Real-Time Debugging by Minimal Hardware Simulation [*]

Frank Mueller[1,2], David B. Whalley[1] and Marion Harmon[2]

[1] Dept. of Computer Science, Florida State University, Tallahassee, FL 32306-4019 (U.S.A.)
[2] Dept. of Computer and Information Systems, Florida A&M University, Tallahassee, FL 32307 (U.S.A.)
e-mail: whalley@cs.fsu.edu *phone: (904) 644-3506*

Abstract. This paper describes a debugging environment for real-time applications that supports the querying of the elapsed time at breakpoints. The environment employs hardware simulation at the level of processor cycles. The hardware simulation is limited only to the aspects relevant to processor cycle accounting and includes instruction caching, instruction frequency accounting, and instruction pipelining. This simulation is performed by program instrumentation to minimize the performance impact. Thus, the environment provides the means to debug a real-time application for an embedded system on a regular workstation in an efficient manner.

1 Introduction

Little attention has been paid to the specific needs of debugging real-time systems, although debugging is a central part of the software development cycle and may account for up to 50% of the development time. When debugging real-time applications, it is often necessary to relate the value of a variable to the elapsed time. *Time distortion*, due to the interference of a debugging tool, has to be minimized. *Deadline monitoring* should be supported to detect missed deadlines. *External events* need to be simulated and deadlines should be monitored during debugging.

In the absence of dedicated real-time debuggers, developers often fall back to hardware monitoring (*e.g.* logic analyzers) with limited capabilities or to hardware simulators, which degrades the application performance by about three orders of a magnitude. The simulation overhead of conventional hardware simulators is due to the necessity to capture the entire logic of a microprocessor and the interpretive nature of the simulation process.

We are proposing a framework for a debugger that permits debugging of real-time applications through non-interpretive, minimal hardware simulation. The environment addresses the issues of time distortion, deadline monitoring, and

[*] This work was supported in part by the Office of Naval Research under contract # N00014-94-1-0006

external events. The hardware simulation is limited to the aspects relevant to processor cycle accounting. These aspects include cache simulation, instruction frequency accounting, and processor pipeline simulation.

2 Related Work

Most commonly used debugging tools do not address the specific demands of real-time debugging. A few debugging tools specifically designed for real-time applications have been described in the literature, some of which focus on multiprocessor debugging. For example, Remedy provides the ability to suspend the execution on all processors when a breakpoint is reached [13]. The DCT tool allows practically non-intrusive monitoring but requires special hardware for bus access [3]. Both RED [6] and ART [18] provide monitoring and debugging facilities at the price of software instrumentation. RED dedicates a co-processor to collect trace data and send it to the host system. The instrumentation is removed for production code. In ART, a special reporting task sends trace data to a host system for further processing. The instrumentation code is a permanent part of the application. It will never be removed to prevent alteration of the timing. Debugging is limited to forced suspension and resumption of entities, viewing and alteration of variables, and monitoring of communication messages. DARTS provides remote debugging by replaying portions of a time-stamped program trace, which is produced during program execution [17]. The debugging is limited to a restricted set of events and only supports a subset of the functionality of common debuggers, *e.g.* excluding data queries. The high volume of trace information and the associated overhead of trace generation may also limit its application to programs with short execution times.

In the absence of real-time debuggers, hardware simulators are often used, which run about three orders of a magnitude slower than the actual application [14]. This performance overhead commonly limits the feasibility of extensive software testing and debugging.

3 The Debugging Environment

The debugging environment described in this paper differs from previous work by its close interaction with the back-end of an optimizing compiler VPO (very portable optimizer) [2]. An overview of the environment is given in Figure 1. The back-end of a compiler has been modified to emit control-flow and instruction information. This information is analyzed statically by a simulator. The simulator determines much of the caching and pipeline behavior statically, thereby reducing the overhead during program debugging. It then emits instrumentation code that is incorporated into the executable. The instrumented program can then be debugged by a regular debugger and the elapsed (virtual) execution time can be queried at any point. The virtual execution time is calculated on-the-fly on request by calculating the elapsed number of processor cycles based on the cache and pipelining simulation up to this point. The number of cycles can be inferred from a frequency counter for each basic block and simple state transitions simulating the remaining cache and pipelining behavior that could not be determined statically.

