isolate temporally unsatisfactory behaviour, and examine short sequences in detail.

**Keywords:** Soft Real Time Systems, Temporal Debugging, Operating Systems, Linux, Simics, Complete System Simulation

## 1 Introduction

Many desktop applications, such as video, audio, games and control software, need soft real-time guarantees. Commodity desktop and server systems, such as Linux, were not designed for this purpose, and implementing support for real-time services in operating systems is hard. The difficulty of the task is compounded by the lack of adequate tools for real-time system analysis.

The debugger is one of the primary tools for finding errors in computer programs. The correctness of real-time programs includes not only logical correctness, but also time elapsed during execution. This is referred to as temporal correctness. Debuggers often interfere with program execution in various ways, and cannot be used to examine time flow. Conventional debuggers are therefore inadequate for validation of real-time applications and operating systems. In order to avoid time distortion while debugging, debuggers based on simulation may be used. For embedded real-time systems, which are much slower than workstations, it is possible to construct temporally accurate simulators using straightforward implementation techniques. These techniques are infeasible for simulating desktop systems, as simulation would be too slow. However, advances in simulation technology have resulted in simulators providing an approximate, but reasonably accurate timing model while executing with a slowdown of roughly 50-200 in relation to native execution [7, 10]. These simulators, referred to as complete system simulators, model an entire workstation at the instruction set level, and runs unmodified operating systems and workloads. A complete system simulator that is deterministic addresses the two major problems in real-time analysis: lack of reproducibility and time distortion resulting from intrusion. The characteristics of complete system simulators make them excellent candidates for building a temporally correct debugger for real-time operating systems.

In this paper, we present a temporal debugger based on complete system simulation. We also demonstrate its functionality by debugging two time critical sequences in Linux: scheduler invocation and interrupt handling.

Section 2 contains the problem definition and motivation for the use of simulation. It also includes a summary of related work and alternate approaches.

In Section 3, complete system simulation is briefly described. Section 4 presents an example of debugging Linux using simulation. Conclusions are presented in Section 5.

## 2 Debugging real-time programs

A debugger allows a programmer to inspect program state. In order for the debugger to be useful, it must not affect correctness by changing program behaviour. Also, as debugging is a repetitive task, the programmer needs to be able to repeat sessions, and observe identical execution each time. For programs whose correctness depend only on predictable input, meeting these requirements is straightforward. However, a debugger for real-time programs must be able to capture and replay program time flow without changing it. In this section we describe how a debugger based on complete system simulation (described in Section 3) addresses these problems and allows debugging of real-time programs.

### 2.1 Temporal debugging using a simulator back-end

A complete debugger setup consists of the debugger program and a target machine running the debugged program. Figure 1 shows the different parts of a debugger setup. We refer to the debugger program itself as front-end and to the target machine/program tuple as back-end. Examples of such tuples are: Unix programs running in the virtual machine provided by the operating system, or an embedded operating system on a separate target board. In our case the tuple consists of an operating system running in a simulated machine.

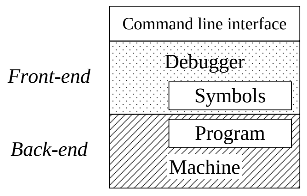


**FIGURE 1:** *Debugger structure.*

The debugger expects a certain set of primitives needed to probe and control the target machine. Such primitives include reading memory, reading registers, single stepping and setting breakpoints. Adapting a simulator to the primitives required by a specific debugger enables symbolic debugging of the simulator workload. Unlike traditional debugging environments, such a setup does not impact on execution of the debuggee. Traditional debuggers need to stop the debugged program in order to probe its state, thereby affecting time as perceived by the target program.

In addition to primitives required by a debugger front-end, the simulator provides services not normally available in a debugger. Debugger front-ends may allow transparent access to the back-end interface, thereby exporting services not known to the debugger.

The service most relevant for real-time analysis is the ability to present current time with cycle count granularity. It enables the user to step through a portion of code, checking for both functional and temporal errors.

