Simulation-Based Debugging and Profiling of Soft Real-Time Applications

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

Abstract

We present a temporal debugger, capable of examining time flow of soft real-time applications. The debugger is attached to a simulator modelling an entire workstation in sufficient detail to run commodity operating systems and workloads. Whereas traditional debuggers need to interfere with program execution, thereby changing the temporal behaviour, a debugger operating on a simulated system does not disturb the timing of the target program. The temporal debugger therefore allows non-intrusive and reproducible debugging of real-time applications in general-purpose operating systems.

We have implemented the temporal debugger by modifying the GNU debugger to operate on applications running in a simulated machine. Debugger implementation is difficult because the debugger expects application-related data, whereas the simulator provides low-level data. We introduce a technique, virtual machine translation, for mapping simulator data to the debugger by parsing operating system data structures in the simulated system.

The debugger environment allows execution time measurement and collection of statistics from multiple levels of the system: hardware, operating system, and application. We show how this profiling data can help a programmer explain missed deadlines; the statistics are correlated to deadline violations, thereby exposing differences between executions when deadlines are met and when they are missed.

The temporal debugger is demonstrated with a debugging session of an MPEG video decoder running in a Linux system. We show how the debugger can be used to detect that the decoder fails to render some frames in time, and how the causes for deadline violations are found by correlating rendering time with runtime statistics.

Keywords: Complete System Simulation, GDB, Linux, Operating Systems, Real-Time Profiling, Simics, Soft Real Time Systems, Temporal Debugging

1 Introduction

Applications demanding short response times and throughput guarantees are increasingly common in general purpose computing environments. These so called soft real-time applications include video decoders, games, and online trading systems. They need to provide both high throughput performance and service requests in predictable time. Unfortunately, it is difficult to program applications to meet these requirements. The difficulty of the task is compounded by the fact that there are few tools available to aid the programmer.

Conventional tools in mainstream systems have no support for analysing real-time requirements and generally ignore the fact that application correctness depends on execution time. Tools specifically tar-

geting soft real-time applications are needed in order to help the programmer minimise the number of missed deadlines occurring during execution. Such tools should locate missed deadlines, explain why the program did not meet performance expectations and provide guidelines for correcting the problem. As the perceived quality of soft real-time applications depends on temporal correctness, they are sensitive to time variations during execution. The tools must therefore keep time distortion low and should be able to display time flow in detail.

The debugger is one of the most important tools for computer programming. Traditional debuggers, however, are inadequate for real-time applications, as they lack a notion of temporal correctness and interfere with program execution. In embedded real-time system design, these problems have been addressed by simulator-based debuggers [2, 22]. Historically, embedded systems have been so much slower than engineering workstations that temporally accurate simulation has been feasible. Until recently, it has not been practical to use simulators to study commodity desktop and server computer systems, as the simulators have been too slow.

Advances in simulation technology have resulted in simulators providing an approximate, but reasonably accurate timing model while executing roughly 50-200 times slower than native execution [8, 11]. These simulators, referred to as complete system simulators, model an entire workstation at the instruction set level, and run 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 systems. Furthermore, as the simulators execute both application and operating system, they provide an authentic model of performance and timing factors caused by the operating system.

In this paper, we present a temporal debugger for soft real-time applications. The debugger is based on the GNU debugger [6], modified to debug Linux applications running on a complete system simulator. As the target program and operating system are executed in an artificial environment, they can be suspended and probed without affecting the execution. Thus, the debugger allows non-intrusive and reproducible application debugging. The debugger is also able to present the time flow of the simulated system using the simulator's time model. Figure 1 shows the debugger user interface. The window in the background is the simulated console, showing output from UltraSPARC Linux during boot.

Figure 1: Temporal debugger user interface.

We have earlier studied temporal debugging of operating systems [1]. An operating system executes directly on the physical machine, which corresponds to the model provided by the simulator. It is therefore straightforward to debug operating systems by using services provided by the simulator. Debugging user level applications, however, is non-trivial. Unix programs execute inside protected environments, provided by the operating system. These environments are referred to as *virtual machines*. As a debugger expects information related to a process, it is necessary to map physical data, as presented by the simulator,

to process-related data. The information necessary to perform this mapping in a robust manner exists only in the operating system running in the simulated machine. We have therefore invented a technique, *virtual machine translation* (VMT), for reading memory and register contents of Linux processes by parsing kernel data structures.

