Temporal debugging and profiling of multimedia applications

Lars Albertsson

Swedish Institute of Computer Science, Box 1263, S-164 29 Kista, Sweden

ABSTRACT

We present a temporal debugger, capable of examining time flow of applications in general-purpose computer systems. The debugger is attached to a complete system simulator, which models an entire workstation in sufficient detail to run commodity operating systems and workloads. Unlike traditional debuggers, a debugger operating on a simulated system does not disturb the timing of the target program, allowing reproducible experiments and large amounts of instrumentation and monitoring without intrusion.

We have implemented the temporal debugger by modifying the GNU debugger to operate on applications in a simulated Linux system. 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 collection of performance statistics from multiple abstraction levels: hardware, operating system, and application level. We show how this data can be used to profile quality of service performance of a video decoder. The debugger is used to detect display jitter, and by correlating runtime statistics to image rendering time, we expose deviations when the application is unable to render an image in time, thereby locating the cause of the display jitter.

Keywords: Complete system simulation, GDB, Linux, operating systems, real-time profiling, Simics, soft real-time systems, temporal debugging

1. INTRODUCTION

The increasing popularity of multimedia applications places new requirements on application development methods and tools. Multimedia applications demand both high throughput performance and predictable quality of service (QoS), for example a smooth display of images in a video conference tool, or short response times in a virtual reality application. Similarly to traditional applications, multimedia applications are developed in an iterative manner: The developers design and implement an initial version of the program, which is tested, and if it does not meet requirements, they use monitoring tools, such as debuggers and profilers to understand the deficiencies. Unfortunately, it is difficult to test, debug, and profile QoS properties in general-purpose computer systems using existing tools. Traditional testing and debugging tools focus on functional correctness, and ignore the fact that multimedia applications have soft real-time requirements and execute with timing constraints. Moreover, traditional testing and debugging techniques require that a problem is reproducible, and are inadequate for capturing timing-related problems, such as QoS deficiencies. Conventional performance profilers focus on throughput performance, and present subroutine execution time measurements aggregated over the whole execution. They are therefore unable to locate subroutines causing execution time jitter.

We propose the use of *complete system simulation* for analysing multimedia computer systems. By executing the applications and operating system in a simulated machine instead of a real computer, we create a robust testing and debugging environment, which supports detailed, non-intrusive observation and performance monitoring. A complete system simulator is implemented entirely in software and provides an accurate model of all the hardware in a commodity computer system, allowing operating systems and applications to run without modifications. It also provides a timing model and is able to present execution time flow in the simulated system.

Further author information: e-mail: lalle@sics.se, telephone: +46 8 633 1551

The benefits of executing a program in a simulator are:

- **Reproducibility.** Execution in a complete system simulator is deterministic, which allows a programmer to reproduce experiments and observe identical executions.
- **Visibility.** Because the simulated machine is a software model, the state of the machine, as well as the operating system and application state, is visible and can be probed at any time during simulation.
- **Non-intrusive probing.** Probing the system does not affect its execution. Large amounts of statistics can therefore be collected without changing the timing of the system nor compromising measurement accuracy. Because the simulator executes both the application and the operating system, the performance and timing behaviour of the operating system is authentic.
- **Flexibility.** Parameters that are difficult to modify on real systems, such as hardware configuration values, can easily be changed.

These properties are very desirable when analysing complex computer systems, and are difficult to achieve through other means. There are, however, complications involved with complete system simulation:

- **Performance.** Most simulators execute programs many orders of magnitude more slowly than real machines, limiting their use to tiny applications. Recent advances in implementation technology, however, have led to the development of simulators capable of modelling complete general-purpose computers with reasonable efficiency and accuracy, allowing simulation of large workloads.[10,15]
- **Mapping data to application level.** Complete system simulators model only the hardware in a system, and provides a programmer with low-level data, such as physical memory contents. The application programmer is usually interested in the application state and contents of the application's virtual memory, which is hidden by the operating system. It is therefore necessary to provide a mapping between virtual and physical memory in order to present application level data.
- **Data collection.** A programmer profiling a multimedia application will find performance statistics from the application, operating system, and hardware level useful. A simulation environment should provide programmable performance data collection from multiple abstraction levels in the system.

