A Worst Case Timing Analysis Technique for Optimized Programs*

Sung-Soo Lim
Dept. of Computer Engineering
Seoul National University
Seoul, Korea, 151-742
sslim@archi.snu.ac.kr

Jihong Kim†
Dept. of Computer Science
Seoul National University
Seoul, Korea, 151-742
jihong@davinci.snu.ac.kr

Sang Lyul Min
Dept. of Computer Engineering
Seoul National University
Seoul, Korea, 151-742
symin@dandelion.snu.ac.kr

Abstract

We propose a technique to analyze the worst case execution times (WCETs) of optimized programs. Our work is based on a hierarchical timing analysis technique called the extended timing schema (ETS). A major hurdle in applying the ETS to optimized programs is the lack of correspondences in the control structure between the optimized machine code to be analyzed and the original source program written in a high-level programming language. We suggest a compiler-assisted approach where a timing analyzer relies on an optimizing compiler for a consistent hierarchical representation and an accurate source-level correspondence that are essential for accurate WCET analysis for optimized programs. In order to validate the proposed approach, we implemented a proof-of-concept version of a timing analyzer for a 256-bit VLIW processor and compared the analysis results with the simulation results. The experimental results show that the proposed solution can accurately predict the WCETs of highly-optimized VLIW programs.

1. Introduction

In building real-time systems, the worst case execution times (WCETs) of tasks should be predicted in advance. The prediction results should be both *safe* (i.e., the predicted WCET should not be smaller than the real WCET) and *accurate* (i.e., the difference between the predicted WCET and the real WCET should be small). Unsafe prediction may cause catastrophic results by unexpected deadline misses of the tasks whose WCETs are not safely predicted. On the other hand, inaccurate prediction leads to a pessimistic schedulability analysis that results in under-utilization of system resources.

To obtain an accurate WCET for a target program, a timing prediction method should recognize the program structure (such as a conditional statement or a loop) of the target program and take into account the architectural features (such as cache and pipelining) of a target hardware platform. Previous studies on the WCET analysis, therefore, focused on three factors: 1) the representation of the target program structure that facilitates the WCET analysis, 2) the modeling of the architectural features that affect the WCET analysis results, and 3) the rules of composition that specify how to calculate the WCET of the target program using the program representation and architecture modeling [7, 6, 3]. One popular choice for representing the target program's structure is a hierarchical organization of program constructs using program syntax trees [10, 7]. This representation nicely depicts the natural hierarchical structure of a high-level target program and allows a simple and accurate timing analysis by recursively applying a small number of timing formula to each program construct.

For simple RISC processors, using the extended timing schema (ETS), we have shown that a timing prediction method based on a hierarchical program representation can accurately predict the WCET [7]. In the ETS, a set of timing formulas is defined for computing the WCETs of high-level program constructs. Each formula (corresponding to a particular program construct) defines a rule for computing the

*This work was supported in part from KOSEF under Grant 97-01-02-05-01-3.
†Jihong Kim was supported in part by Equipment Award for New Faculty from the College of Natural Sciences, Seoul National University.

WCET of the corresponding program construct. The ETS accurately accounts for the timing effects of pipelined execution and cache memory not only within but also across program constructs. To compute the WCET of a program, these formulas are recursively applied to the syntax tree of the whole program. For predicting the WCETs of unbounded loop statements, loop iteration bounds are additionally provided by user. For an accurate timing prediction, the execution time for a basic block is directly computed from the machine code generated by a compiler.