We also describe how the services of the temporal debugger can be used to profile real-time applications. The debugger facilitates the explanation of deadline misses by providing measurements of system metrics, for example counts of hardware and operating system events, application variables, and execution time of functions. This allows a programmer to compare measurements and find differences in execution between successful iterations and iterations where a deadline was missed.

Section 2 provides some background on complete system simulation and the benefits of using it for real-time analysis. A description of the design and implementation of the temporal debugger, including the virtual machine translation, is presented in Section 3. It also contains an example debugging session of an MPEG video decoder running in a Linux system. In Section 4, we describe how the debugger allows measurement of system metrics and how the results are used for profiling the video decoder. Section 5 contains a short survey of related work in real-time debugging and simulation. Conclusions and future work are presented in Section 6.

2 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 the 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, the simulator runs unmodified operating system and application software. This limits the sources of errors to those introduced by the hardware model, and by models of simulation input feed.

2.1 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. In contrast, 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 the workload and the time scale of its deadlines. For large workloads, a reasonably accurate time model is usually sufficient to obtain a coarse understanding of the timing behaviour in the system.

2.2 Reproducibility

A simulated computer is purely artificial and deterministic. Thus, a simulated system starting execution in a known state will always execute along the same path. This property is essential, 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 technique is similar to the methodology used for debugging logical correctness of conventional programs. As time is part of the state the user wishes to verify, it is crucial that temporal behaviour is preserved between debugging sessions.

Execution is reproducible only if all input to the simulator is deterministic. As the physical world is inherently unpredictable, the simulator must not communicate with real, unpredictable input sources. Instead, all input sources must be modelled, with traces, synthetic models, or other simulators. In order to obtain a realistic input feed, the simulator may be connected to the real world once, with recording enabled. Further experiments can then use the

recorded trace.

2.3 Probe immunity

In physical systems, measurement of the system generally affects its behaviour. This is referred to as *probe effect*. Although measurements change time flow in the system, soft real-time applications tend to have deadlines separated by long intervals, and are therefore not very sensitive to these changes. The probe effect does, however, put a limit on the amount of measurement. Nevertheless, a temporal debugger requires non-intrusive probing to achieve reproducible execution. Even a minimal change in execution time flow could affect decisions in the applications and the operating system, resulting in a completely different execution path.

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 Simics

The simulator used for this work is Virtutech Simics [21]. 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 or a part of a memory or device stall. In this mode, Simics has a rather simple view of the timing of a modern system, and assumes a linear penalty for events such as translation look-aside buffer (TLB) misses, data cache misses, and instruction cache misses. Simics supports efficient programming of models for the components most important for performance modelling: cache hierarchies, synchronisation in multiprocessor machines, and I/O devices.

Simics's programming interface allows the user to add more detailed timing models. If an application is constrained by a particular bottleneck in the system, the user can program a detailed model of that part and trade some simulation performance for a better model. The performance impact can be significant if the user adds an inefficient timing model for a central component, such as the CPU pipeline. In this case, performance can be improved by switching models at runtime. Examples of simulation model tradeoffs and dynamic switching of models have been presented by Herrod [8].

3 Debugging real-time applications

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 user must 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. A debugger for real-time programs, however, 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 allows correct debugging of real-time applications.

3.1 Simulation-based debugging

A complete debugger setup consists of the debugger program and a target machine running the debugged program. Figure 2 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.

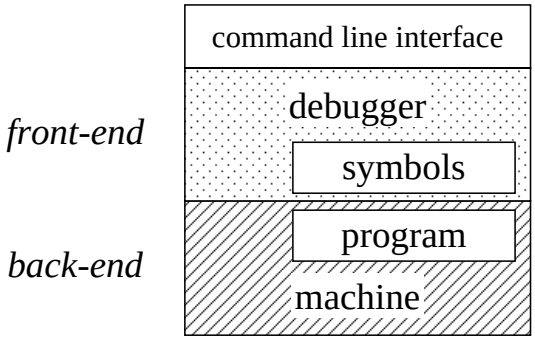

Figure 2: Debugger structure.

