Integrating Gnat/Gcc into a Timing Analysis Environment

Dragan Macos and Frank Mueller
Humboldt-Universität zu Berlin, Institut f. Informatik
10099 Berlin (Germany)
e-mail: mueller@informatik.hu-berlin.de
phone: (+49) (30) 20181-276 fax:-280

Abstract

Recent research on timing analysis has focused on static methods to predict worst-case execution times that provide safe approximations of the actual worst-case execution time of real-time tasks. Since these predictions are verifiable, they can be used in schedulability analysis so that the overall temporal behavior of tasks can be analyzed. This paper details our efforts to integrate a commonly available and widely accepted compiler, the Gnat/Gcc family, into our tool set for timing analysis. We discuss the design and implementation of our modifications to Gnat/Gcc, where we accommodated our enhancements in a widely portable way within the common back-end of Gnat and Gcc. With this work, it is our aim to make the analytical approach of static timing analysis available to the real-time community.

1. Introduction

Real-time applications are concerned with the timely execution of tasks in response to events. In this context, the deadline of a task describes the latest point in time where such a task should have finished its execution. Several scheduling paradigms have been developed specifically for real-time systems to guarantee that deadlines are met, *e.g.* rate monotone scheduling, earliest deadline first, deadline monotonic etc. [14, 4]. These scheduling paradigms formulate schedulability tests that can be checked either off line (static schedulability tests) or on line before dispatching a task (dynamic schedulability test). The tests guarantee that deadlines can be met given a worst-case execution time (WCET) for each task within the system.

A prerequisite for using these schedulability tests is the derivation of a task's WCET. Two fundamentally different approaches exist. The first method derives the WCET experimentally by timing the tasks during sample runs of the application with various input. The second method derives the WCET analytically from the code of a task by simulating the characteristics of a processor without ever executing the task. Each approach has its merits.

This paper focuses on making the analytical approach available to the real-time community. It describes the set of tools involved in analytical derivation of the WCET (*static timing analysis*). In particular, modifications to the Gnu Ada Translator (Gnat) / the Gnu C Compiler (Gcc) are described that allow the integration of a widely accepted compiler into our tool set. Notice that Gnat and Gcc share a common back-end that is suitable to accommodate the required modification, enabling its use for Ada real-time applications and embedded C programs at the same time. It is our intent to demonstrate with such a prototype environment that static timing analysis should become an integral part of the software development cycle for hard real-time applications and portions of soft real-time applications requiring high reliability.

2. Related Work

In the past years, research in the area of predicting the WCET of programs has intensified. Conventional methods for static analysis have been extended from unoptimized programs on simple CISC processors to optimized programs on pipelined RISC processors [13, 8], and from uncached architectures to instruction caches [3, 11, 9] and data caches [18, 10, 12, 21].

Today, mainly three fundamental models for static timing analysis exist. First, a source-level oriented timing schema propagates times through a tree and handles pipelined RISC processors with first-level split caches [17, 13, 10]. Their most recent work includes modifications to the research compiler Lcc [7] to extract instruction information.

Second, a constraint-based method models architectural aspects, including caches, via integer linear programming

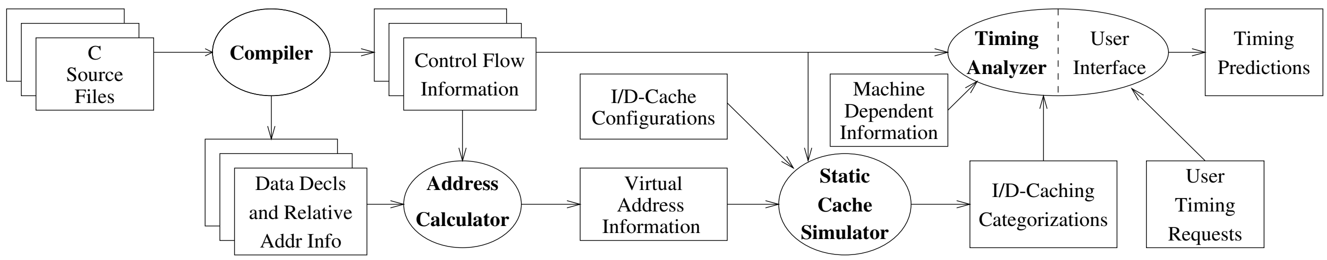

Figure 1: Framework for Timing Predictions

[11, 12]. They analyze the executable to extract the required instruction information. Another constraint-based approach along the same lines uses hand-generated information about instructions [16].