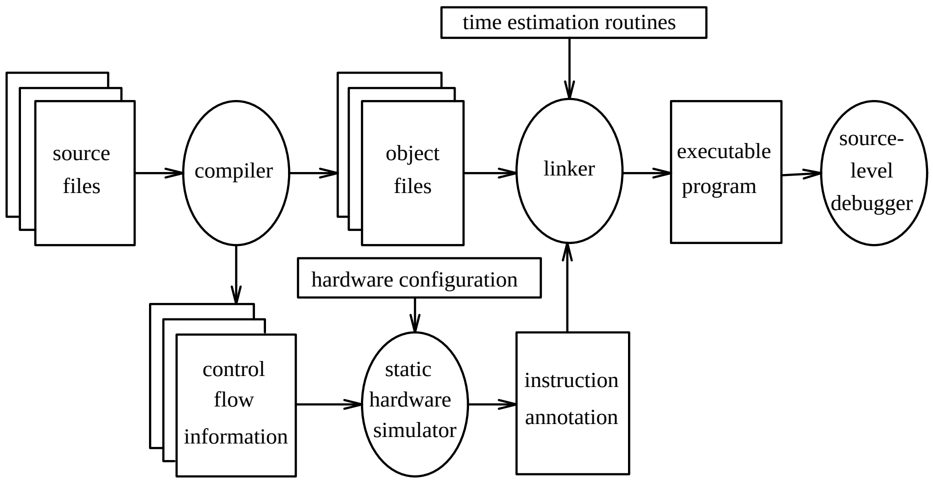

Fig. 1. Overview of the Debugging Environment

4 Instruction Cache Simulation

One part of tracking the elapsed number of cycles is the simulation of instruction caches. We are assuming an on-chip instruction cache of a configurable size with a direct-mapped architecture. Recent results have shown that direct-mapped caches have a faster access time for hits, which outweighs the benefit of a higher hit ratio in set-associative caches for large cache sizes [7]. As a result, current processor designs incorporate direct-mapped caches more frequently than set-associative caches.

The static hardware simulator includes a static cache simulation that predicts the caching behavior of instructions. Each instruction in a program is determined to be in one of the following categories:

– An *always hit* denotes an instruction that always is in cache when referenced, thus resulting in a cache hit.
– An *always miss* denotes an instruction that never is in cache when referenced, thus resulting in a cache miss.
– A *first miss* denotes an instruction that is not in cache on its first reference (cache miss) but is in cache on any subsequent reference (cache hit).

– A *conflict* denotes an instruction that may or may not be in cache on any reference, *i.e.* the caching behavior cannot be determined statically.

The categorization of each instruction is performed via data-flow analysis based on the control-flow information and the cache configuration. Once the static cache simulator has categorized all instructions, it can emit code to instrument the program. The instrumentation includes simple frequency counting to track always hits, always misses, and first misses. It also includes local state transitions similar to a finite state automaton to determine at execution time whether a conflict results in a cache hit or a cache miss. The details of static cache simulation can be found elsewhere [12, 10, 9].

Our prior work dealt with an ideal RISC processor that exhibits a single cycle overhead for an instruction execution on a cache hit and a constant overhead for a cache miss (estimated at 10 cycles). Our current work concentrates on applying the hardware simulation to an existing processor and enhance the simulation to include the effects of pipelining in a retargetable fashion.

The chosen target processor is the MicroSPARC I [16], which is commonly used in Sun SPARCclassic workstations. The processor has an on-chip direct-mapped instruction cache of 4kB with a line size of 8 instructions. Assuming the absence of pipeline stalls (discussed in the next section), the following timing behavior can be assumed for cache hits and misses. A cache hit will generally take one cycle. A cache miss on a cache line causes the first and second instructions of the line to become available after a 7 and 8 cycle delay, respectively, then a dead cycle occurs, and the process is repeated for the remaining instructions of the line (with a repeated dead cycle). A cache miss results in bringing an entire program line into cache. This process cannot be interrupted, even if the instructions in the program line do not coincide with the control flow at some point.