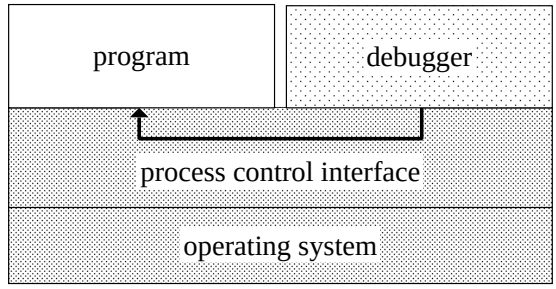

Figure 3: Traditional debugger.

A debugger requires a certain set of primitives for probing and controlling 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.

In addition to primitives required by a debugger front-end, the simulator provides services not normally available in a 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.

3.2 User process debugging

In time sharing operating systems, such as Linux, each program runs in a protected environment, with private registers, memory, and operating system resources. This is referred to as a *virtual machine*. In order for a debugger to debug a program, it needs to control execution and probe state of the corresponding virtual machine. A traditional debugger does not implement this controlling mechanism. Instead, it uses an interface provided by the operating system, illustrated in Figure 3. The operating system performs necessary administrative tasks, such as virtual memory translation and virtual register lookups. It thereby exports an image of a consistent virtual machine to the debugger.

As a simulation-based debugger runs in a different operating system than the target process, it cannot use operating system services to probe the target, and is restricted to simulator services. The simulator provides primitives for probing hardware state, such as register, memory, and disk contents. This information is useful for probing the operating system itself, which is the program running directly on the hardware. Without post-processing, however, the information is not useful for analysing user space processes.

In order to support debugging of processes in the simulated system, we introduce an intermediate filter between the debugger and the simulator, the *virtual machine translator* (VMT). The VMT answers a debugger's queries for virtual machine state. Queries for memory content refer to virtual addresses, and must be translated to physical addresses. The VMT performs this translation, which is normally provided by the operating system. It starts by looking up the head of the process list, which is a global variable whose address is found in the kernel symbol table. It finds the appropriate process entry by following pointers referring to data structures in the simulated memory. It proceeds in the same way in the process's page table until the mapping for the virtual address is found. If the page resides in physical memory, the VMT queries the simulator for the contents and responds to the debugger. If the sought page is paged out to disk, the VMT needs to walk the kernel data structures further to find which disk block it was written to and query the simulator for disk contents.

The debuggers are separate programs, communi-

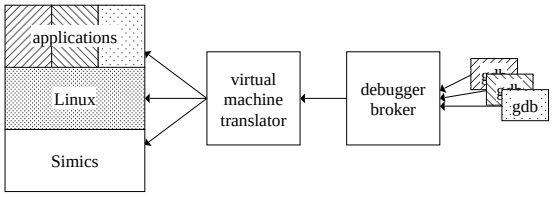

Figure 4: Simulation-based debugger. Arrows represent directions of data queries.

cating with the VMT via a proxy, a *debugger broker*, which supports concurrent debugging of multiple processes in the simulated system. The broker keeps state for each debugger connected and maps debugger queries to the corresponding virtual machine. It controls simulator execution, stepping forward only when all connected debuggers are ready to execute. The setup is shown in Figure 4. The debugger broker also makes sure all debugger breakpoints refer to physical memory. It maintains a list of breakpoints in use and inserts (removes) simulation breakpoints when code pages are mapped (unmapped) to physical memory.

3.3 Virtual machine translator implementation

Writing code for parsing data structures of a simulated system is a tedious and error-prone process. In order to simplify and automate the task, a few engineering tricks have been employed.

The symbol table for the operating system kernel contains information on the size and memory layout of all types defined in the kernel source code. We have implemented a meta-tool that uses a symbolic debugger to parse a symbol table, and then generates a C++ class for each type found in the table. The code generated includes routines for constructing objects by probing simulated memory. This code generator provides a programming environment with strong typing, enabling compile time checking for simple mistakes. It also facilitates performance optimisations, for example cached variable lookups or lazy memory probing. Moreover, the Linux kernel is written in C, which is a subset of C++. Therefore, definitions from the kernel source code can be reused with little or no modification. The dependencies between VMT modules are illustrated in Figure 5.