Third, our approach uses data-flow analysis to model the cache behavior separate from pipeline simulation, which is handled later in a timing analyzer via path analysis [3, 8, 21]. The first and second approaches use integrated analysis of caches while our approach uses separate analysis. This allows us to deal with multi-level memory hierarchies or unified caches [15]. Instruction information was extracted using the VPCC/VPO research compiler [5]. This work extends the applicability of our work to Gnat/Gcc compilation environments [19, 20].

Another approach using data-flow analysis to modeling caching originally used the same categorizations as our approach but a different data-flow model [2]. Recently, the approach has been generalized to handle a number of data-flow solutions with differing complexity and accuracy [6]. They extract instruction information directly from executables.

3. Motivation

This work concentrates on *static* timing analysis to provide the means to verify timing constraints within real-time programs. The previous section illustrates that related work on static timing analysis has been based either on extracting instruction information from executables or on modifying compilers to generate this information. The disassembly of executables allows timing of library functions whose sources are unavailable but the approach requires extensive work when migrating to new architectures, *i.e.,* it lacks portability. Modifications to a compiler facilitate the portability of the overall tool set, in particular for Gnat/Gcc since the compiler is already available for a multitude of architectures. Furthermore, Gnat and Gcc share a common back-end that is particularly suitable for modifications, as will be outlined later. However, library code has to be present as sources or the instruction information has to be available as a result of prior compilation. This constraint may be a limitation of our current approach. However, we can easily integrate our tool set with approaches to analyze executables. Furthermore, operating system calls remain beyond the reach of analyzing executables implying that timing information needs to be supplied by the vendor in any case. Finally, our approach is aimed at a modular analysis method in the long run to cope with library code, operating system calls and the complexity of timing analysis in the same token.

Most of the related work discussed in the last section has been performed for a single RISC architecture with uniform instruction format for each experiment. Our current work is aimed at making our approach more portable to work with arbitrary RISC and CISC architectures.

4. Static Timing Analysis

Our framework of WCET prediction uses a set of tools as depicted in Figure 1. An optimizing compiler has been modified to emit control-flow information, data information, and the calling structure of functions in addition to regular target code generation. Currently, the research compiler VPCC/VPO [5] performs this task. In this paper, we describe our early experience in integrating Gnat/Gcc [19, 20] into this environment.

A static cache simulator uses the control-flow information and calling structure in conjunction with the cache configuration to produce instruction and data categorizations, which describe the caching behavior of each instruction and data reference, respectively. We currently use a separate analyzer for instruction and data caches since data references require separate preprocessing *via* an address calculator. Current work also includes a single analyzer for unified caches and the handling of secondary caches [15]. The timing analyzer uses these categorizations and the control-flow information to perform path analysis of the program. The timing analyzer produces WCET predictions for portions of the program or the entire program, depending on user requests.

5. Modifications of Gnat/Gcc

We modified the back-end of the Gnat/Gcc compiler to emit instruction information for Ada and C programs. The compiler provides the input for the static cache simulator, which returns results of the cache simulation and the pipeline simulator. The latter also uses the results of the cache simulation and returns timing predictions. In our approach, most modifications are performed on the RTL-level in order to provide high portability. The overall approach can be described as a register-switch system where switches provide information about analyzed program constructs and registers supply concrete information about relevant program entities. Switches are booleans and registers are integers and stacks. For example, if the switch *it_was_conditional_jump* evaluates to true, then we know that the analysis has just passed a conditional jump encoded as an RTL. If the top of the stack *count_loop_stack* contains the value 3, then this indicates that the current program constructs are on the loop nesting level of depth 3. The register *count_instructions* counts the machine instructions. It would be possible to define the operational semantics of the register-switch system through transitions on the states whose components are sets of switches, sets of registers and the program represented through RTLs.

The entire program analysis is performed before assembler code emission. All operations are defined to cooperate with peephole optimization [1] so that the correctness of our approach is guaranteed for all optimizations, even for machine-dependent ones.