### 2.2 Reproducibility

An artificial system has few unpredictable factors. Thus, a simulated system starting execution in a known state will always execute along the same path. This is useful, both for experiments and debugging, as it is possible to reproduce a state reached in execution. A user of a temporal debugger may detect that excessive time has passed at one point in execution, and restart the simulation to examine recently executed routines more carefully. This is similar to the methodology used for debugging logical correctness of conventional programs. However, as time is part of the state the user wishes to verify, it is crucial that temporal behaviour is preserved between debugging sessions.

### 2.3 Probe immunity

In physical systems, measurement of the system generally affects its behaviour. This is referred to as the *probe effect*. Because real-time system analysis tends to focus on short periods of time, even small amounts of temporal intrusion affect measurement quality. This limits both the accuracy and amount of measurement in such systems.

In a simulated system, the time scale of the system under study is decoupled from the time scale of the system running the simulator. When the user stops execution, simulation is suspended, and simulated time is frozen. Time distortion due to the probe effect is thereby eliminated.

### 2.4 Related work

In many existing real-time operating systems and environments, only conventional, non-real-time debugging tools are available. These systems may only be used for validating and debugging functional, not temporal, correctness. However, there are vendors providing support for alternative debugging methods. Some of these methods are discussed below.

When developing programs for small embedded systems, it is common to use an emulator (tool for executing programs in foreign environments) as debugging back-end. Emulators generally focus on the functional model and do not provide a time model, which is necessary to avoid intrusion and to reproduce sessions. Some processor manufacturers provide simulators with models of cache and pipeline, resulting in good execution time modelling. Prior to recent advances in complete system simulation, such simulators were not useful for running commodity operating systems and large applications. The tools available are either too incomplete to run general-purpose operating systems or too slow to run desktop applications [2, 5].

Support for non-interactive debugging of real-time programs may be provided by logging execution to a trace, which is sent over a network to a separate system. The trace may be generated by dedicated hardware [6, 13, 15] or additional program code [3, 12]. Both approaches are inconvenient, and the amount of monitoring is limited. Furthermore, it is inflexible, as the receiving system may not query for additional data.

The R2D2 debugger [14] is based on monitoring of software generated traces. It has been extended with a low priority task in the target system to answer queries from the debugger in case the system is idle. This provides some support for interactive debugging. However, sessions cannot be authentically repeated, and the target system may be unable to provide information when it is under stress.

Mueller and Whalley [11] propose debugging of real-time applications using execution time prediction. The application is executed in a conventional debugger, supported by a cache simulator. Time elapsed is predicted by the simulator and reported during debugging. However, this prediction does not take operating system effects into account and works best for small programs.

The work presented here is made possible by advances in simulation of computer systems, allowing the construction of accurate simulators of complete computer systems [4, 7, 10]. The results presented in this paper have previously been presented in [1], which also contains further discussion on related work and issues important to debugging real-time programs.

## 3 Complete system simulation

Many design and research areas benefit from simulation of computer systems. Thus, the level of detail provided by simulators range from models of microprocessor chip logic to coarse models of an execution environment including operating system and libraries. The simulator used in this paper is a complete system simulator. It provides a model of a machine at the instruction set level, which represents a well-defined border between hardware and software. It models all the hardware in a system, and only the hardware. As the simulation model is functionally identical to a real system, operating system and application software need not be modified. This limits the sources of errors to those introduced by the hardware model, and by models of simulation input feed.

### 3.1 Simulation model

The model provided by a simulator can be thought of as having two components, functional and temporal. The simulator must provide an almost exact functional model to be able to run unmodified software. However, the accuracy of the temporal model can be compromised without breaking the operating system and applications. It is therefore possible to trade accuracy for speed by using approximative models. The appropriate degree of approximation depends on the size of workload and the time scale of its deadlines. Note that temporal accuracy is less important than reproducibility and absence of probe effect (discussed in section 2). A reasonably accurate time model is usually sufficient to obtain a coarse understanding of the timing behaviour of the system.
When using a simulation based methodology, we assume that we are able to identify and model the major sources of delay in a system. For large applications, this primarily involves delays from devices, such as disk subsystems, and the memory hierarchy, including the memory management unit, caches, and memory buses. If a user wishes to increase accuracy, the timing model may be changed at need to include other delays, for example from CPU pipeline stalls or out-of-order execution. However, this has a significant impact on simulation performance, and is often not necessary.