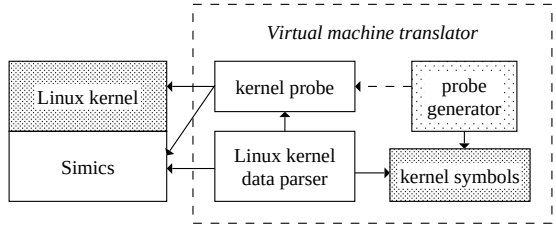

Figure 5: Implementation of the virtual machine translator. Solid arrows represent data queries. The dashed arrow represents code generation.

The Linux-specific module for parsing kernel structures uses the generated code whenever it needs to probe kernel memory. It is therefore independent of the exact code layout, and needs to query the simulator directly only occasionally. This independence provides some portability and robustness when porting to new architectures or kernel versions.

3.4 Real-time debugging example

As a demonstration of debugging soft real-time applications, we present examples from a debugging session of the MPEG video decoder `mpeg_play` [13]. The decoder displays a video clip with a frame rate of 30 frames per second. In this example, Simics is used to model an UltraSPARC workstation. The simulated machine boots from a disk image containing an installation of UltraPenguin Linux 1.1. The system is configured to run the MPEG decoder at boot. The system also runs standard Unix daemons and, in order to make the workload more complex, a CPU-bound background task with low priority. The startup sequence is shown in Figure 1.

We have executed simulation forward until the video decoder is started, and attached GDB to the `mpeg_play` process. In order to present simulated time flow, the modified GDB provides a magic variable, `sim_time`. Whenever the variable is referenced, GDB queries the simulator for the number of cycles executed since boot. As the variable is integrated into

GDB, it can be used as any other program variable, for example in GDB scripting or conditional breakpoints. Other types of simulator information, such as hardware or operating system statistics, can be presented in a similar fashion.

Listing 1 shows how we set a breakpoint at the routine `ExecuteDisplay`, which is called at the end of the rendering loop. We use the `sim_time` variable to check whether the deadline was met for each frame. The machine runs at 225 simulated megahertz, and must therefore render a frame in less than 7.5 million cycles to meet the deadline. A short GDB command sequence locates the first missed deadline for us.

Listing 1 Example of locating a missed deadline.

```
(gdb-sim) break ExecuteDisplay
Breakpoint 1 at 0x200cc: file gdith.c, line 942.
(gdb-sim) continue
Continuing.

Breakpoint 1, ExecuteDisplay (vid_stream=0x11a160,
    frame_increment=1, xinfo=0x116654) at gdith.c:942
942       if (xinfo!=NULL) display=xinfo->display;
(gdb-sim) display $sim_time
$1 = 979949561
(gdb-sim) set $start = $sim_time
(gdb-sim) commands 1
Type commands for when breakpoint 1 is hit, one per line.
End with a line saying just "end".
>silent
>printf "Frame completed at %d, in %d cycles\n", $sim_time,
    $sim_time - $start
>if $sim_time - $start > 7500000
 >printf "Deadline missed\n"
 >else
 >set $start = $sim_time
 >continue
 >end
>end
(gdb-sim) continue
Continuing.
Frame completed at 980856912, in 907351 cycles
Frame completed at 981744876, in 887964 cycles
Frame completed at 989462277, in 7717401 cycles
Deadline missed
(gdb-sim)
```

When a missed deadline is found, the user can restart the session and investigate the unsatisfactory behaviour in detail. As the simulated machine is deterministic, an identical execution will be observed.

4 Profiling real-time applications

A profiling tool assists a programmer in deciding where in a program to make optimisations. The most common type of profiling tool is the performance profiler. It measures execution time spent in different code sections, and shows the programmer where the majority of execution time is spent. This gives the programmer a hint on where to concentrate his efforts to improve program performance. Certain advanced profilers also collect statistics on how well hardware or operating systems resources are used. Such statistics may explain why a program fails to meet performance expectations, and are used to suggest to the programmer how to modify the program.

When optimising soft real-time applications, the programmer aims to minimise the number of missed deadlines. Traditional profiling tools are inadequate for this purpose, as they present total time spent in a code block, measured over the whole execution. Instead, a real-time profiler should present differences in execution times between iterations when the deadline was missed and iterations when the deadline was met. This presentation gives the programmer a hint as to which routines he should modify to eliminate deadline misses.