The modifications of the compiler could be split into two sets:

1. collecting data-flow information and
2. counting machine instructions.

The entire data-flow analysis is performed on the RTL-level. It provides information about jumps, loops and function calls and creates basic block nodes that are relevant for our cache and pipeline simulation models. Every basic block contains information about its number of instructions, first and last RTLs, etc. Figure 2 shows the structure of basic blocks generated by our data-flow analysis. Unfortunately, Gnat/Gcc does not incrementally update such information but rather calculates it on demand from the RTLs

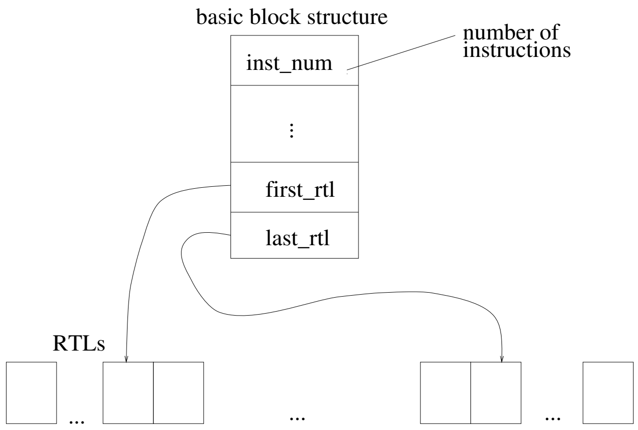

Figure 2: Structure of Basic Blocks

when needed.

The counting of machine instructions is performed by inspecting instruction templates during assembler code emission. For the Gnat/Gcc compiler, it is impossible to perform this operation on the RTL-level because machine-dependent components of the compiler contain templates that transform one RTL into multiple machine instructions. Instead, instruction counting is done in a machine-independent program module of the compiler back-end. For the RISC-architectures, we also had to modify one machine-dependent program module because the machine code is directly emitted into the output stream.

Functions for operations on switches, registers and functions for computing information that we need for the data-flow analysis are implemented in an additional program module. Figure 3 depicts the infrastructure of the modified compiler back-end.

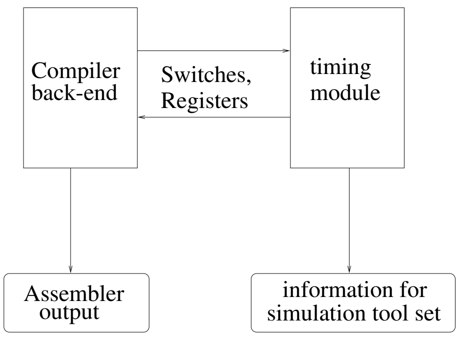

Figure 3: Extended Compiler Back-End

This modular approach simplifies the development and, at the same time, facilitate the integration of our modifications into new versions of Gnat/Gcc compiler releases. We expect to support future compiler upgrades by simply merging calls to our new module into the components of the compiler back-end. Even when the Gnat/Gcc back-end is modified, we can still insert the calls at the proper place unless major structural changes of the back-end occur. This allows us to support future Gnat/Gcc releases, even though our enhancements for timing analysis are not an integral part of Gnat/Gcc.

6. Future Work

After integrating the Gnat/Gcc compiler with the static instruction cache simulator, we will add the proper extensions to generate input for static data cache analysis and for pipeline simulation. This requires the availability of dominator information and knowledge about the regularity of data accesses within a loop, if there are any regularities and if the compiler can determine them analytically. Furthermore, we are developing concepts for simulating variable pipeline stages, instruction-level parallelism and branch prediction within the pipeline simulator.

7. Conclusion

In this paper, we described our work of integrating a commonly available and widely accepted compiler, the Gnat/Gcc family, into our tool set for timing analysis. We discussed the design and implementation of our modifications to Gnat/Gcc. Our enhancements were integrated by inserting calls to our analysis module from machine-independent modules within the common back-end of Gnat and Gcc. We performed the analysis immediately before emitting assembler code to cover the effect of peephole optimization, which is the last optimization within Gnat/Gcc. Due to the inconsistent handling of machine dependencies within Gnat/Gcc, a small number of modifications had to be made within machine-dependent code. Furthermore, we had to recalculate control-flow and data-flow information since they are calculated on demand only within certain optimizations. This work is one step on the way to make the analytical approach of static timing analysis available to the real-time community. Using these tools, the resulting timing predictions, which are verifiable, can be used in schedulability analysis so that the overall temporal behavior of tasks can be analyzed.