### 3.2 Simics

The simulator used for this work is Virtutech Simics [16]. Simics simulates the SPARC V9 instruction set and models single or multiprocessor systems corresponding to the sun4u architecture from Sun Microsystems, for example the Enterprise 3500.

Simics consists of a core interpreter that offers basic services such as an instruction set interpreter, a general event model, and a module for simulating and profiling memory activity. A programming interface allows the addition of device models, which may be connected to the "real world" or models thereof.

Simics supports a simple time model in its default configuration. This model approximates time by defining a cycle as either an executed instruction, a taken trap, or a part of a memory or device stall. In this mode, Simics thus has a rather simple view of the timing of a modern system, and assumes a linear penalty for events such as TLB miss, data cache miss, and instruction cache miss.

## 4 Temporal debugging of Linux

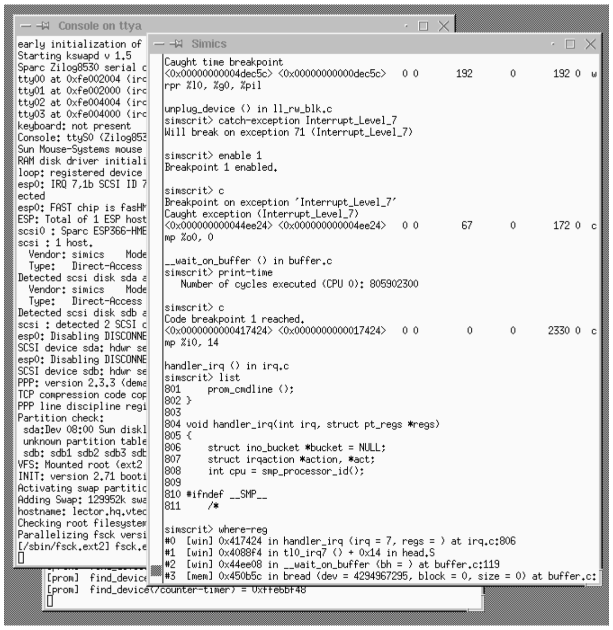
As a demonstration of debugging real-time aspects of operating systems, we describe results from an example debugging session. In our scenario, a user wishes to examine scheduling in the Linux kernel. The purpose may be to implement a new scheduling algorithm or to design a resource reservation scheme, or simply to learn about operating system internals. In any case, being able to carefully step through time sensitive sequences is valuable.

### 4.1 Experimental setup

In the experiment, Simics is used to simulate an UltraSPARC workstation. The simulator reads a file representing a disk image and boots the operating system contained therein. The image used contains an installation of UltraPenguin 1.1, including a Linux kernel version 2.1.126. We use a very simple timing model, where each instruction takes one cycle and memory system latencies are ignored.

Figure 2 shows a screenshot from a debugging session, using Simics' internal debugger module as front-end. The console in the background shows the output of UltraSPARC Linux during boot.