In order to apply the ETS to a target program, the program syntax tree of the target program must match the control structure of the machine code to be analyzed. This was the case for simple RISC processors when a target program is well-structured (i.e., no goto statements). However, for highly optimized processors such as VLIW processors, the control structure of an optimized machine code is often different from that of a high-level target program. For example, two nested loops may be collapsed into a single loop after appropriate loop optimizations, resulting in a quite different control structure in the optimized code, compared with the original syntax tree from the high-level target program. Moreover, in an optimized loop, the number of iterations can be different from that of an original loop. This is often the case when a software pipelining is used. If a loop is software-pipelined, a software-pipelined version of the loop may have a different loop iteration bound [5]. If a software-pipelined loop was unbounded, a user-provided iteration bound should be modified as well before the WCET for the loop is computed. Therefore, to apply the ETS to optimized programs, two requirements should be satisfied: 1) a program representation should match the control structure of an optimized code (*consistent hierarchical representation*), and 2) the loop correspondence relations between high-level loops and optimized loops, and information on modified loop iteration bounds should be known to a timing analyzer (*the loop correspondence requirement*).

This paper proposes an approach that can provide a timing analyzer with a consistent hierarchical program representation and an accurate loop correspondence between a high-level program and its corresponding machine-level program. In this approach, an optimizing compiler outputs the intermediate information on how the original loops of the high-level program are transformed by compiler optimizations performed. A timing analyzer then predicts the WCET of the optimized program using the intermediate information generated from an optimizing compiler. We call this type of a timing tool *a compiler-assisted timing analyzer*. Unlike the previous work [2], our method takes a minimalist's approach in defining the intermediate information.

In order to validate the proposed approach, we implemented a proof-of-concept version of timing analyzer for TMS320C6201 processor, a 256-bit wide VLIW digital signal processor (DSP) from Texas Instruments. We chose TMS320C6201 as a target processor because optimizing compilers for VLIW processors generally employ the most aggressive compiler optimizations such as software pipelining. To validate the analysis results, we compared the timing analysis results with the simulation results measured on a TMS320C6201 simulator from Texas Instruments. The preliminary results show that the proposed approach can accurately predict the WCETs of highly-optimized TMS320C6201 programs.

The rest of this paper is organized as follows: In Section 2, we survey the related work. Section 3 briefly describes the ETS on which our work is based. Section 4 summarizes the issues in applying the ETS to optimized programs and explains the design of a compiler-assisted timing analyzer. In Section 5, we describe an initial implementation of a timing analyzer for TMS320C6201 and present the experimental results. The conclusion and future work are given in Section 6.

2. Related Work

The effects of compiler optimizations on the WCET analysis have been recently studied by Vrchoticky [15] and Engblom, Ermedahl, and Altenbernd [2]. In order to maintain the source-level correspondence during the optimization phases, Vrchoticky proposed an extension of a hierarchical program representation called a timing tree. In this approach, as the optimization phases proceed, an initial timing tree is repeatedly modified to reflect optimizations performed. Vrchoticky's work focused on the standard optimization tasks such as constant propagation and dead code elimination, and did not consider in detail more aggressive optimization techniques that may change the program structure significantly. For example, it is not clear how a software pipelining can be supported using this technique. The lack of efficient support for loop optimizations may make a timing tree-based technique inapplicable to modern high-performance processors (such as VLIW processors) that depend on the loop optimizations for realizing their potential high performance.

Engblom, Ermedahl, and Altenbernd proposed a similar approach called *co-transformation* to reflect the effects of compiler optimizations on the WCET analysis. This approach is different from the Vrchoticky's work in that an optimizing compiler is not fully responsible for maintaining the source-level correspondence. Instead, the source-level correspondence is maintained by cooperations between an optimizing compiler and a separate timing tool. The authors proposed a description language called the *optimization description language (ODL)* to specify the effect of an optimization technique on the changes to the original program structure. Using the ODL description of optimization tech-