References

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

[2] M. Alt, C. Ferdinand, F. Martin, and R. Wilhelm. Cache behavior prediction by abstract interpretation. In *Static Analysis Symposium*, September 1996.

[3] R. Arnold, F. Mueller, D. B. Whalley, and M. Harmon. Bounding worst-case instruction cache performance. In *IEEE Real-Time Systems Symposium*, pages 172–181, December 1994.

[4] N. Audsley, A. Burns, and A. J. Wellings. Deadline monotonic scheduling theory and application. *J. Control Engineering Practice*, pages 71–78, 1993.

[5] 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.

[6] C. Ferdinand, F. Martin, and R. Wilhelm. Applying compiler techiniques to cache behavior prediction. In *ACM SIGPLAN Workshop on Language, Compiler, and Tool Support for Real-Time Systems*, pages 37–46, June 1997.

[7] C. Fraser and D. Hanson. *A Retargetable C Compiler*. Addison-Wesley, 1995.

[8] C. A. Healy, D. B. Whalley, and M. G. Harmon. Integrating the timing analysis of pipelining and instruction caching. In *IEEE Real-Time Systems Symposium*, pages 288–297, December 1995.

[9] Y. Hur, Y. H. Bea, S.-S. Lim, B.-D. Rhee, S. L. Min, Y. C. Park, M. Lee, H. Shin, and C. S. Kim. Worst case timing analysis of RISC processors: R3000/R3010 case study. In *IEEE Real-Time Systems Symposium*, pages 308–319, December 1995.

[10] S. Kim, S. Min, and R. Ha. Efficient worst case timing analysis of data caching. In *IEEE Real-Time Technology and Applications Symposium*, June 1996.

[11] Y.-T. S. Li, S. Malik, and A. Wolfe. Efficient microarchitecture modeling and path analysis for real-time software. In *IEEE Real-Time Systems Symposium*, pages 298–397, December 1995.

[12] Y.-T. S. Li, S. Malik, and A. Wolfe. Cache modeling for real-time software: Beyond direct mapped instruction caches. In *IEEE Real-Time Systems Symposium*, pages 254–263, December 1996.

[13] S.-S. Lim, Y. H. Bea, G. T. Jang, B.-D. Rhee, S. L. Min, Y. C. Park, H. Shin, and C. S. Kim. An accurate worst case timing analysis for RISC processors. In *IEEE Real-Time Systems Symposium*, pages 97–108, December 1994.

[14] C.L. Liu and James W. Layland. Scheduling algorithms for multiprogramming in a hard-real-time environment. *Journal of the Association for Computing Machinery*, 20(1):46–61, January 1973.

[15] F. Mueller. Timing predictions for multi-level caches. In *ACM SIGPLAN Workshop on Language, Compiler, and Tool Support for Real-Time Systems*, pages 29–36, June 1997.

[16] G. Ottosson and M. Sjodin. Worst-case execution time analysis for modern hardware architectures. In *ACM SIGPLAN Workshop on Language, Compiler, and Tool Support for Real-Time Systems*, pages 47–55, June 1997.

[17] C. Y. Park. Predicting program execution times by analyzing static and dynamic program paths. *Real-Time Systems*, 5(1):31–61, March 1993.

[18] J. Rawat. Static analysis of cache analysis for real-time programming. Master's thesis, Iowa State University, 1995.

[19] Richard M. Stallman. *Using and Porting GNU CC*. Free Software Foundation, Inc., 2.7.2 edition, November 1995.

[20] New York University. The Gnu NYU Ada Translator (GNAT). Available by anonymous FTP from cs.nyu.edu.

[21] R. White, F. Mueller, C. Healy, D. Whalley, and M. Harmon. Timing analysis for data caches and set-associative caches. In *IEEE Real-Time Technology and Applications Symposium*, pages 192–202, June 1997.