On Debugging Real-Time Applications *

Frank Mueller and David B. Whalley
Dept. of Computer Science
Florida State University
Tallahassee, FL 32306-4019
e-mail: whalley@cs.fsu.edu phone: (904) 644-3506

Abstract

Debugging is an integral part of the software development cycle which can account for up to 50% of the development time of an application. This paper discusses some of the challenges specific to real-time debugging. It explains how developing real-time applications can be supported by an environment which addresses the issues of time deadline monitoring and distortion due to the interference of debugging. The current implementation of this environment provides the elapsed time during debugging on request at breakpoints. This time information corresponds to the elapsed execution time since program initiation. Delays due to the interference of the debugger, for example input delays at breakpoints, are excluded from the time estimates. The environment includes a modified compiler and a static cache simulator which together produce instrumented programs for the purpose of debugging. The instrumented program supports source-level debugging of optimized code and an efficient cache simulation to provide timing information at execution time. The overhead in execution time of an instrumented program is only approximately 1 to 4 times slower than the corresponding unoptimized program. Conventional hardware simulators could alternatively be used to obtain the same information but would run much slower. The environment facilitates the debugging of real-time applications. It allows the monitoring of deadlines, helps to locate the first task which misses a deadline, and supports the search for code portions which account for most of the execution time. This facilitates hand-tuning of selected tasks to make a schedule feasible.

1 Introduction

The issue of debugging real-time applications has received little attention in the past. Yet, in the process of building real-time applications, debugging is commonly performed just as in the development of non-real-time software and may account for up to 50% of the development time [16]. The debugging tools used for real-time applications are often ordinary debuggers which do not cater to specific needs of real-time systems listed below.

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

Time distortion: The notion of real time is central to real-time applications. Hardware timers are commonly used to inquire timing information during program execution to synchronize the application with periodic events. Yet, during debugging the notion of real time should be replaced by the notion of virtual time to compensate for time distortion due to the interference of debugging. External events have to be simulated based on the elapsed (virtual) time of tasks. Thus, values of variables can be related to the elapsed time which is essential for debugging real-time applications.

Deadline monitoring: During the implementation phase, deadlines may not always be met. A real-time debugger should display the elapsed time for a task on request. This would facilitate finding the first task which fails to meet a deadline. It could also be used to inquire at which point during the execution a deadline was missed. Furthermore, the elapsed time may help in tuning tasks by locating where most of the execution time is spent.

Uniprocessor vs. multiprocessor: Multiprocessor applications are sometimes simulated on uniprocessors during debugging. In this case, a virtual clock has to be kept for each processor which is shared by a set of tasks running on this processor.

This work concentrates on time distortion and deadline monitoring.

A debugging environment has been developed which permits the user to query the elapsed time. This time corresponds to the virtual time from program initiation to the current breakpoint excluding debugging overhead and is calculated on demand. In contrast, time queries in current debuggers correspond to the wall-clock time and include the delay of user input at breakpoints as well as the debugger trap overhead.

The environment can be used to debug a set of real-time tasks which do not meet their deadline. It facilitates the analyses of the tasks and helps to find out where a task spends most of it's execution time or which portion of a task completed execution before missing the deadline. This knowledge can then be utilized to fine-tune the task which is missing its deadlines or any of the previous tasks in the schedule. Thus, this debugging environment assists the process of designing a feasible schedule in a step-by-step fashion.

The elapsed time of a task is estimated based on the caching behavior of the task. The caching information is updated during execution and provides an estimate of the number of elapsed processor cycles.

The dynamic simulation of cache performance necessitates the tracking of events and their ordering to determine a cache miss *vs.* a cache hit. It can be quite a challenge to perform order-dependent events efficiently. This paper describes the design and implementation of such an environment within the framework of a compiler, a static cache simulator [12], and an arbitrary source-level debugger. The compiler translates a program into assembly code and provides control-flow information to the static cache simulator. The static cache simulator analyzes the caching behavior of the program and produces instrumentation code which is merged into the assembly code. The source program corresponding to the resulting assembly code can then be debugged and the elapsed time can be requested at breakpoints.