With this paper, we attempt to address these complications. We present a temporal debugger for applications running in Simics,[34] a commercially available complete system simulator. The debugger is based on the GNU debugger (GDB), modified to operate on applications in a simulated Linux system, and to report time flow in the simulated system. We describe *virtual machine translation*, a technique for mapping simulation data to application level by parsing operating system data structures. The temporal debugger is demonstrated with an example debugging session of an MPEG video decoder. We also show an example of how the debugger environment can be used to collect performance data from multiple abstraction levels, and how these data can be used for profiling the video decoder.

2. COMPLETE SYSTEM SIMULATION

A complete system simulator is a computer implemented in software. It interprets a program's instructions and lets the instruction operands refer to a model of machine state, which includes register, memory, and disk contents. The simulator contains a model of all significant devices in a computer. The device models provide the same binary interface as the corresponding hardware, and the simulator can therefore execute unmodified binaries of operating systems and applications. Figure 1 shows the components of a complete system simulator. The device models are separate modules, which can receive input from programmed models or be connected to the real world through corresponding devices in the host machine.

The simulator also provides a temporal model and advances simulated time after each interpreted instruction. Whereas the functional model must be almost authentic to run unmodified programs, 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

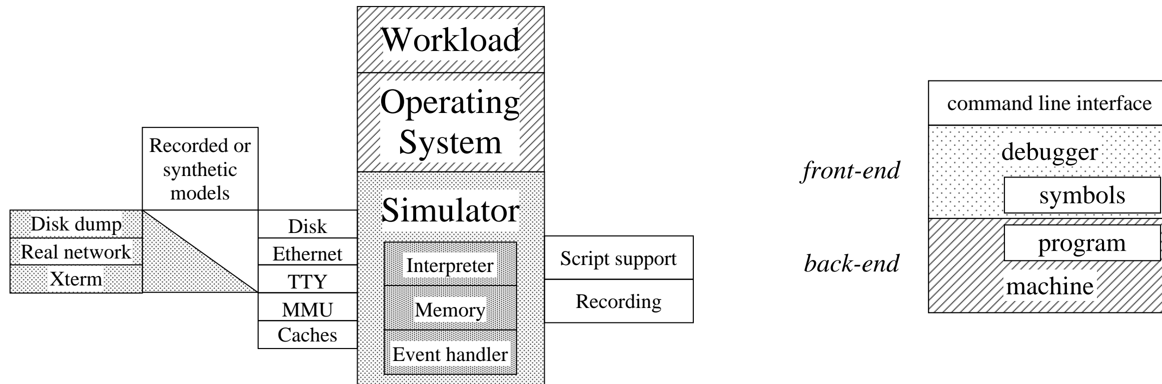

Figure 1: Simulator components.

Figure 2: Debugger structure.

by using approximate 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.

2.1. Reproducibility

A simulated computer is purely artificial and deterministic. 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 programmer debugging an application's QoS properties 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. Because time is part of the state the programmer 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. The physical world is inherently unpredictable, and 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.

2.2. Non-intrusive probing

In physical systems, measurement of the system generally affects its behaviour. This is referred to as the *probe effect*.[7] Although measurements change time flow in the system, multimedia applications tend to have long periods, 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 independent of 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.

2.3. Simics

The simulator used for this work is Virtutech Simics.[34] Simics simulates the SPARC V9 instruction set and models single or multiprocessor systems corresponding to the sun4u architecture from Sun Microsystems. Simics is also capable of simulating distributed systems by connecting multiple simulated machines to a simulated network.

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. When simulating at this level of detail, Simics executes programs approximately 100 times more slowly than the host machine.

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 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. Simulation performance can also be improved by saving a checkpoint of simulation state prior to the interesting part of an experiment.

3. TEMPORAL DEBUGGING

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 time-sensitive 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 temporal debugging of time-sensitive 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.

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 most important service for analysis of time-sensitive applications is the ability to present the time of the simulated system. It enables the programmer to step through 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 a controlling and probing 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.