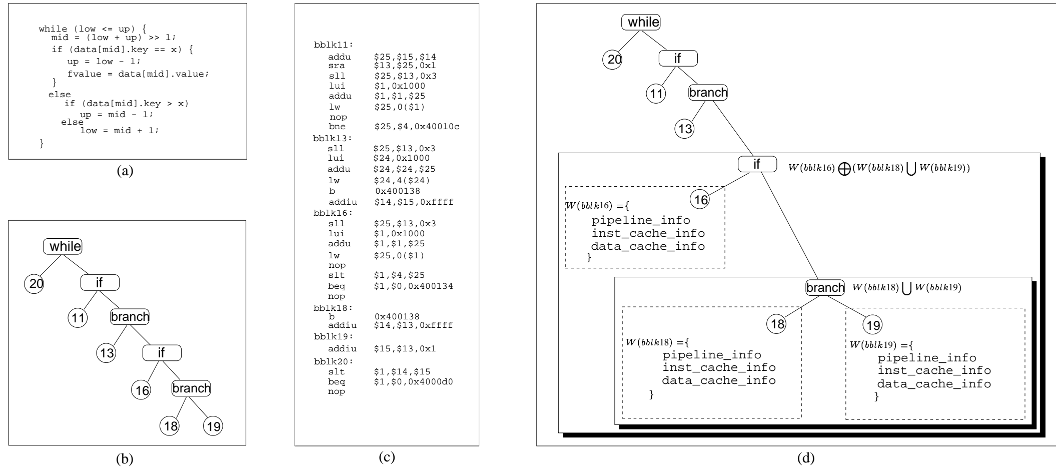

Figure 1. Overall processing steps in the ETS: (a) a sample high-level program, (b) the syntax tree, (c) the assembly code, and (d) the steps for applying the ETS to the sample program.

niques performed for a high-level program, an optimizing compiler produces optimization traces. The separate timing tool then reasons about the source-level correspondence of an optimized code based on the optimization traces and the ODL description of optimization techniques.

While this approach provides the well-defined interface for both compiler developers and timing tool developers to maintain the source-level correspondence of an optimized code, it imposes unnecessary additional burdens on them. The compiler developers should describe all the optimization techniques implemented in their compilers using the ODL, and modify the compilers to understand the ODL description and to produce optimization traces. Likewise, the timing tool developers should understand the ODL description and optimization traces to reconstruct the source-level correspondence. However, for a hierarchical framework such as the ETS, the detailed compiler optimizations provided by the ODL may not be all necessary. For example, the ETS requires only the loop correspondence information, not the exact specification of the optimization used. Our approach provides an ETS-based hierarchical framework with only the minimal information necessary to maintain the loop correspondence.

3. Extended Timing Schema

Figure 1 shows the overall processing steps in the original ETS approach. Figure 1(a) is a high-level program fragment for the binary search algorithm and Figure 1(b) is the corresponding syntax tree generated from the front-end of a compiler. Figure 1(c) is the assembly code using the instruction set of MIPS R3000 processor which is the hardware platform of our previous work [7]. As shown in Figure 1(a), the example program consists of a `while` loop and two nested conditional statements are within the loop. The syntax tree in Figure 1(b) correctly represents the structure of the sample program. Moreover, the basic blocks in the assembly code in Figure 1(c) are well mapped to the nodes in the syntax tree, which provides the correct source-level correspondence. For each node in the syntax tree, the timing-related information called the Worst Case Timing Abstraction (WCTA) is associated. The WCTA encodes the architectural features that may cause the timing variation of the program construct. The encoding of WCTAs is done in such a way that allows for refinement of the execution path's WCET when the detailed information about the surrounding execution paths becomes available.

Figure 1(d) illustrates the processing steps in applying the ETS to the sample program. In the figure, the boxes with dotted lines are the timing information for basic blocks. On the other hand, the larger boxes with solid lines represent nested structure of the program, where predefined timing formulas[1] are recursively applied. For example, for the **branch** node, the timing formula $W(bblk18) \bigcup W(bblk19)$ is applied. On the other hand, for the **if** node, the timing formula $W(bblk16) \bigoplus (W(bblk18) \bigcup W(bblk19))$ is applied [7].

As shown in Figure 1, in the original ETS work [7], the syntax tree was generated from the front-end of a modified RISC compiler. Although there exist various types of compiler optimizations that can improve the performance of

[1]In the formulas, $\bigoplus$ denotes the concatenation of two blocks, which models sequential execution of the two blocks. On the other hand, by $\bigcup$, all the possible execution paths in a program construct are considered.

