Predictable Timing Behavior by using Compiler Controlled Operation

Vesa Hirvisalo and Sami Kiminki
Helsinki University of Technology
Laboratory of Information Processing Science
P.O. Box 5400, FIN-02015 HUT, Finland
Vesa.Hirvisalo@cs.hut.fi, Sami.Kiminki@iki.fi

Abstract

We propose coordinated use of compiler techniques to improve predictability of timing behavior of hard real-time systems, and thus, to tighten their worst-case execution times. We aim at a generic methodology of compiler optimizations that replace the use of unpredictable hardware and operating system features by the use of more predictable features. We call the approach compiler controlled operation, because it is based on using compilers to control operations that are traditionally controlled by hardware or operating systems. As an example of the approach, we overview our work in progress on a small experimental system.

1 Introduction

This paper discusses how compiler techniques can be used to build software systems having predictable timing behavior. Predictability of timing behavior is needed to guarantee temporal correctness of hard real-time systems.

There are several trends that make giving such guarantees increasingly difficult. New applications based on the use of real-time software components are rapidly emerging. To cope with the complexity of the software systems, high-level software development tools are being adapted. Many applications require high performance in addition to predictable timing behavior. Because of the increasing performance requirements and the use of generic purpose hardware components to limit production costs, hardware for real-time systems is becoming complex. These factors increase dynamism of the systems and make their timing behavior hard to predict.

Timing guarantees are given by schedulability analysis that is typically divided into intra-task analysis and inter-task analysis. Intra-task analysis resolves worst-case execution times (WCET) for tasks. Improving execution speed can be useless, even if the improvements yield tight worst-case execution times, unless such tight worst-case execution times can be guaranteed. Inter-task analysis determines whether the tasks can be guaranteed to be scheduled so that they meet their deadlines. Similarly, techniques for faster average-case execution can be useless, or even harmful, when inter-task real-time analysis is considered.

We concentrate on unpredictability caused by modern hardware and typical operating system features. Such features include – but are not limited to – cache memories, interrupts, and context switches. There are two things common to features considered by us. First, they make timing analysis difficult by using features that are not apparent from the application code. Second, more predictable techniques exist for implementing them in special cases. Using such alternative techniques yield tighter worst-case execution times.

Our generic solution is to use compiler techniques to transform the software to use more predictable implementation techniques, when such transformations are possible. We call the approach *compiler controlled operation*, because it is based on using compilers to control operations that are traditionally controlled by hardware or operating systems. Compiler controlled operation is based on static program analysis. Therefore, it is closely related to the use of WCET analysis methods based on static program analysis. In addition to improving timing behavior, compiler controlled operation can be used to other purposes, e.g., energy saving.

The structure of the rest of this paper is the following. In Section 2, we discuss compiler controlled operation in general. In Section 3, we consider briefly some possible realizations for compiler controlled operation. In Section 4, we overview our work in progress on an experimental system that concentrates on using fast on-chip RAM memory (often called scratchpad memory) to implement the operation of classic cache hardware and compiler-time scheduling to partially implement a process abstraction. The last section draws some conclusions and discusses some related work.

2 Compiler controlled operation

Programs are abstract entities, but they are executed in some concrete execution environment that usually has several specific features supporting the execution of programs. Typically, the execution environment includes the various features of the operating system and the whole underlying hardware (including both on-chip and off-chip features). The environment significantly affects the timing behavior of a program.

Compiler controlled operation means that some operations in the execution environment of an application are controlled by the compiler that compiles the application. This requires cooperation between the compiler and the execution environment.

The main task of operating systems is to implement process abstraction. This includes managing processes, scheduling processes, and providing processes with inter-process communication, synchronization and protection. Especially scheduling combined with synchronization causes problems in timing prediction, because of the run-time decisions made by the operating system.

Modern hardware includes speculative features to increase speed and supporting features to increase flexibility. The use of such features often make timing prediction hard. The typical speculative hardware feature that causes problems in timing prediction is the cache memory. The combined use of parallel processes and cache memories can cause severe problems in predictability [11].

Traditionally, the programmers of a system are responsible for using predictable techniques instead of the generic ones (e.g., application-controlled memory management). There are situations, where this can be automated. The use of predictable techniques instead of the generic ones can be implemented as optimizations done by a compiler. Such compiler optimizations consider the whole execution environment instead of the instruction set architecture of the processor. As in traditional compilers, several optimizations can be used in a coordinated way, and they can be used to promote predictability (e.g., tight WCETs) instead of average speed.