4.1 Source code profiling example

In order to explain why the video decoder missed some deadlines in the example in Section 3.4, we will measure and compare time spent in a subroutine. By running `mpeg_play` in a conventional performance profiler, we observe that most of the execution time is spent in the function `OrderedDitherImage`. We set breakpoints at the start and end of this function and measure execution time for each iteration with a simple script (similar to that in Listing 1). The results are shown in Figure 6. The axes correspond to execution time spent in `OrderedDitherImage` and rendering time for the whole frame. Surprisingly, `OrderedDitherImage` executes in constant time — except for a few odd values, caused by interrupts and page faults — and there is no correlation between a

deadline miss and execution time spent in this function.

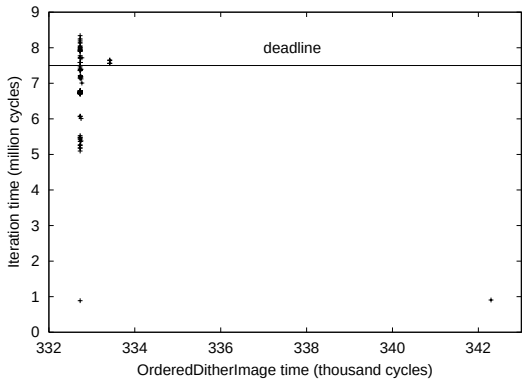

Figure 6: Correlation to OrderedDitherImage execution time.

We could continue searching for causes of missed deadlines by measuring execution time spent in other functions, hoping they would show better correlation. Instead, we will present how to perform instrumented profiling by correlating deadline violations with other metrics.

4.2 Instrumented profiling

Execution time for a code section may differ between iterations, even though the program executes along identical paths. In a general purpose operating system, the stochastic behaviour of the memory hierarchy, I/O devices, and competing processes affects execution time in unpredictable ways.

Variations in execution time between iterations may be explained by counting system events, and comparing event statistics for each iteration with execution time. We collect statistics from different levels in the system using different methods.

- Hardware level. The simulator counts performance related hardware events occurring in the simulated machine. Events counted include cache misses, TLB misses, and I/O transactions.
- Operating system level. The virtual machine translator is able to collect three types of statistics.
  - Event count. Some operating system events, such as page faults and context switches, have great impact on performance. These are counted by inserting a breakpoint in a corresponding kernel routine, for example the scheduler. Whenever the breakpoint is triggered, the event counter is incremented.
  - Kernel variable values. Kernel variables may be sampled by reading simulated memory, and the addresses are found in the kernel symbol table. The process id of the current task is an example of a useful kernel variable.
  - Computed kernel variables. Variable values that are not available immediately in memory are calculated using a specific routine for each variable. A traditional debugger would execute code in the target machine to perform such computations. This method is unacceptable for a simulation-based debugger, as it would modify the simulation and make it non-deterministic. Instead, the user must implement a lookup routine for each computed variable. The length of the run queue is an example of a computed kernel variable.
- Application level. The user may choose to include statistics from application events and variables in the same way as for the operating system. Routines for collecting application statistics are best implemented on top of the debugger interface, as the debugger services are necessary. The code generation techniques used for implementing the virtual machine translator (described in Section 3.3) are applicable also for programming application level probing.

The ability to collect execution statistics from multiple levels in a computer system is very useful. It allows efficient problem solving without requiring knowledge about the nature of the problem. Most

performance tools collect data from a single level, and therefore require that the programmer knows which tool to use and where to look.

4.3 Instrumented profiling example

We can now proceed to try to explain the missed deadlines in the MPEG video decoder by correlating statistics from the application and operating system level.

Listing 2 Reading application variables.

```
(gdb-sim) list video.h:100
97      /* Macros for picture code type. */
98
99      #define I_TYPE 1
100     #define P_TYPE 2
101     #define B_TYPE 3
(gdb-sim) break ExecuteDisplay
Breakpoint 1 at 0x200cc: file gdith.c, line 942.

(gdb-sim) set $start = $sim_time
(gdb-sim) commands 1
Type commands for when breakpoint 1 is hit, one per line.
End with a line saying just "end".
>silent
>printf "Frame number %d", TotalFrameCount
>printf ", type %d", vid_stream->picture.code_type
>printf ", %d cycles\n", $sim_time - $start
>set $start = $sim_time
>continue
>end
(gdb-sim) continue
Continuing.
Frame number 9, type 1, 7933769 cycles
Frame number 10, type 3, 7179460 cycles
Frame number 11, type 3, 6730860 cycles
Frame number 12, type 2, 7364510 cycles
Frame number 13, type 3, 6062254 cycles
Frame number 14, type 3, 6742310 cycles
Frame number 15, type 1, 7897064 cycles
```