In order to examine Linux scheduling properties, a small synthetic benchmark is executed in the simulated machine. The benchmark is a CPU bound application with a real-time requirement. It needs nearly all of the CPU each period in order to meet its deadline. Listing 1 shows pseudo-code for the benchmark.



```
early initialization of
Starting kswapd v 1.5
Sparc Zilog8530 serial c
tty00 at 0xfe002004 (irc
tty01 at 0xfe002000 (irc
tty02 at 0xfe004004 (irc
tty03 at 0xfe004000 (irc
keyboard: not present
Console: ttyS0 (Zilog853
Sun Mouse-Systems mouse
RAM disk driver initiali
loop: registered device
esp0: IRQ 7,1b SCSI ID 7
ected
esp0: FAST chip is fasHM
ESP: Total of 1 ESP host
scsi0 : Sparc ESP366-HME
scsi : 1 host.
  Vendor: simics    Mode
  Type:   Direct-Access
Detected scsi disk sda a
  Vendor: simics    Mode
  Type:   Direct-Access
Detected scsi disk sdb a
scsi : detected 2 SCSI c
esp0: Disabling DISCONNE
SCSI device sda: hdwr se
esp0: Disabling DISCONNE
SCSI device sdb: hdwr se
PPP: version 2.3.3 (dema
TCP compression code cop
PPP line discipline regi
Partition check:
 sda:Dev 08:00 Sun diskl
 unknown partition table
 sdb: sdb1 sdb2 sdb3 sdb
VFS: Mounted root (ext2
INIT: version 2.71 booti
Activating swap partitic
Adding Swap: 129952k swa
hostname: lector.hq.vtec
Checking root filesystem
Parallelizing fsck versi
[/sbin/fsck.ext2] fsck.e
▯
    [prom]  find_device
    [prom]  find_device(/counter-timer) = 0xffe6bf48
    ▯
```

```
Caught time breakpoint
<0x000000000004dec5c> <0x00000000000dec5c>   0 0      192       0       192 0  w
rpr %l0, %g0, %pil

unplug_device () in ll_rw_blk.c
simscrit> catch-exception Interrupt_Level_7
Will break on exception 71 (Interrupt_Level_7)

simscrit> enable 1
Breakpoint 1 enabled.

simscrit> c
Breakpoint on exception 'Interrupt_Level_7'
Caught exception (Interrupt_Level_7)
<0x00000000044ee24> <0x000000000004ee24>   0 0       67       0       172 0  c
mp %o0, 0

__wait_on_buffer () in buffer.c
simscrit> print-time
   Number of cycles executed (CPU 0): 805902300

simscrit> c
Code breakpoint 1 reached.
<0x000000000417424> <0x000000000017424>   0 0        0       0      2330 0  c
mp %i0, 14

handler_irq () in irq.c
simscrit> list
801     prom_cmdline ();
802 }
803
804 void handler_irq(int irq, struct pt_regs *regs)
805 {
806     struct ino_bucket *bucket = NULL;
807     struct irqaction *action, *act;
808     int cpu = smp_processor_id();
809
810 #ifndef __SMP__
811     /*

simscrit> where-reg
#0  [win] 0x417424 in handler_irq (irq = 7, regs = ) at irq.c:806
#1  [win] 0x4088f4 in tl0_irq7 () + 0x14 in head.S
#2  [win] 0x44ee08 in __wait_on_buffer (bh = ) at buffer.c:119
#3  [mem] 0x450b5c in bread (dev = 4294967295, block = 0, size = 0) at buffer.c:
```

**FIGURE 2:** *Example of Simics debugging interface.*

The benchmark competes with a `find` and a `grep` process searching through all the files in the local file system. These tasks are supposed to be considered unimportant. Thus, their process priorities have been set as low as possible. Despite competing processes, the real-time benchmark manages to meet most of its deadlines. However, a few are missed, and we will examine one of them.

**Listing 1:** *A soft real-time benchmark*

```
install_signal_handler();
while (true) {
  deadline_missed = false;
  set_timer();
  for (i = 0; i < num_iterations; i++)
    dummy_operation();
  clear_timer();
  if (deadline_missed)
    printf("Missed deadline\n");
}
```

### 4.2 CPU scheduling in Linux

Linux, being a general-purpose operating system, is not designed to meet strict quality of service requirements. As a consequence, the CPU scheduler is optimised for throughput and responsiveness, rather than guaranteed resource sharing. Scheduling decisions are made when a process changes to or from running state, and during timer interrupts. Scheduling granularity is limited by the distance between timer interrupts, which is 10 ms on the configuration under study. As our benchmark is very sensitive to scheduling variations, we suspect that low timer interrupt resolution is the cause of deadline misses. In the following section, we debug a period where the benchmark misses its deadline to verify our suspicion. We also look at another cause of scheduling jitter in operating systems, namely interrupt service latency. We will use our example setup to produce a time stamped call graph of an interrupt handler invocation.

## 4.3 Debugging scheduling decisions

Temporal debugging of the benchmark is similar to traditional debugging. We execute our program, searching for an error. When an error is found, we restart the session in order to examine execution before the error occurred.