In addition to the transformations, analysis required for the transformations can be done by compilers, as well as timing verification. As for many WCET tools, user support may be needed to guide the compiler. However, some tasks can be fully automatized, e.g., the use of a scratchpad memory as a cache.

Considered from the point of view of the application developer, transparency is important regardless whether the analysis can be made fully automatic. Independent of the implementation, the same way of coding and the same interfaces should be used.

3 Various possibilities for compiler controlled operation

Traditionally, compilers are aware of the execution environment only partially. They know the target hardware architecture including target processor and memory layout, but often the operating system semantics and semantics of other applications in the system are completely unknown. Operating system interfacing is typically provided by libraries. Isolation is even a goal in many operating system designs.

In closed systems, there exists less reasons for such isolation, as often all application and operating system semantics are completely available at design time. Providing such information for the compiler reveals many exciting possibilities for optimization. We give some examples on how compiler might exploit extended information on execution environment.

A compiler can automate scratchpad memory usage and allocation. As scratchpad operations are explicit in the program code, they pose no inherent unpredictability. Instead, program code using scratchpad operations can be analyzed with existing WCET tools. Thus, the problems of cache behavior unpredictability can be avoided (see [10] as an introduction to such scratchpad usage).

When timing information of tasks is provided, the compiler can perform scheduling optimizations. For example, compiler might statically schedule and join multiple periodic tasks of same frequency or integer multiple of some base frequency into one task [1]. Further, if the information of hardware interrupt rate boundaries is known, interrupt handling may be transformed to polling by the compiler. Polling and branch prediction may improve WCET guarantees in some cases.

If the compiler is aware of operating system semantics, optimizations to inter-process communications can be performed. Semaphore synchronization may be transformed to statical task rescheduling in some cases, as well as many remote procedure call patterns are transformable to simpler inter-process procedure calls. Such transformations simplify scheduling analysis, and are thus susceptible to promote WCET guarantees.

The transformations need not to concern only application code, but may also be directed to the operating system. A typical task performed manually is the tuning of operating system features, such as the sizes of various buffers, and implementation techniques of features such as inter-process communication primitives. The operating system feature-selection and tuning can be seen as global optimization problem for the compiler.

Furthermore, when high-volume systems are of concern, the compiler could even tune the hardware execution environment. As with the operating system, tuning of hard-

ware execution environment can also be seen as global optimization problem. For example, the amount of scratchpad memory in the system, number of registers, special instructions (such as division, multiplicate-and-accumulate, and scratchpad transfer instructions), can all be seen as parameters to a global optimization problem. Today, hardware features are selected and tuned manually, and the choices may have great effect to the system performance.

4 Work in progress

The PAD system consist of two components: a compiler, padCC, and an operating system, padOS. It is a small experimental system that is designed for the study of platforms for closed embedded control systems that have periodic hard real-time timing requirements. The goal of the system is to promote predictable timing and low energy consumption in high performance applications.

padOS is a small real-time operating system. It supports multitasking, but has no memory protection, because it is designed for closed applications. It implements EDF scheduling and prioritized interrupt handling. padOS supports background tasks and inter-process communication.

padCC is a C compiler. In addition to optimization based on the instruction set architecture of the target processor, padCC supports optimizations based on the execution environment. padCC uses scratchpad memory hardware to implement the operation of classic cache hardware. Instead of using associative memory that is able to handle misses, padCC generates code that uses scratchpad memory instead of of main memory to store and access data. The analysis and code generation closely resembles the register allocation techniques used in optimizing compilers. Thus, all memory transfers are statically known.

padCC does partial compiler-time scheduling. A C program with operating system primitives is considered as a concurrent program that is compiled into a sequential program when possible (see [4] for an introduction to such techniques). The static scheduling is coordinated with the scratchpad memory allocation. Inter-task scratchpad allocation is realized by giving the sequentialized code to the scratchpad memory allocator.

All the transformations described above are optimizations. They are done by padCC, when they are possible. If the scratchpad cannot be allocated for some memory operation or some task cannot be scheduled statically, then that part of the code is left unchanged. As typical for optimizations, these actions are transparent to the user. Their main effect is to improve predictability of the timing of the execution. However, a deep understanding of the timing is needed to tune a program to fully use the features of the PAD system.