The elapsed time is calculated based on the cache simulation up to the current breakpoint, i.e. the number of cache hits and misses are multiplied by the access time for hits and misses respectively. This provides an estimate of the executed numbers of processor cycles corresponding to the elapsed (virtual) time since program initiation.

It may be argued that the virtual execution time can be provided by the operating system. Notice though that the debugging process affects the execution of the real-time task, *e.g.* the caching behavior. The cache simulation discussed here estimates the timing of the task in an actual real-time environment disregarding the interference of debugging.

Another problem is posed by the debugging of optimized code. Conventional compilers only support source-level debugging of unoptimized code. Clearly, unoptimized code causes further time distortion which cannot be accepted for real-time systems. Thus, a compiler has been modified to support source-level debugging of optimized code with certain restrictions, which are discussed later in the paper.

2 Related Work

Conventional debugging tools, whether at the assembly-level or at the source-level, do not address the specific demands of real-time debugging. The amount of work in the area of real-time debugging has been limited with a few exceptions.

The Remedy debugging tool [14] addresses the customization of the debugging interface for real-time purposes and synchronizes on breakpoints by suspending the execution on all processors. DCT [5] is a tool that allows practically non-intrusive monitoring but requires special hardware for bus access. Both RED [8] and ART [17] 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 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.

The DARTS system [16] approaches the debugging problem in two stages. It first generates a program trace and then allows for debugging based on the trace data which is time-stamped to address the time distortion problem. The debugging is limited to a restricted set of events which is extracted from the control flow. This tool 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. None of the systems make use of the compiler to enhance the debugging process.

In the absence of real-time debuggers, hardware simulators are often used which run considerably slower than the actual application and, consequently, allow only selective and not very extensive testing. In addition, changing the simulated architecture of hardware simulators is typically complicated.

3 A Real-Time Debugging Environment

The current work concentrates on monitoring deadlines based on the cache analysis of a task and the corresponding estimate of the elapsed (virtual) execution time. This facility can be used in conjunction with a conventional debugger. The debugger does not need to be modified.

The cache simulation overhead at run time is reduced by analyzing the cache behavior statically. A large number of cache hits and misses can be determined prior to execution time by considering the control flow of each function and the call graph of the program. The remaining references are simulated at execution time.

Figure 1 depicts an overview of the environment. A set of source files of a program are translated by a compiler. The compiler generates object code and passes information about the control flow of each source file to the static cache simulator. The static cache simulator performs the task of determining which instruction references can be predicted prior to execution time. It constructs the call graph of the program and the control-flow graph of each function based on the information provided by the compiler. The cache behavior is then simulated for a given cache configuration. Furthermore, the static simulator produces instruction annotations and passes them to the linker which modifies the object code according to the annotations and cre-

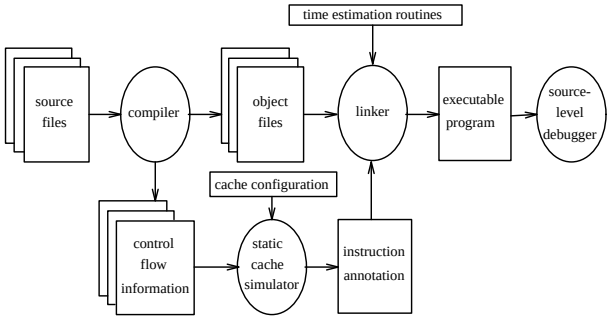

Figure 1: Overview of the Environment

ates an executable program including library routines for the time estimation. The executable may then be run within a source-level debugger. The elapsed time can be inquired at any breakpoint by calling the library routine which estimates the number of processor cycles executed based on the number of cache hits and misses up to that point.

3.1 Static Cache Simulation

The task of static cache simulation is to determine whether each instruction reference will result in a cache hit or miss during program execution.[1] This is accomplished by analyzing the call graph and the control flow for each function. Since it is not always possible to determine if a reference is a hit or miss, instructions are classified to be in the categories of *always-hit, always-miss, first-miss,* or *conflict.* If an instruction is always (never) in cache, then it is denoted as an always-hit (always-miss). If an access to an instruction results in a miss on the first access and in hits for any subsequent accesses, then it is classified as a first-miss. If an access to a program line results in either a hit or a miss depending on the flow of control, then it is referred to as a conflict.