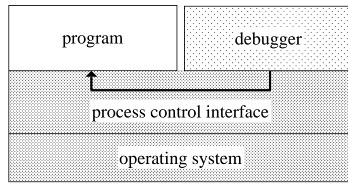

Figure 3: Traditional debugger.

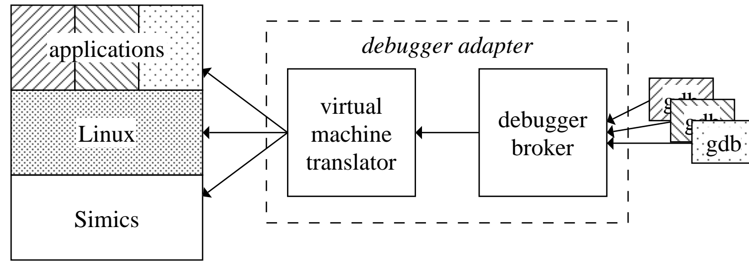

Figure 4: Simulation-based debugger. Arrows represent directions of data queries.

In order to support debugging of processes in the simulated system, we introduce an intermediate filter between the debugger and the simulator, a *debugger adapter*. The debugger adapter consist of two modules, a *virtual machine translator* (VMT)* and a *debugger broker*. The VMT is responsible for answering debugger 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 otherwise 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 has been written 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, communicating with the VMT via the 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. It is possible to simplify and automate the task by reusing information available in the operating system kernel's debug symbol table. The symbol table 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 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++. Definitions from the kernel source code can therefore be reused with little or no modification. The dependencies between VMT modules are illustrated in Figure 5.

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. Temporal 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`.[21] 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

*The VMT has previously been briefly described by the author.[1]

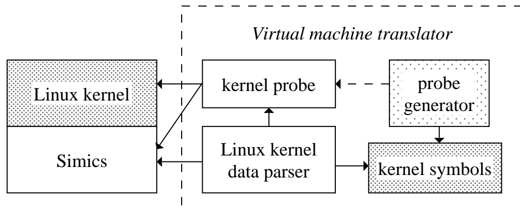

Figure 5. Implementation of the virtual machine translator. Solid arrows represent data queries. The dashed arrow represents code generation.

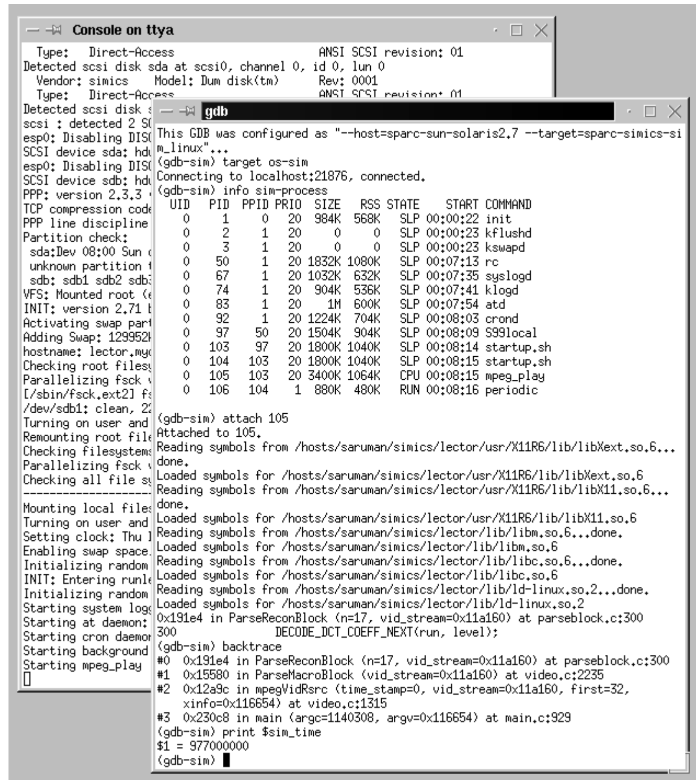

Figure 6: Temporal debugger user interface.

priority. The startup sequence is shown in Figure 6. The window in the background is the simulated console, showing output from UltraSPARC Linux during boot.