RISC processors, we assumed that most of these optimizations maintain the structure of the program syntax tree after the optimization transformations. However, as explained in Section 1, for aggressively optimized programs, it is no longer a safe assumption that an optimized machine code has the same control structure as an original program syntax tree. In predicting the WCETs of optimized programs under the ETS framework, the consistent hierarchical representation and loop correspondence requirements can be best satisfied if an optimizing compiler can generate both a hierarchical representation of an optimized program and information to make the correspondence between high-level loops and machine-level loops.

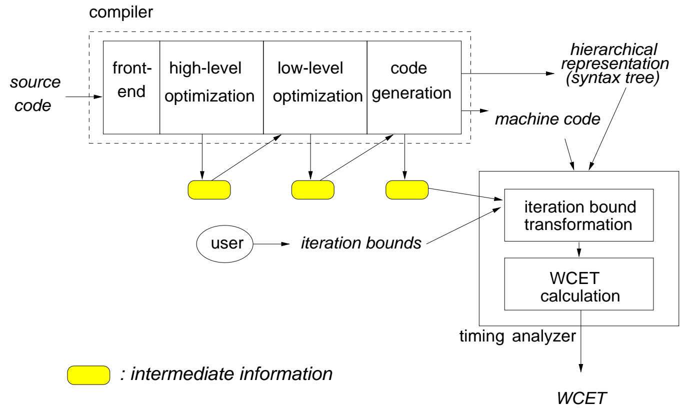

Figure 2. Overview of a compiler-assisted timing analyzer.

4. Compiler-Assisted Timing Analyzer

We propose a framework where a timing analyzer is assisted by an optimizing compiler to satisfy the consistent hierarchical representation and loop correspondence requirements. The overview of the framework is shown in Figure 2. In the figure, the timing analyzer uses several outputs from the optimizing compiler: the hierarchical representation, the machine code and the intermediate information. The hierarchical representation is generated at the last stage of the code generation phase while, in the original ETS, the syntax tree from the front-end phase was used. Therefore, the hierarchical representation would correctly reflect the effect of the optimization transformations applied to the original high-level program during the optimization phases. In order to link the loops of high-level code to the loops of machine code, special labels are attached to the loops in the high-level source program by the compiler and maintained until the machine code is generated. This allows a user to interact with a timing tool at the original source program level for each loop. The intermediate information produced by the compiler contains the following information: 1) special labels for loops that were attached by the compiler and included in the machine code, and 2) an expression to recompute the loop iteration bounds for optimized loops. Using the loop-bound recomputing expression, the iteration bound transformation module translates a user-provided loop bound to a new loop bound reflecting the optimizations performed for a loop. In case that a loop is optimized by multiple stages of compiler optimizations, the intermediate information from each stage of compilation is read to the succeeding stage for the subsequent update.

As an example of compiler-assisted timing analysis information, consider a high-level source code shown in Figure 3(a). In order to maintain the loop correspondence after optimizations, two special labels are attached to the optimized source code as shown in Figure 3(b). Two labels indicate that the subsequent statements are loops. After the compiler optimizations, the modified source code is compiled to the assembly code shown in Figure 3(d). The original nested loops are transformed into the single loop by software pipelining. (The assembly instructions shown in Figure 3(d) are TMS320C6201 assembly instructions[2].) In the intermediate information file shown in Figure 3(c), the formula for computing a modified iteration bound is specified next to the attached special labels `LOOP_1_!` and `LOOP_2_!`. Two labels shown in the same line mean that two original loops are transformed to a single loop. In this optimization, the iteration bound of the optimized loop was changed to `SIZE-2` from `SIZE` in the original loops. As another example, consider a loop which is unrolled by U times. In this case, the loop iteration bound formula will be $\lfloor$`SIZE/U`$\rfloor$. The special labels attached to the loops and the expression for the number of loop iterations in the optimized loop allows the ETS to satisfy two requirements for analyzing the WCETs of the optimized program.