The categorization of instruction references will be discussed briefly. A more formal description and the corresponding algorithms can be found elsewhere [12, 11]. An abstract cache state of a basic block in the control-flow graph denotes the subset of program lines which can potentially be cached before executing the block, *i.e.* there exists an execution path for each program line of the subset such that the program line is cached at the beginning of the block.

A program line reference results in an always-miss if the program line is not in the abstract cache state. An always hit occurs if the line is in the abstract cache state and no other program line mapping into the same cache line is in the abstract cache state. A first miss occurs if the line is in the abstract cache state, and all other lines in the abstract cache state (mapping into the same cache line) cannot be reached anymore through the control flow. The remaining instructions are classified as conflicts.

The calculation of the abstract cache states is performed by an iterative algorithm similar to the data-flow analysis performed in optimizing compilers. The categorization amounts to a traversal of the program lines, *i.e.* a traversal of the program lines for each basic block within the combination of the call graph and the control-flow graphs of each function.

Based on the category of a program line, the static simulator emits instruction annotations, placed at the beginning of basic blocks, which provide frequency counts and simulate the "conflicts" using simple state transitions at execution time. The additional code affects the run time of the program but not the calculation of the elapsed time since the latter is based on the cache simulation performed on the original, uninstrumented program.

3.2 Querying the Elapsed Time

The elapsed execution time can be queried at any breakpoint while debugging a program without modifying the debugger. The time is calculated based on the cache analysis. The number of cache hits and misses can be calculated on the fly from the frequency counters. The elapsed time is then calculated as

$$t_{elapsed} = hits * hit\_penalty + misses * miss\_penalty \text{ [cycles]}$$

where the hit penalty is typically one cycle while the miss penalty is ten cycles [9] or even more, depending on the clock rate and the access time of main memory. This time estimate can be converted into seconds by multiplying it by the cycle time. The calculation of hits and misses takes only a short time and can therefore be repeated whenever the program stops at a breakpoint without much overhead. The code performing the calculation is hidden in linked-in library code.

The debugged program has been compiled with full optimizations to avoid time distortion. The compiler was modified to emit debugging information for unoptimized code *as well as* optimized code. Emitting accurate debugging information for optimized code is a non-trivial task and subject to ongoing research [1, 6, 10]. Contrary to debugging unoptimized code, debugging optimized code typically restricts the scope of breakpoints and the displaying of data structures. In the debugging environment described in this paper, a breakpoint set on a source line is approximated as a breakpoint at the beginning of the corresponding basic block when code is optimized. In addition, the value of variables assigned to a register will only be displayed if all live ranges are assigned to the same register [2]. Register-mapped values may still be inconsistent at times due to global optimizations, such as common subexpression elimination which is a common problem when debugging optimized code.

The fact that optimized code is executed during debugging speeds up the execution over conventional de-

[1] The current implementation is limited to direct-mapped instruction caches. The work is being extended to include set-associative caches and to also handle data caches.

```
> dbx fft
Reading symbolic information...
Read 396 symbols
(dbx) stop at 43                                /* set breakpoint on line 43                    */
(2) stop at 43
(dbx) stop at 114                               /* set breakpoint on line 114                   */
(3) stop at 114
(dbx) stop at 123 if elapsed_cycles() > 4000000 /* set cond. breakpoint                         */
(4) stop  at 123 if elapsed_cycles() > 4000000
(dbx) display elapsed_cycles()                  /* display function return value on breakpoint */
elapsed_cycles() = 0                            /* 0 cycles since program has not started       */
(dbx) run                                       /* start program execution                      */
Running: fft
stopped in main at line 114                     /* execution stopped on first breakpoint        */
  114     printf("Objective: measure exec. time of 128 FFT.\n");
elapsed_cycles() = 22                           /* 22 cycles executed before first breakpoint   */
(dbx) cont                                      /* resume execution until next breakpoint       */
Objective: measure exec. time of 128 FFT.       /* program output                               */
stopped in four at line 43
   43     mmax=2;
elapsed_cycles() = 29413
(dbx) next                                      /* single step to next source line statement    */
stopped in four at line 44
   44     while(n>mmax) {
elapsed_cycles() = 29428
(dbx) print mmax                                /* print out value of variable                  */
mmax = 2
(dbx) cont
stopped in four at line 43
   43     mmax=2;
elapsed_cycles() = 70547
(dbx) clear                                     /* clear current breakpoint (line 43)           */
(dbx) next
stopped in four at line 44
   44     while(n>mmax) {
elapsed_cycles() = 70553
(dbx) cont
stopped in main at line 123                     /* execution stopped on conditional breakpoint */
  123       four(tdata,nn,isign);
elapsed_cycles() = 4015629
(dbx) clear
(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
(dbx) quit
```