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 for 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 decoder was able to render the previous frame in time. The machine runs at 225 simulated megahertz, and must therefore render a frame in less than 7.5 million cycles to display frames without jitter. If this deadline is missed, the image will flicker, affecting the perceived quality of the application. A short GDB command sequence locates the first missed deadline for us.

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 tells the programmer 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.

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)
```

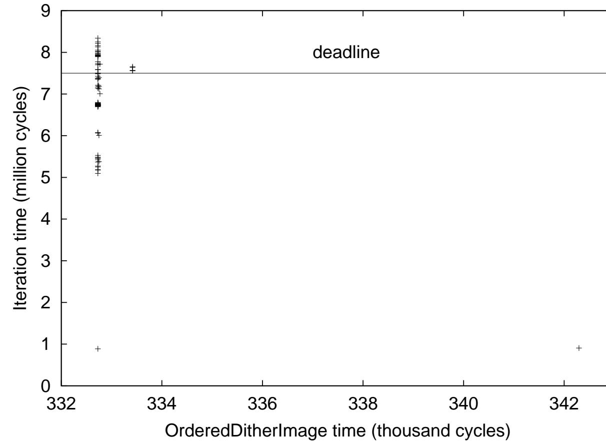

Figure 7: Correlation to OrderedDitherImage execution time.

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

We will try to explain why the video decoder missed some deadlines in the example in Section 3.4 by measuring and comparing 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 7. 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. This illustrates that conventional profilers are inadequate for profiling soft real-time applications.

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 identifier 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, since the debugger services are needed to probe application state. 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 will proceed with explaining the missed deadlines in the MPEG video decoder by correlating statistics from the application and operating system level.

An MPEG stream contains a compressed representation of video frames. Some frames are encoded as a JPEG pictures (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.

The results, shown in Figure 8, 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

Listing 2 Reading application variables.

```
(gdb-sim) list video.h:99
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, 67308 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
```

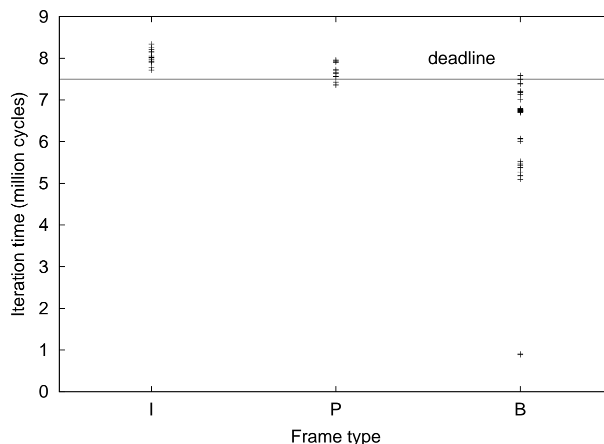

Figure 8: Correlation to MPEG frame type.

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)
```

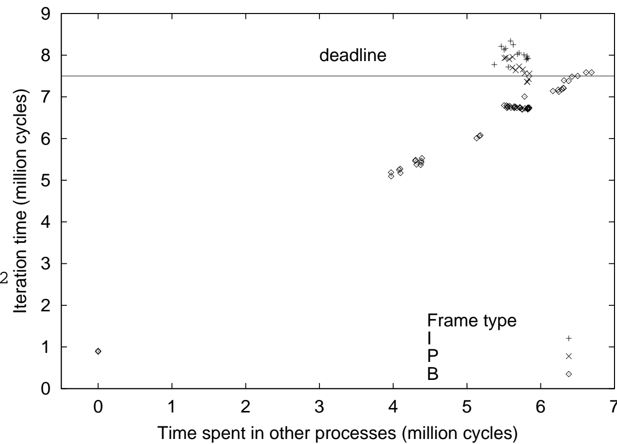

Figure 9: Correlation to time spent in other processes.

decoding. Optimising code performance is often difficult, however, and the programmer may want to tune other system parameters to improve application quality. Figure 8 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 9, 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 can 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 understanding of the causes of deadline violations.