For example, consider a sequence of instructions in a program line containing a label L1 for instruction 3 and an unconditional jump instruction for instruction 5 as depicted in Figure 2. Assume that the current instruction transfers control to label L1 via a branch. Furthermore, assume that the depicted program line is not in cache. The reference to instruction 3 at L1 will cause a cache miss. Thus, instruction 3 will be available after a 7 cycle delay, instruction 4 one cycle afterwards, followed by a dead cycle. The jump instruction 5 and instruction 6 in the delay slot will be available after 10 and 11 cycles, respectively. At this point, control is transferred to L2 but the rest of the program line is brought into cache until cycle 17. The execution may continue in parallel at L2 if the instruction reference at L2 results in a cache hit. Conversely, a cache miss at L2 cannot be resolved until the previous program line is cached. Thus, if a miss occurs for the instruction at L2, a memory fetch is requested only after cycle 17. The instruction at L2 becomes then available for the processor at cycle 24.

The timing overhead of a cache miss depends on the taken control-flow path, as illustrated by the example. The following cases have to be distinguished:

– If instruction i resulting in a cache miss is followed by $8 - i$ sequential instructions without any transfer of control, then the miss overhead can

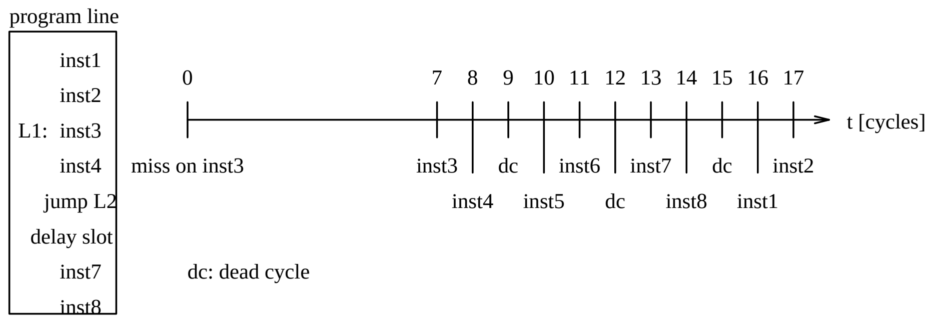

Fig. 2. Cache Miss

be associated with the instruction causing the miss. This overhead would be 10+(i-1) cycles since there is an initial delay of 7 cycles, 3 dead cycles occur during the line fetch, and the first $i-1$ instruction have to be fetched as well.

- If a cache miss on instruction i is followed by a transfer of control in instruction k to an instruction that already resides in cache, the overhead of the miss would be 7 cycles for the initial delay plus $\lfloor (k+2-i)/2 \rfloor$ dead cycles for memory fetches up to the branch delay slot of instruction $k+1$.
- If a cache miss on instruction i is followed by a transfer of control on instruction k to another instruction that does not reside in cache, the overhead of the miss would be $17-k+i$ cycles to cache the entire program line (including the initial delay, 3 dead cycles, and the instruction fetches past instruction k to fill the line).

As mentioned earlier, the cycle accounting for conflict instructions is performed via state transitions at execution time. These transitions have to reflect all possible paths in the control flow if a cache miss occurs. These details can be automatically determined by the static cache simulator based on a configuration file containing the cache configuration and a description of the memory access protocol on cache misses.

5 Pipeline Simulation

During cache hits, most instructions will account for one cycle execution overhead. For some instructions though, the execution takes multiple cycles. Since out-of-order execution is not permitted for the MicroSPARC I, the instruction pipeline will be stalled by such instructions. An instruction timing file can be used to distinguish instructions with varying execution overhead.

The main challenge of the simulation remains in capturing pipeline interlocks. These situations occur when an instruction in the pipeline stalls due to either a result that cannot yet be made available or to a *resource conflict* with another instruction in a later pipeline stage. For example, when a memory load into a register (assumed to be a cache hit) is followed by a reference to the same register, the referencing instruction will stall for one cycle until the value becomes available. This scenario is often referred to as a load-use interlock [5]. The MicroSPARC I interlocks the pipeline for a number of other instruction combinations [8].

The traditional approach to detect pipeline conflicts employs resource vectors and reservation tables [5]. The resource vector for an instruction describes the processor resources during each pipeline stage of the instruction processing. A reservation table is a sequence of resource vectors whose interstruction processing is interleaved for as many pipeline stages as possible. The goal on a RISC processor is to process one instruction per cycle, unless two consecutive instructions try to access the same processor resource during a cycle. The reservation table can be used to detect these resource conflicts. Unfortunately, this traditional approach requires an instruction analysis at the level of pipeline stages. If this analysis was performed statically, one would have to generate reservation tables for instruction sequences along the possible control-flow paths. This approach would possibly impose a considerable overhead.