A missed deadline has been detected by watching benchmark output on the simulated console. Once the failing period is located in time, we wish to get an overview of time flow during that period. This is obtained by setting breakpoints on strategic kernel routines. When a breakpoint is triggered, time, position and currently active process are printed. In the example shown below, breakpoints are set in the scheduler, the timer set routine and the timer overflow routine. From the output, presented in Table 1, we can see that `find` is scheduled during the period. It executes only briefly, but for long enough to make the benchmark miss its deadline.

| Time since boot | Time since previous event | Event | Process currently running |
|---|---|---|---|
| 1348040300 | | set timer | benchmark |
| 1360269869 | 12229569 | scheduler | benchmark |
| 1360980692 | 710823 | scheduler | find |
| 1373709842 | 12729150 | timer overflow routine | benchmark |

**TABLE 1:** *Scheduling during one period. Time unit is CPU cycles.*

We proceed by inserting more breakpoints to increase level of detail. The simulator has now been told to break on traps and some exceptions, such as interrupts and MMU exceptions. We also insert breakpoints at the points where interrupt and trap handlers return. Detailed time flow for a fraction of the period is shown in Table 2. It reveals that a timer interrupt occurs every 1680000 cycles. Therefore, the scheduler cannot run more often unless the competing process performs a blocking system call. As our benchmark is sensitive to even smaller variations than this, it confirms our guess that timer interrupt granularity limits scheduling resolution.

| Time since boot | Time since previous event | Event | Process currently running |
|---|---|---|---|
| 1358589276 | 1679357 | timer interrupt | benchmark |
| 1358589919 | 643 | return to user space | kernel |
| 1360269276 | 1679357 | timer interrupt | benchmark |
| 1360269869 | 593 | scheduler | kernel |
| 1360284738 | 14869 | return to user space | kernel |
| 1360285101 | 363 | trap number 16 | find |
| 1360285168 | 67 | system call | kernel |
| 1360285444 | 276 | return to user space | kernel |
| Further system calls in find | | | |
| 1360293863 | 888 | trap number 16 | find |
| 1360293951 | 88 | system call | kernel |
| 1360294823 | 872 | return to user space | kernel |
| 1360294979 | 156 | TLB miss | find |
| 1360295292 | 313 | return to user space | kernel |
| 1360295293 | 1 | TLB miss | find |
| 1360295303 | 10 | protection exception | find |
| 1360299254 | 3951 | return to user space | kernel |
| Further system calls and 2 more TLB misses in find | | | |
| 1360973991 | 370 | system call | find |
| 1360980213 | 6222 | interrupt vector exception | kernel |
| 1360980236 | 23 | SCSI interrupt | kernel |
| 1360980685 | 449 | return to kernel | kernel |
| 1360980692 | 7 | scheduler | kernel |
| 1360981101 | 409 | return to user space | kernel |
| 1360981570 | 469 | interrupt vector exception | benchmark |
| 1360981593 | 23 | SCSI interrupt | benchmark |
| 1360983380 | 1787 | return to user space | kernel |
| 1361949276 | 965896 | timer interrupt | benchmark |
| 1361949919 | 643 | return to user space | kernel |

**TABLE 2:** *Excerpt from time flow.*

## 4.4 Debugging interrupt handling

Even though we found the cause of the problem, we are not able to satisfy the needs of our benchmark without changing timer resolution and scheduling algorithm. Scheduling resolution may be improved by allowing timer interrupts at variable intervals [9]. However, experience with such modifications to Linux have shown that long interrupt service latency becomes a major source of scheduling jitter [8]. Therefore, we will proceed by inspecting interrupt service, trying explain variations in interrupt service time.