5. RELATED WORK

Apart from multimedia applications programming, there are other contexts in which temporally correct and deterministic debugging is useful. The real-time research community has come up with many proposals for capturing and reproducing program time flow, mostly focusing on embedded systems. Techniques for preserving time flow are also of interest in parallel programming research; in order to recreate race conditions, programmers debugging multithreaded applications need to preserve thread scheduling decisions, which are usually triggered by timer interrupts.

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,[35] 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. The performance of such simulators can be increased by using special hardware support.[11]

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,[5,16,24,31,32] which is inconvenient and inappropriate for modern processors where the results of most program operations are contained in caches and thereby hidden from the monitoring hardware. Traces can also be generated in software by additional program code,[9,23,27] run-time system,[4,19] operating system,[18,29] or by an external monitoring program.[17] Generating a trace in software is intrusive, and the amount of monitoring is limited. The problems regarding intrusive monitoring and related research has been further described by Marinescu.[16] Moreover, data exploration in traces is limited to the data subset collected, unless the program execution is reproducible.

Mueller and Whalley present a debugger for real-time applications, which is complemented by a cache simulator predicting program execution time.[22] The debugger operates on a running program, and the cache simulator estimates execution time by inspecting memory references. This approach ignores time spent executing the operating system and only works for programs whose execution flow is independent of elapsed time.

In order to provide reproducible debugging sessions for real-time and multithreaded applications, a technique called *deterministic replay* has been proposed. A monitoring system collects information on application input and events driven by the clock, such as interrupts and scheduling decisions. When the system is executed in a debugger, the input is taken from the recorded trace, and all clock-based events are replaced with the events recorded in the trace. The timing information and interleavings of events in the original execution are thereby recreated. The monitoring and replay system can be implemented in the operating system,[30] run-time system,[12, 28] or by using hardware support.[33]

The work in this paper is made possible by many different advances in simulator implementation technology.[3, 6, 10, 13, 14, 26] A few simulation research groups have managed to model a complete hardware system with sufficient detail and efficiency to run commodity operating systems with large workloads.[6, 15] The SimOS project[10] 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.[10, 20, 25] Herrod presents examples of simulation model accuracy/performance trade-offs and dynamic switching of timing models in SimOS.[10] Gibson et al. have validated the SimOS simulation models with performance measurements on real machines.[8]

6. CONCLUSIONS

We propose using complete system simulation as a platform for testing, debugging, and profiling of programs with quality of services requirements, such as multimedia applications. By adapting the GNU debugger to operate on applications in simulated Linux systems, we have created a deterministic and non-intrusive debugging and monitoring environment. The debugger is able to present application time flow, and allows collection of performance statistics from the hardware, operating system, and application level.

A symbolic debugger expects application level data, and it is necessary to translate the low-level data, provided by a simulator, to data useful to a debugger. The main contribution of the paper is virtual machine translation, a technique for performing this translation by traversing data structures in the operating system kernel.

We demonstrate how the temporal debugger can be used to test for display jitter in an MPEG video decoder. We also show how the debugger makes application and operating system internals visible during simulation, and how runtime statistics from different system levels can are correlated to deadline misses in the video decoder, thereby explaining the causes of display jitter.

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 could measure large amounts of data eagerly, without compromising accuracy.

ACKNOWLEDGMENTS

This work is funded by the national Swedish Real-Time Systems research initiative ARTES, supported by the Swedish Foundation for Strategic Research.