Our design involves a different approach. Pattern matching (similar to peephole optimization) can be applied to instruction patterns that match specified pipeline interlocks. The current environment includes a modified back-end of a compiler. The compiler back-end could be further modified to recognize patterns of instructions that cause pipeline interlocks. This approach has the advantage that instruction patterns can be determined once and for all for a given architecture and stored in a file. The size of the peephole window is bounded by the maximum delay possible for a pipeline conflict. The window my span instructions along the control-flow paths of the program. Upon detecting a pipeline stall, a pattern of instructions would be annotated with the number of cycles associated with the pipeline stall. The stall cycles could then be reported to the static hardware simulator to include them in the cycle accounting. For example, the patterns to describe a load-use resource conflict would be as follows.

```
ld        *,reg[i]        |        ld        *,reg[i]
*         reg[i],*,*      |        *         *,reg[i],*
```

The patterns describe a load instruction for register i, followed by any instruction referencing register i, either as the first or second operand.

During static hardware simulation, the interaction between pipeline stalls and caching for straight-line code (within a basic block) can be resolved statically as long as the caching behavior can also be determined statically. Pipeline stalls reaching across basic blocks or involving dynamically dependent caching behavior will have to be incorporated into the dynamic simulation process. This can be accommodated by additional state transitions but will impact the dynamic execution overhead.

6 External Events

The described environment provides debugging capabilities via simulation. Thus, external events have to be simulated as well. The current design assumes that an event table of periodic and aperiodic requests is provided by the user. The corresponding event should be triggered by the simulation environment once the calculated cycle time equals or exceeds the specified event time.

7 Using the Environment

The output shown in Figure 3 depicts excerpts from a debugging session of a program performing fast fourier transformations within the environment using the unmodified source-level debugger *dbx* [15].

```
(dbx) stop at 123 if elapsed_cycles() > 4000000 /* set cond breakpoint */
(dbx) display elapsed_cycles()    /* show return value on breakpoint   */
(dbx) run                         /* start program execution           */
stopped in main at line 123       /* exec stopped on cond breakpoint   */
  123       four(tdata,nn,isign);
elapsed_cycles() = 4015629
(dbx) cont
K = 100   Time = 0.290000 Seconds /* program output                    */
elapsed cycles() = 4095351        /* total number of executed cycles   */
execution completed, exit code is 1
program exited with 1
```

Fig. 3. Annotated Excerpts from a Sample Debugging Session

First, a conditional breakpoint is set on a function call that checks for a deadline miss after 4 million cycles. The display command ensures that the elapsed time estimated in cycles is displayed at each breakpoint. The execution is stopped on line 123 after over 4 million cycles, which indicates that the task could not finish within the given deadline. This conditional breakpoint was placed on a repeatedly executed function call to periodically check this condition. The deadline miss can be narrowed down to an even smaller code portion by setting further conditional breakpoints. At program termination, the final number of processor cycles is displayed.

The timing information can be used during debugging to locate portions of code that consume most of the execution time. This information can be used to hand-tune programs or redesign algorithms.

When a set of real-time tasks is debugged, one can identify the task that is missing a deadline, either by checking the elapsed time or by setting a conditional breakpoint dependent on the elapsed time. The schedule can then be fixed in various ways. One can tune the task that missed the deadline. Alternatively, one can tune any of the preceding tasks if this results in a feasible schedule. The latter may be a useful approach when a task overruns its estimated execution time without violating a deadline, thereby causing subsequent tasks to miss their deadlines. The debugger will help to find the culprit in such situations. Another option would be to redesign the task set and the schedule, for example by further partitioning of the tasks [4].

8 Current Status and Future Work

We have implemented the debugging environment for an ideal processor with a cache hit time of one cycle and a cache miss time of ten cycles [11]. We are enhancing this implementation according to the design described in this paper to reflect the specifics of the MicroSPARC I instruction cache and to take pipelining into account. The correctness of resulting virtual time accounting during debugging will be verified by comparison with the observed program timing on a stand-alone VME board with a MicroSPARC I under a non-preemptive embedded real-time operating system [1]. The operating system is designed to exhibit predictable execution behavior and to provide more accurate timing than regular operating systems, such as UNIX.