Figure 2: Annotated Sample Debugging Session

bugging of unoptimized code. The cache simulation, on the other hand, adds to the execution time. A quantitative analysis of the effect of these issues will be given in the measurement section.

4 Applications

The output shown in Figure 2 illustrates a short debugging session of a program performing fast fourier transformations within the environment using the unmodified source-level debugger *dbx* [15].

First, a few breakpoints are set including a conditional breakpoint on a subroutine call which checks on a deadline miss after 4 million cycles. The display command ensures that the elapsed time estimated in cycles is displayed at each breakpoint as seen later during execution. The value of the variable `mmax` can be printed although it has been assigned to a register due to code optimization. Notice that the breakpoint on line 43 is reached twice. The difference in the number of cycles between line 43 and line 44 is 15 cycles during the first iteration but only 6 cycles during the second iteration. A closer investigation reveals that during the first iteration, one of the six instructions in the basic block references a program line which results in a compulsory miss estimated as 10 cycles. On the second iteration, the same reference results in a hit due to temporal locality estimated as 1 cycle. 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 subroutine 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 which 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

Name	Description	Size [bytes]	unopt. code	opt. code with time estimates 1kB	2kB	4kB	8kB
cachesim	Cache Simulator	8,452	1.1	2.0	1.4	1.3	1.2
cb	C Program Beautifier	4,968	1.4	6.8	5.8	3.4	2.6
compact	Huffman Code Compression	5,912	2.1	10.3	7.8	6.0	2.7
copt	Rule-Driven Peephole Optimizer	4,144	1.4	2.5	1.7	1.4	1.4
dhrystone	Integer Benchmark	1,912	1.6	2.7	1.6	1.6	1.6
fft	Fast Fourier Transform	1,968	1.3	1.4	1.2	1.2	1.2
genreport	Detailed Execution Report Generator	17,716	1.4	3.6	2.5	2.4	2.3
mincost	VLSI Circuit Partitioning	4,492	1.6	5.0	3.2	2.2	1.8
sched	Instruction Scheduler	8,352	2.1	22.9	14.6	8.3	4.1
sdiff	Side-by-side Differences between Files	7,288	4.1	27.1	8.1	4.0	3.0
whetstone	Floating point benchmark	4,812	1.2	2.0	2.0	1.5	1.2
average		6,365	1.8	7.8	4.5	3.0	2.1

Table 1: Performance Overhead

identify the task which 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 which 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 [7].

5 Measurements

The environment discussed above was implemented for the SPARC architecture. It includes a modified compiler back-end of VPO (very portable optimizer) [4], the static simulator for direct-mapped caches [12], and the regular system linker and source level debugger DBX under SunOS 4.1.3. Calling a library routine to query the elapsed time takes a negligible amount of time in the order of one millisecond. Thus, this section focuses on measuring the overhead of cache simulation during program execution. The correctness of the instruction cache simulation was verified by comparison with a traditional trace-driven cache simulator. The execution time was measured for a number of user programs, benchmarks, and UNIX utilities using the built-in timer of the operating system to determine the overhead of cache simulation at run time. Table 1 shows programs of varying program size (column 3), the overhead of unoptimized code (column 4), and the support of virtual timing information through dynamic cache simulation (column 5-8) as a factor of the execution time of optimized code for cache sizes of 1kB, 2KB, 4kB, and 8kB.

On average, unoptimized programs ran 1.8 times slower than their optimized version. Running the optimized program and performing cache simulation to provide virtual timing information was on average 2.1 to 7.8 times slower than executing optimized code.[2] In other words, the optimized code with cache simulation was roughly 1 to 4 times slower than the unoptimized code typically used for program debugging.