An MPEG stream contains a compressed representation of video frames. For some frames, the whole frame is encoded as a JPEG picture. These are referred to as I frames. Other frames are represented only by the difference to past or future frames (P and B frames). The decoder handles each frame type in a different way. Thus, decoding time is probably dependent on the frame type. In order to find out whether most deadline violations occur for a particular frame type, we instruct the debugger to break at each iteration and print the variable containing the frame type code. The debugger commands are shown in Listing 2.

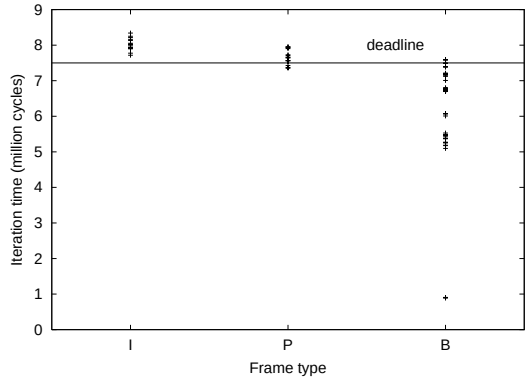

Figure 7: Correlation to MPEG frame type.

The results, shown in Figure 7, indicate that most deadline violations occur while decoding I and P frames. This implies that the programmer of `mpeg_play` should concentrate optimisation efforts on the code executed during I and P frame decoding. Optimising code performance is often difficult, however, and the programmer may want to tune other system parameters to improve application quality. Figure 7 shows that rendering time varies, even for frames of the same type. We suspect that this variance is caused by the background load. In order to measure the amount of CPU resources consumed by the competing program, we instruct the simulator to insert a breakpoint at the point where Linux returns from kernel to user space, and inform us whenever a new process is scheduled. This procedure is shown in Listing 3. A comparison of rendering time and time spent in other processes is shown in Figure 8, indicating a clear correlation. Therefore, limiting CPU usage of other processes, for example by using a real-time scheduling policy, may improve application performance in the scenario presented. The correlation may also be a side-effect, if the application spends a large amount of time waiting for I/O. By measuring time spent on blocking system calls, the programmer can find out whether the application misses deadlines because of I/O wait. He can continue validating his theories this way, through further measurements and correlations, until he has obtained sufficient under-

standing of the causes of deadline violations.

Listing 3 Measuring context switches.

```
(gdb-sim) break ExecuteDisplay
Breakpoint 1 at 0x200cc: file gdith.c, line 942.
(gdb-sim) set sim os-context-switches on
(gdb-sim) continue
Continuing.
Context switch from 105 to 106 at 981749467.
Context switch from 106 to 105 at 987309788.

Breakpoint 1, ExecuteDisplay (vid_stream=0x11a160,
    frame_increment=1, xinfo=0x116654) at gdith.c:942
q942        if (xinfo!=NULL) display=xinfo->display;
(gdb-sim)
```

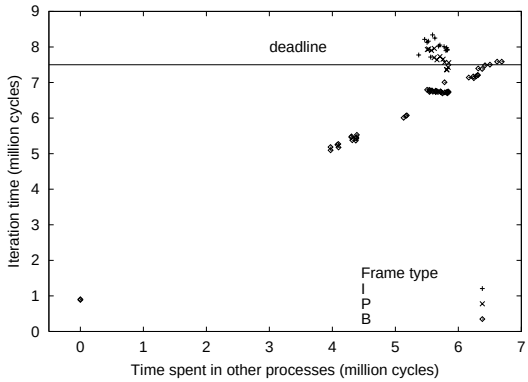

Figure 8: Correlation to time spent in other processes.

5 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. There are however vendors providing support for alternative debugging methods. Some of these methods are discussed below.