5. Preliminary Results

In this section, we describe a proof-of-concept version of a compiler-assisted timing analyzer for a VLIW digital signal processor from Texas Instruments, TMS320C6201, and compare the analysis results with the simulation results for a validation purpose. Before the timing analyzer is explained, we briefly describe TMS320C6201.

5.1 TMS320C6201 VLIW DSP

[2]The assembly instruction format contains several special features: 1) the parallel bars (||) represent that the instruction is simultaneously executed with the immediately preceding instruction, 2) the brackets indicate the conditional execution of the corresponding instructions, and 3) the functional unit where the instruction is executed is explicitly shown as a field starting from a dot (e.g., `.L1`). For a more detailed description on the instruction format, see the reference [13].

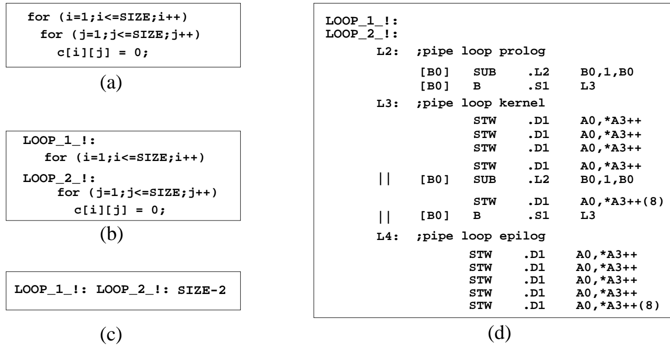

Figure 3. Compiler-assisted timing analysis information: (a) a high-level source code, (b) the source code with two special labels attached for loop correspondence, (c) the intermediate information on the optimizations performed, and (d) the optimized assembly code with the special labels.

TMS320C6201 is a fixed-point DSP processor based on the VelociTI^{TM} VLIW architecture [14]. Figure 4 shows a high-level block diagram of the major functional blocks of TMS320C6201. It has eight functional units and can execute up to eight instructions in a single cycle using a 256-bit wide instruction format. Eight functional units are grouped into two sets of functional units, A and B. Each set of functional units consists of a 40-bit integer ALU (**.L1** and **.L2**), a 40-bit shifter (**.S1** and **.S2**), a 16-bit multiplier (**.M1** and **.M2**), and a 32-bit adder/address generator (**.D1** and **.D2**). TMS320C6201 fetches a 256-bit eight-instruction group (called a fetch packet) at once. A single fetch packet is further divided into multiple execution packets, depending on how many instructions (out of eight instructions) can be executed in parallel. All instructions in an execution packet are dispatched together [11].

As with other VLIW processors, TMS320C6201 heavily depends on the optimizing compiler for high performance. The optimizing compiler performs various aggressive optimizations to fully take advantage of the TMS320C6201's performance-enhancing features such as the 8-way instruction-level parallelism, 11-stage pipelined organization and conditional instructions [12].

5.2. Proof-of-Concept Implementation of TMS-320C6201 Timing Analyzer

The compiler-assisted approach explained in Section 4 requires some efforts to build a timing analyzer because an optimizing compiler should be modified to produce a hierarchical representation and loop optimization information. Furthermore, for our target processor, TMS320C6201,

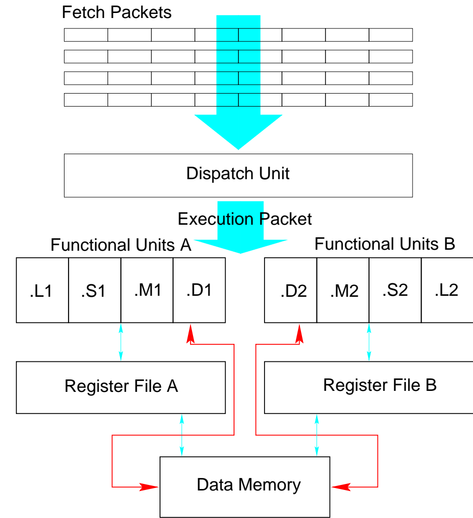

Figure 4. TMS320C6201 architecture.