In the example used above, there is some interrupt activity during benchmark execution. From Table 2, we notice that the second SCSI interrupt takes 1787 cycles to service, whereas the previous interrupt was serviced in only 449 cycles. In order to find the difference in service time, simulation is restarted and time breakpoints are set to stop simulation at these two events. By single stepping through both interrupt handlers, we find that the first interrupt only processes an acknowledgement. However, the second interrupt terminates a SCSI transaction, which triggers execution of the kernel I/O subsystem. A time stamped call graph for the second interrupt is shown in Table 3. In order to minimise interrupt service time, this work could be postponed. However, in Linux, it is performed within the interrupt handler so that the operating system can schedule the process which is waiting for this data. Postponing the work would affect I/O performance and responsiveness of interactive applications.

| Time | Function | Call duration | Note |
|---:|---|---|---|
| 0 | interrupt dispatch | | |
| 92 |  handler_irq | | |
| 138 |   esp_intr | | |
| 317 |    esp_do_phase_determine | | |
| 350 |     esp_do_status | | |
| 522 |      esp_done | | |
| 535 |       mmu_release_scsi_sgl | | spill |
| 607 |       return from mmu_release_scsi_sgl | 72 | |
| 611 |       scsi_old_done | | |
| 619 |        update_timeout | | spill |
| 650 |         scsi_delete_timer | | spill |
| 676 |          del_timer | | spill |
| 710 |          return from del_timer | 34 | |
| 715 |         return from scsi_delete_timer | 65 | |
| 717 |        return from update_timeout | 98 | |
| 784 |        __wake_up | | |
| 790 |        return from __wake_up | 6 | |
| 802 |        rw_intr | | |
| 817 |         sd_devname | | |
| 826 |          sprintf | | |
| 843 |           vsprintf | | spill |
| 994 |           return from vsprintf | 151 | |
| 996 |          return from sprintf | 153 | |
| 998 |         return from sd_devname | 181 | |
| 1029 |         scsi_free | | |
| 1083 |         return from scsi_free | 54 | |
| 1091 |         end_scsi_request | | |
| 1118 |          end_buffer_io_sync | | |
| 1122 |           mark_buffer_uptodate | | |
| 1148 |           return from mark_buffer_uptodate | 26 | |
| 1160 |           __wake_up | | |
| 1205 |           return from __wake_up | 45 | |
| 1207 |          return from end_buffer_io_sync | 89 | |
| 1221 |          add_blkdev_randomness | | |
| 1235 |           add_timer_randomness | | |
| 1411 |            __wake_up | | spill |
| 1438 |            return from __wake_up | 27 | |
| 1438 |           return from add_timer_randomness | 203 | |
| 1440 |          return from add_blkdev_randomness | 219 | |
| 1450 |          __wake_up | | |
| 1458 |          return from __wake_up | 8 | |
| 1462 |          __wake_up | | |
| 1470 |          return from __wake_up | 8 | |
| 1472 |          scsi_release_command | | |
| 1497 |          return from scsi_release_command | 25 | |
| 1500 |         return from end_scsi_request | 409 | |
| 1505 |         requeue_sd_request | | |
| 1510 |          do_sd_request | | |
| 1524 |          return from do_sd_request | 14 | |
| 1527 |         return from requeue_sd_request | 22 | |
| 1529 |        return from rw_intr | 727 | |
| 1531 |       return from scsi_old_done | 920 | |
| 1535 |      return from esp_done | 1013 | fill |
| 1558 |     return from esp_do_status | 1208 | fill |
| 1581 |    return from esp_do_phase_determine | 1264 | fill |
| 1618 |   return from esp_intr | 1480 | fill |
| 1669 |  return from handler_irq | 1577 | fill |
| 1787 | return to user space | 1787 | fill |

**TABLE 3:** *Temporal call graph of SCSI interrupt service. Indentation level indicates call depth.*

During interrupt service, function call depth surpasses register window capacity, triggering traps to software handlers. The rightmost column of Table 3 shows whether the function call caused a register window spill trap or the return caused a register window fill trap. Register window trap handling only takes a small part of interrupt service time. However, the fact that the traps are noticed illustrates an advantage of using a simulator. As an operating system is an asynchronously event-driven program, it is difficult to predict its execution flow. A user proficient in operating system internals and computer architecture may guess how to probe a real computer system for appropriate and hopefully accurate information. However, most users benefit significantly from a detailed view of program flow.

The user can proceed using this methodology, searching for scheduling jitter in the system. When a temporal hazard is found, he may zoom in on a time window, down to the instruction level if necessary. As a complete system simulator provides deterministic execution, this debugging method is robust and sessions can be authentically repeated.