REFERENCES

1. Lars Albertsson. Simulation-based debugging of soft real-time applications. In *Proceedings of the Real-Time Application Symposium*. IEEE Computer Society, IEEE Computer Society Press, May 2001.
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. R. Cypher and E. Leu. Efficient race detection for message-passing programs with nonblocking sends and receives. In *Symposium on Parallel and Distributed Processing (SPDP '95)*, pages 534–543, Los Alamitos, Ca., USA, October 1995. IEEE Computer Society Press.
5. P. S. Dodd and C. V. Ravishankar. Monitoring and debugging distributed real-time programs. *Software Practice and Experience*, 22(10):863–877, October 1992.
6. J. K. Doyle and K. I. Mandelberg. A portable PDP-11 simulator. *Software Practice and Experience*, 14(11):1047–1059, November 1984.
7. Jason Gait. A debugger for concurrent programs. *Software Practice and Experience*, 15(6):539–554, June 1985.
8. Jeff Gibson, Robert Kunz, David Ofelt, Mark Horowitz, and John Hennessy. FLASH vs. (simulated) FLASH: Closing the simulation loop. In *Proceedings of the 9th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 49–58. ACM, November 2000.
9. Attila Gürsoy and Laxmikant V. Kalé. Simulating Message-Driven Programs. In *Proceedings of International Conference on Parallel Processing*, volume III, pages 223–230, August 1996.
10. Stephen Alan Herrod. *Using Complete Machine Simulation to Understand Computer System Behavior*. PhD thesis, Stanford University, February 1998.
11. Lauterbach. http://www.lauterbach.com.
12. Thomas J. LeBlanc and John M. Mellor-Crummey. Debugging parallel programs with instant replay. *IEEE Transactions on Computers*, C-36(4):471–482, April 1987.
13. Peter Magnusson and Bengt Werner. Efficient memory simulation in SimICS. In *Proceedings of the 28th Annual Simulation Symposium*, 1995.
14. Peter S. Magnusson. Efficient instruction cache simulation and execution profiling with a threaded-code interpreter. In *Proceedings of Winter Simulation Conference 97*, 1997.
15. 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.
16. D. C. Marinescu, J. E. Lumpp, Jr., T. L. Casavant, Siegel, and H. J. Models for monitoring and debugging tools for parallel and distributed software. *Journal of Parallel and Distributed Computing*, 9(2):171–184, June 1990.
17. Barton P. Miller, Mark D. Callaghan, Jonathan M. Cargille, Jeffrey K. Hollingsworth, R. Bruce Irvin, Karen L. Karavanic, Krishna Kunchithapadam, and Tia Newhall. The Paradyn parallel performance measurement tools. Technical report, Computer Sciences Department, University of Wisconsin – Madison, 1994.
18. Barton P. Miller, Cathryn Macrander, and Stuart Sechrest. A distributed programs monitor for Berkeley UNIX. *Software Practice and Experience*, 16(2):183–200, February 1986.
19. A. Mok and G. Liu. Efficient run-time monitoring of timing constraints. In *Proceedings of the Third IEEE Real-Time Technology and Applications Symposium (RTAS '97)*, pages 252–262, Washington - Brussels - Tokyo, June 1997. IEEE.
20. 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.
21. The Berkeley MPEG player, version 2.3. http://bmrc.berkeley.edu/frame/research/-mpeg/mpeg_play.html.
22. 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.
23. 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.
24. Bernhard Plattner. Real-time execution monitoring. *IEEE Transactions on Software Engineering*, SE-10(6):756–764, November 1984.

25. 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.
26. 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.
27. James D. Schoeffler. A real-time programming event monitor. *IEEE Transactions on Education*, 31(4):245–250, November 1988.
28. Kuo-Chung Tai, Richard H. Carver, and Evelyn E. Obaid. Debugging concurrent Ada programs by deterministic execution. *IEEE Transactions on Software Engineering*, 17(1):45–63, January 1991.
29. Ariel Tamches and Barton P. Miller. Using dynamic kernel instrumentation for kernel and application tuning. *The International Journal of High Performance Computing Applications*, 13(3):263–276, Fall 1999.
30. Henrik Thane and Hans Hansson. Using deterministic replay for debugging of distributed real-time system. In *Proceedings of the 12th euromicro conference on real-time systems*, pages 256–272, Stockholm, June 2000.
31. M. Timmerman and F. Gielen. The design of DARTS: A dynamic debugger for multiprocessor real-time applications. *IEEE Transactions on Nuclear Science*, 39(2):121–129, April 1992.
32. 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.
33. Jeffery J. P. Tsai, Kwang-Ya Fang, Horng-Yuan Chen, and Yao-Dong Bi. A noninterference monitoring and replay mechanism for real-time software testing and debugging. *IEEE Transactions on Software Engineering*, 16(8):897–916, August 1990.
34. Virtutech Simics v0.97/sun4u. http://www.simics.com.
35. Wind River. http://www.windriver.com.