Developers of programs for small embedded systems commonly 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 vendors, for example Motorola [2] and Wind River [22], provide simulators with models of caches and CPU pipeline, resulting in good execution time modelling. The tools generally available are either too incomplete to run general-purpose operating systems or too slow to run desktop applications.

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 [7, 16, 20] or additional program code [4, 15]. Both approaches are inconvenient, and the amount of monitoring is limited. Furthermore, data exploration is limited to the data subset collected.

The R2D2 debugger [17] 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. Sessions cannot be authentically repeated, however, and the target system may be unable to provide information when it is under stress.

Mueller and Whalley [14] 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. This prediction does not take operating system effects into account and works best for small programs.

The work in this paper is made possible by many different advances in simulator implementation technology [3, 5, 8, 9, 10, 19]. A few simulator research groups have managed to model a complete hardware system with sufficient detail and efficiency to run commodity operating systems with large workloads [5, 11]. The SimOS project [8] has made similar achievements, although the simulator presented does not model a complete binary interface and requires operating system modifications. Due to the accurate timing model provided, complete system simulators have proven to be effective tools for performance profiling [8, 12, 18].

6 Conclusions

The primary contribution of this paper is the temporal debugger, an important tool for development of soft real-time applications. The temporal debugger implementation is based on the GNU debugger, modified to connect to a simulator modelling a complete workstation. The simulator runs unmodified operating system and application software and allows reproducible analysis of system time flow.

The major difficulty in debugger implementation is the mapping of low-level simulation data to application-related data useful to the debugger. We address this problem using a new technique, virtual machine translation, which operates by traversing data structures in the operating system kernel. This technique has been implemented for UltraSPARC Linux systems.

We demonstrate how the temporal debugger can be used to detect missed deadlines in an MPEG video decoder, running in Linux. We also show how the debugger makes application and operating system internals visible during simulation, enabling collection of runtime statistics from different system levels. The statistics are correlated to deadline misses in the video decoder, thereby explaining the causes of deadline violations.

Currently, the temporal debugger environment supports manual examination, or simple automated operations using GDB scripts. A programmer would benefit more from an automated profiler, visualising correlations between missed deadlines and many different types of statistics, for example cache misses, function call arguments, page faults, and branch decisions. As the debugger environment allows non-intrusive probing, an automated tool can measure large amounts of data eagerly, without compromising accuracy.

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, IEEE Computer Society Press, August 2000.

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

[3] Robert C. Bedichek. Some efficient architecture simulation techniques. In USENIX Association, editor, *Proceedings of the Winter 1990 USENIX Conference, January 22–26, 1990, Washington, DC, USA*, pages 53–64, Berkeley, CA, USA, January 1990. USENIX.

[4] 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.

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

[6] The GNU debugger, version 5.0. http://sources.redhat.com/gdb.

[7] 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.

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

[9] Peter Magnusson and Bengt Werner. Efficient memory simulation in SimICS. In *Proceedings of the 28th Annual Simulation Symposium*, 1995.

[10] Peter S. Magnusson. Efficient instruction cache simulation and execution profiling with a threaded-code interpreter. In *Proceedings of Winter Simulation Conference 97*, 1997.

[11] 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.

[12] Johan Montelius and Peter Magnusson. Using SimICS to evaluate the Penny system. In Jan Małuszyński, editor, *Proceedings of the International Symposium on Logic Programming (ILPS-97)*, pages 133–148, Cambridge, October 13–16 1997. MIT Press.

[13] The Berkeley MPEG player, version 2.3. http://bmrc.berkeley.edu/frame/research/-mpeg/mpeg_play.html.

[14] 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.

[15] 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.

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

[17] The R2D2 debugger, Zentropix. http://www.zentropix.com.

[18] Mendel Rosenblum, Edouard Bugnion, Scott Devine, and Stephen Alan Herrod. Using the SimOS machine simulator to study complex computer systems. *ACM Transactions on Modeling and Computer Simulation*, 7(1):78–103, January 1997.

[19] Mendel Rosenblum, Stephen A. Herrod, Emmett Witchel, and Anoop Gupta. Complete computer system simulation: The SimOS approach. *IEEE parallel and distributed technology: systems and applications*, 3(4):34–43, Winter 1995.

[20] 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.

[21] Virtutech Simics v0.97/sun4u. http://www.simics.com.

[22] Wind River. http://www.windriver.com.