The PAD system has its roots in on our previous work on cache performance analysis [8]. The current version is designed for the ARM7TDMI processor [2] in a system that has 8kB of scratchpad memory.

5 Conclusion

The techniques used by us are not unique. The use of simple hardware features to promote predictability is very common in hard real-time systems. Also, using compiler techniques to implement statically-decided operation has been studied. However, we feel that multiple compiler techniques should be used in a coordinated way to promote predictability. In this way, our approach can be seen as an extension to the software synthesis approach [5] (exemplified by the compiler-based static scheduling) with new hardware-related optimizations [3, 14] (exemplified by scratchpad usage).

Our approach is a compromise between approaches toward very predictable systems (see [6, 12]) and the current practice. The building of predictable systems is also related to the corresponding analyses (e.g., [13]). Our research is especially dependent on the development of WCET analysis based on static program analysis and its relation to hardware development (see [7]). Our current practical work is limited. In the future, we aim at an implementation that makes full scale experimentation possible. Because of the successful previous studies on specific techniques (e.g., [9, 10]), we expect good results.

6 Acknowledgments

This work has been supported by Finnish Academy grant 51509. We also thank Kimmo Tuomainen and Juha Tukkinen for their input during early stages of this work.

References

[1] P. Altenbernd. *Timing Analysis, Scheduling, and Allocation of Periodic Hard Real-Time Tasks*. PhD Dissertation, Paderborn university, Department of Mathematics and Computer Science, 1996.

[2] ARM7TDMI Technical Reference Manual, 2001. Document code ARM DDI 0210B, rev 4, www.arm.com.

[3] Keith D. Cooper and Timothy J. Harvey. Compiler-controlled memory. In *Proceedings of the ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 2–11, San Jose, California, USA, 1998.

[4] S.A. Edwards. Compiling Concurrent Languages for Sequential Processors. *ACM Transactions on Design Automation of Electronic Systems (TODEAS)*, 8(2):141–187, 2003.

[5] R. K. Gupta and G. De Micheli. Hardware-software Co-synthesis for Digital Systems. *IEEE Design and Test of Computers*, pages 29–41, September 1993.

[6] J. Gustafsson, B. Lisper, R. Kirner, and P. Puschner. Input-Dependency Analysis for Hard Real-Time Software. In *Proceedings of the IEEE International Workshop on Object-Oriented Real-Time Dependable Systems (WORDS)*, October 2003.

[7] R. Heckmann, M. Langenbach, S. Thesing, and R. Wilhelm. The Influence of Processor Architecture on the Design and Results of WCET Tools. *Proceedings of the IEEE Symposium on Real-Time System (RTSS)*, 91(7):1038–1054, July 2003. Special Issue on Real-Time Systems.

[8] V. Hirvisalo. *Using Static Program Analysis to Compile Fast Cache Simulators*. PhD Dissertation, Helsinki University of Technology, March 2004.

[9] B. Lin. Efficient Compilation of Process-Based Concurrent Programs without Run-Time Scheduling. In *Proceedings of the Conference on Design, Automation and Test in Europe (DATE)*, pages 211–217, February 1998.

[10] P. Marwedel, L. Wehmeyer, M. Verma, S. Steinke, and U. Helmig. Fast, Predictable and Low Energy Memory References through Architecture-Aware Compilation. In *In Proceedings of Design Automation Conference Asia and South Pacific (ASPDAC)*, Yokohama, Japan, January 2004.

[11] I. Puaut. Cache Analysis vs Static Cache Locking for Schedulability Analysis in Multitasking Real-Time Systems. In *Proceedings of the International Workshop on Worst-Case Execution Time Analysis (WCET)*, Vienna, Austria, June 2002.

[12] P. Puschner and A. Burns. Writing Temporally Predictable Code. In *Proceedings of the IEEE International Workshop on Object-Oriented Real-Time Dependable Systems (WORDS)*, pages 85–91, January 2002.

[13] J. Schneider. Cache and Pipeline Sensitive Fixed Priority Scheduling for Preemptive Real-Time Systems. In *Proceedings of the IEEE Real-Time Systems Symposium (RTSS)*, pages 195–204, Orlando, Florida, USA, November 2000.

[14] J. Sjödin and C. von Platen. Storage allocation for embedded processors. In *Proceedings of the International Conference on Compilers, Architecture, and Synthesis for Embedded Systems (CASES)*, pages 15–23, Atlanta, Georgia, USA, 2001.