The cache size influences the overhead factor considerably which can be explained as follows: For small cache sizes, programs do not fit into cache and capacity misses occur frequently which requires the dynamic overhead of simulating program lines classified as "conflicts". For larger cache sizes, a larger portion of the program fits into cache reducing capacity misses and thereby reducing the number of "conflicts". Once the entire program fits into cache, no "conflicts" need to be simulated. Rather, frequency counters are sufficient to simulate the cache behavior. This reduces the overhead considerably.

6 Future Work

The work could be extended to take external events into account. The user will be required to specify the occurrence of events in a time table. The events are then simulated by the debugging environment based on the elapsed time. At program termination, the monitored activities (*e.g.* completion time, deadline) could be summarized in a table.

The interaction of the debugging environment with a compiler provides the means to introduce a compilation `pragma zero_time` which excludes a code portion from virtual time accounting. This can be used to insert conditionally compiled debugging code which does not affect the overall timing.

The work is also being extended to include the impact of data caching and pipeline stalls to improve the accuracy of the time estimates. Another application of this work is to design a detailed profiler which facilitates the search for code portions which consume most of the execution time.

[2] Notice that the static cache simulation makes this approach feasible. Traditional trace-driven cache simulation is reported to slow down the execution time by one to three orders of a magnitude [18].

Furthermore, this environment could also be used for multi-threaded applications where a thread corresponds to a task. The application could be designed for a non-preemptive embedded system[3] but may be debugged on a regular workstation using the environment to simulate the embedded system efficiently.

7 Conclusion

This work discusses some challenges of real-time debugging which have not yet been addressed adequately. A debugging environment is proposed which addresses the problem of time distortion during debugging. In this environment, the notion of real time is replaced by virtual time based on the estimated number of elapsed processor cycles. The first implementation step has been completed and provides the elapsed time based on instruction cache simulation at any breakpoint during debugging. This time information can be used for deadline monitoring, identifying the task which first misses a deadline, and locating time-consuming code portions to support hand-tuning of tasks until a schedule becomes feasible. To provide this timing information, the execution speed of the application during debugging is 1-4 times slower in average than the speed of the corresponding unoptimized application. In contrast, conventional hardware simulators may provide the same information but are less portable and much slower. The environment facilitates the debugging of real-time programs when timing-related problems occur which have to be reproduced during debugging.

References

[1] A. Adl-Tabatabai and Thomas Gross. Detection and recovery of endangered variables caused by instruction scheduling. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 13–25, June 1993.

[2] A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers – Principles, Techniques, and Tools*. Addison-Wesley, 1986.

[3] 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.

[4] 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.

[5] 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.

[6] G. Brooks, G. Hansen, and S. Simmons. A new approach to debugging optimized code. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 1–11, June 1992.

[7] R. Gerber and S. Hong. Semantics-based compiler transformations for enhanced schedulability. In *IEEE Symposium on Real-Time Systems*, pages 232–242, December 1993.

[8] C. R. Hill. A real-time microprocessor debugging technique. In *ACM SIGSOFT/SIGPLAN Software Engineering Symposium on High-Level Debugging*, pages 145–148, 1983.

[9] M. Hill. A case for direct-mapped caches. *IEEE Computer*, 21(11):25–40, December 1988.

[10] U. Hoelzle, C. Chambers, and D. Ungar. Debugging optimized code with dynamic deoptimization. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 32–43, June 1992.

[11] F. Mueller and D. B. Whalley. Efficient on-the-fly analysis of program behavior and static cache simulation. In *Static Analysis Symposium*, September 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] Frank Mueller. A library implementation of POSIX threads under UNIX. In *Proceedings of the USENIX Conference*, pages 29–41, January 1993.

[14] 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.

[15] Sun Microsystems, Inc. *Programmer's Language Guide*, March 1990. Part No. 800-3844-10.

[16] 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.

[17] 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.

[18] D. B. Whalley. Fast instruction cache performance evaluation using compile-time analysis. In *SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, pages 13–22, June 1992.

[3] A multi-threaded real-time kernel has been designed for such an embedded system based on a SPARC VME bus board [3, 13].