## 5 Conclusions

We have presented a simulation-based debugger, capable of temporal debugging of operating systems. The debugger consists of a command line user interface connected to a simulator modelling a complete computer system. The simulator executes the operating system in a protected environment and provides an approximate time model. It is deterministic and can be suspended without disturbing time flow of the simulated system. These properties makes the simulator suitable for debugging temporal behaviour of general-purpose operating systems, such as Linux. We have demonstrated our tool by analysing scheduling and interrupt handling in Linux. We have shown how the temporal debugger allows a user to step carefully through time sensitive sequences, searching for temporal errors, and reexamine execution whenever unsatisfactory behaviour is noticed.

## Acknowledgements

This work is funded by the national Swedish Real-Time Systems research initiative ARTES (www.artes.uu.se), supported by the Swedish Foundation for Strategic Research.

## References

[1] Lars Albertsson and Peter S. Magnusson. Using complete system simulation for temporal debugging of general purpose operating systems and workloads. In *Proceedings of MASCOTS 2000*. IEEE Computer Society, August 2000.

[2] William Anderson. An overview of Motorola's PowerPC simulator family. *Communications of the ACM*, 37(6):64–69, June 1994.

[3] Monica Brockmeyer, Farnam Jahanian, Constance Heitmeyer, and Bruce Labaw. An approach to monitoring and assertion-checking of real time specifications in Modechart. In *Proceedings of the Second IEEE Real-Time Technology and Applications Symposium*, Boston, USA, June 1996. IEEE Computer Society.

[4] J. K. Doyle and K. I. Mandelberg. A portable PDP-11 simulator. *Software Practice and Experience*, 14(11):1047–1059, November 1984.

[5] Embedded support tools corporation. www.estc.com.

[6] F. Gielen and M. Timmerman. The design of DARTS: A dynamic debugger for multiprocessor real-time applications. In *Proceedings of 1991 IEEE Conference on Real-Time Computer Applications in Nuclear, Particle and Plasma Physics*, pages 153–161, Julich, Germany, June 1991.

[7] Stephen Alan Herrod. *Using Complete Machine Simulation to Understand Computer System Behavior.* PhD thesis, Stanford University, February 1998.

[8] Robert Hill. Improving Linux real-time support: Scheduling, I/O subsystem, and network quality of service integration. Master's thesis, University of Kansas, Lawrence, Kansas, June 1998.

[9] Robert Hill, Balaji Srinivasan, Shyam Pather, and Douglas Niehaus. Temporal resolution and real-time extensions to Linux. Technical report, Department of Electrical Engineering and Computer Science, University of Kansas, June 1998.

[10] Peter S. Magnusson, Fredrik Dahlgren, Håkan Grahn, Magnus Karlsson, Fredrik Larsson, Fredrik Lundholm, Andreas Moestedt, Jim Nilsson, Per Stenström, and Bengt Werner. SimICS/sun4m: A Virtual Workstation. In *Proceedings of the 1998 USENIX Annual Technical Conference*, 1998.

[11] Frank Mueller and David B. Whalley. On debugging real-time applications. In *ACM SIGPLAN Workshop on Language, Compiler, and Tool Support for Real-Time Systems*, June 1994.

[12] Edgar Nett, Martin Gergeleit, and Michael Mock. An adaptive approach to object-oriented real-time computing. In Kristine Kelly, editor, *Proceedings of First International Symposium on Object-Oriented Real-Time Distributed Computing (ISORC'98)*, pages 342–349, Kyoto, Japan, April 1998. IEEE Computer Society, IEEE Computer Society Press.

[13] Bernhard Plattner. Real-time execution monitoring. *IEEE Transactions on Software Engineering*, SE-10(6):756–764, November 1984.

[14] R2D2 debugger, Zentropix. www.zentropix.com.

[15] Jeffery J. P. Tsai, Kwang-Ya Fang, and Horng-Yuan Chen. A noninvasive architecture to monitor real-time distributed systems. *Computer*, 23(3):11–23, March 1990.

[16] Virtutech Simics v0.97/sun4u. www.simics.com.