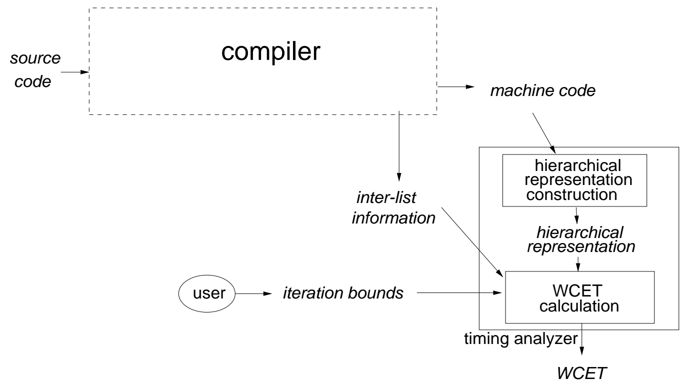

Figure 5. Overview of the proof-of-concept implementation of timing analyzer.

no public domain optimizing compiler is currently available. Because of these difficulties, in order to validate the proposed approach, we decided to build a proof-of-concept timing analyzer instead of a complete compiler-assisted timing analyzer for TMS320C6201.

Because of several shortcut solutions used, we were able to build a proof-of-concept timing analyzer quickly. As shown in Figure 5, a proof-of-concept timing analyzer accepts as an input the same machine code as a complete version, but differs in three aspects. First, the hierarchical representation is extracted from the machine code produced by the optimizing compiler because the TMS320C6201 compiler does not generate the required hierarchical representation. In constructing the hierarchical representation from the machine code, we adapted a structuring algorithm from Cifuentes' decompilation work [1]. Second, in order to maintain the loop correspondence, instead of the interme-

Benchmark Programs	Description
MatMul	multiplies two 5×5 integer matrices
JFDCTINT	performs the forward Discrete Cosine Transform used in JPEG
FIR	performs a 32-taps Finite Impulse Response (FIR) filtering operation
FFT	performs the Fast Fourier Transform (FFT) on 256 floating point numbers

Table 1. The benchmarks used for the experiments.

Benchmark Programs	Simulation Results	Analysis Results
MatMul	1673	1739
JFDCTINT	4456	4780
FIR	30940	32218
FFT	2879360	4567872

Table 2. Predicted and measured execution cycles of the benchmark programs.

diate information shown in Figure 2, we use the inter-list information that is optionally generated by the optimizing compiler. The inter-list information includes the line numbers of a high-level source program that correspond to assembly code fragments. The timing analyzer can reconstruct the loop correspondence information using the inter-list information. Third, the loop iteration bound transformation is done manually. Without the help of an optimizing compiler, it is very difficult to guess how an original loop was transformed. In the current implementation, for an optimized loop, a new loop bound is manually computed by analyzing an original high-level loop code and its optimized assembly code.

5.3. Experimental Results

In order to validate the timing analyzer for TMS320-C6201, we compared the analysis results from the timing analyzer with the simulation results. The simulation results are based on the measurements from a cycle-accurate TMS320C6201 simulator available from Texas Instruments. The benchmark programs used for the experiments are given in Table 1. Table 2 compares the analysis results and the simulation results for the benchmark programs. Among the benchmark programs, the *MatMul*, *JFDCTINT* and *FIR* show accurate analysis results with the less than 5% overestimation. These programs have a single program execution path, thus do not suffer from the infeasible path problem [8]. On the contrary, the *FFT* shows much larger overestimation than the results for the other benchmark programs. This is because the *FFT* benchmark program has multiple program execution paths and some of them (including the worst case execution path computed by the timing analyzer) are the infeasible paths by the program logic. The static timing analysis technique such as ours cannot identify the infeasible paths during the analysis. We consider the elimination of infeasible paths using dynamic path analysis is an issue orthogonal to our approach. Existing techniques for eliminating infeasible paths of a program such as ones described in [8, 9] can be easily integrated with the approach proposed in this paper.