At this point, an instrumented, optimized program runs at about 1-4 times the speed of the uninstrumented, unoptimized version that is typically used for debugging [11]. The number varies according to the ratio of program size and cache size. The additional work due to minimal dynamic pipeline simulation is expected to increase this overhead. Yet, the overhead should still be well below that of conventional hardware simulators.

We are also working on the design of a simulator for data caching. Under certain restrictions (*e.g.*, absence of pointers and heap allocation) many addresses of data references can be calculated statically. This includes global data, local data allocated on the stack (in the absence of recursion), and certain patterns of array references. The effect of data caching should be included into the hardware simulation in a manner similar to the handling of instruction caching.

9 Conclusion

We believe that our work of a minimal hardware simulator for the purpose of cycle accounting is unprecedented. The simulation framework is being designed and implemented for the MicroSPARC I architecture but should be retargetable for other RISC architectures. The application to the debugging of real-time programs provides the means to test an embedded system on a regular workstation using simulation. This facilitates the process of debugging for the user. It supports queries for the elapsed (virtual) time, which can be used to relate debugging output to time information. Time distortion during debugging is minimized and deadlines can be monitored. Thus, a deadline miss can be located and the corresponding task may be tuned by determining where most of the time is spent. Alternatively, the task schedule can be redesigned to meet the deadline requirements. The minimal hardware simulation accounts for the effects of instruction caches and pipeline stalls. The effect of data caches is subject to future research.

References

1. T. P. Baker, F. Mueller, and Viresh Rustagi. Experience with a prototype of the POSIX "minimal realtime system profile". In *IEEE Workshop on Real-Time Operating Systems and Software*, pages 12–16, 1994.
2. M. E. Benitez and J. W. Davidson. A portable global optimizer and linker. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 329–338, June 1988.
3. D. Bhatt, A. Ghonami, and R. Ramanujan. An instrumented testbed for real-time distributed systems development. In *IEEE Symposium on Real-Time Systems*, pages 241–250, December 1987.
4. R. Gerber and S. Hong. Semantics-based compiler transformations for enhanced schedulability. In *IEEE Symposium on Real-Time Systems*, pages 232–242, December 1993.
5. J. Hennessy and D. Patterson. *Computer Architecture: A Quantitative Approach.* Morgan Kaufmann, 1990.
6. C. R. Hill. A real-time microprocessor debugging technique. In *ACM SIGSOFT/SIGPLAN Software Engineering Symposium on High-Level Debugging*, pages 145–148, 1983.
7. M. Hill. A case for direct-mapped caches. *IEEE Computer*, 21(11):25–40, December 1988.
8. Adolf Leung. Personal communications. Sun Microsystems (Engineering), February 1994.
9. F. Mueller. *Static Cache Simulation and its Applications.* PhD thesis, Dept. of CS, Florida State University, July 1994.
10. F. Mueller and D. B. Whalley. Efficient on-the-fly analysis of program behavior and static cache simulation. In *Static Analysis Symposium*, September 1994.
11. F. Mueller and D. B. Whalley. On debugging real-time applications. In *ACM SIGPLAN Workshop on Language, Compiler, and Tool Support for Real-Time Systems*, June 1994.
12. F. Mueller, D. B. Whalley, and M. Harmon. Predicting instruction cache behavior. In *ACM SIGPLAN Workshop on Language, Compiler, and Tool Support for Real-Time Systems*, June 1994.
13. P. Rowe and B. Pagurek. Remedy: A real-time, multiprocessor, system level debugger. In *IEEE Symposium on Real-Time Systems*, pages 230–239, December 1987.
14. K. So, F. Darema, D. A. George, V. A. Norton, and G. F. Pfister. PSIMUL – a system for parallel execution of parallel programs. *Performance Evaluation of Supercomputers*, pages 187–213, 1988.
15. Sun Microsystems, Inc. *Programmer's Language Guide*, March 1990. Part No. 800-3844-10.
16. Texas Instruments. *TMS390S10 Integrated SPARC Processor*, February 1993.
17. M. Timmerman, F. Gielen, and P. Lambix. A knowledge-based approach for the debugging of real-time multiprocessor systems. In *IEEE Workshop on Real-Time Applications*, pages 23–28, 1993.
18. H. Tokuda, M. Kotera, and C. W. Mercer. A real-time monitor for a distributed real-time operating system. In *ACM/ONR Workshop on Parallel and Distributed Debugging*, pages 68–77, 1988.