6. Conclusion

We described a compiler-assisted approach that can extend an ETS-based timing prediction technique for aggressively optimized programs. In our hierarchical framework, a compiler-assisted timing analyzer relies on an optimizing compiler for a consistent hierarchical representation and information for the loop correspondence that are essential for accurate WCET analysis for optimized programs. In our approach, the minimal information is passed to a timing analyzer from an optimizing compiler, thus requiring less implementation burdens from compiler developers and timing tool developers. For a validation purpose, we built a proof-of-concept timing analyzer for Texas Instruments' new VLIW DSP, TMS320C6201. Preliminary experimental results suggest that the proposed approach can accurately predict the WCETs of highly-optimized programs.

Our next task is to build a complete compiler-assisted timing analyzer interfacing directly with intermediate information and a hierarchical program representation generated by an optimizing compiler. We plan to implement a timing tool using the Stanford University Intermediate Form (SUIF) compiler system [4].

References

[1] C. Cifuentes. *Reverse Compilation Techniques.* PhD thesis, The School of Information Technology, Queensland University of Technology, July 1994.

[2] J. Engblom, A. Ermedahl, and P. Altenbernd. Facilitating Worst-Case Execution Times Analysis for Optimized Code. In *Proceedings of the 10th Euromicro Workshop on Real-Time Systems*, June 1998.

[3] C. A. Healy, D. B. Whalley, and M. G. Harmon. Integrating the Timing Analysis of Pipelining and Instruction Caching. In *Proceedings of the 16th Real-Time Systems Symposium*, pages 288–297, December 1995.

[4] H. Kienle and U. Holzle. Introduction to the SUIF 2.0 Compiler System. Technical Report TRC97-22, Dept. of

Computer Science, University of California Santa Barbara, December 1997.

[5] M. Lam. Software Pipelining: An Effective Scheduling Technique for VLIW Machines. In *Proceedings of the SIGPLAN '88 Conference on Programming Language Design and Implementation*, pages 318–328, 1988.

[6] Y. S. Li, S. Malik, and A. Wolfe. Efficient Microarchitecture Modeling and Path Analysis for Real-Time Software. In *Proceedings of the 16th Real-Time Systems Symposium*, pages 298–307, December 1995.

[7] S.-S. Lim, Y. H. Bae, G. T. Jang, B.-D. Rhee, S. L. Min, C. Y. Park, H. Shin, K. Park, and C. S. Kim. An Accurate Worst Case Timing Analysis for RISC Processors. *IEEE Transactions on Software Engineering*, 21(7):593–604, July 1995.

[8] C. Y. Park. Predicting Program Execution Times by Analyzing Static and Dynamic Program Paths. *Real-Time Systems*, 5(1):31–62, March 1993.

[9] P. Puschner and C. Koza. Calculating the Maximum Execution Time of Real-Time Programs. *Journal of Real-Time Systems*, 1(2):159–176, September 1989.

[10] A. C. Shaw. Reasoning About Time in Higher-Level Language Software. *IEEE Transactions on Software Engineering*, 15(7):875–889, July 1989.

[11] Texas Instruments. *TMS320C62xx CPU and Instruction Set Reference Guide*, 1997.

[12] Texas Instruments. *TMS320C62xx Optimizing C Compiler User's Guide*, 1997.

[13] Texas Instruments. *TMS320C62xx Programmer's Guide*, 1997.

[14] Texas Instruments. *TMS320C62xx Technical Brief*, 1997.

[15] A. Vrchoticky. Compilation Support for Fine-Grained Execution Time Analysis. In *Proceedings of the ACM SIGPLAN Workshop on Languages, Compilers and Tools for Real-Time Systems*, June 1994.