Static Timing Analysis of Embedded Software

Sharad Malik Margaret Martonosi
Yau-Tsun Steven Li
Department of Electrical Engineering, Princeton University

Abstract

This paper examines the problem of statically analyzing the performance of embedded software. This problem is motivated by the increasing growth of embedded systems and a lack of appropriate analysis tools. We study different performance metrics that need to be considered in this context and examine a range of techniques that have been proposed for analysis. Very broadly these can be classified into path analysis and system utilization analysis techniques. It is observed that these are interdependent, and thus need to be considered together in any analysis framework.

1 The Emergence of Embedded Systems

Embedded systems are characterized by the presence of processors running application specific programs. Typical examples include printers, cellular phones, automotive engine controller units, etc. A key difference between an embedded system and a general-purpose computer is that the software in the embedded system is part of the system specification and does not change once the system is shipped to the end user.

Recent years have seen a large growth of embedded systems. The migration from application specific logic to application specific code running on processors is driven by the demands of more complex system features, lower system cost and shorter development cycles. These can be better met with software programmable solutions made possible by embedded systems. Two distinct points are responsible for this.

Flexibility of Software Software is easier to develop and is more flexible than hardware. It can implement more complex algorithms. By using different software versions, a family of products based on same hardware can be developed to target different market segments, reducing both hardware cost and design time. Software permits the designer to enhance the system features quickly so as to suit the end users' changing requirements and to differentiate the product from its competitors.

Increasing Integration Densities The increase in integration densities makes available 1-10 Million transistors on a single IC today. With these resources, the notion of a "system on a chip" is becoming a viable implementation technology. This integrates processors, memory, peripherals and a gate array ASIC on a single IC. This high level of integration reduces size, power consumption and cost of the system. The programmable component of the design increases the applicability of the design and thus the sales volume, amortizing high manufacturing setup costs. Less reusable application specific logic is getting increasingly expensive to develop and manufacture and is the solution only when speed constraints rule out programmable solutions.

The pull effect offered by flexibility of software and the push effect from increasingly expensive application specific logic solutions make embedded systems an attractive solution. As system complexity grows and microprocessor performance increases, the embedded system design approach for application specific systems is becoming more appealing. Thus, we are seeing a movement from the logic gate being the basic unit of computation on silicon, to an instruction running on an embedded processor. This motivates research efforts in the analysis of embedded software. Our capabilities as researchers and tool developers to model, analyze and optimize the gate component of the design must now be extended to handle the embedded software component. This paper examines one such aspect for embedded software – techniques for statically analyzing the timing behavior (i.e., the performance) of embedded software.

By *static analysis*, we refer to techniques that use results of information collected at or before compile time. This may include information collected in profiling runs of the code executed before the final compilation. In contrast, dynamic performance analysis refers to on-the-fly performance monitoring while the embedded software is installed and running. We limit the scope of this paper by considering only single software components, i.e. the execution of a single program on a known processor. The analysis of multiple processes belongs to the larger field of system level performance analysis.

We start by examining the various performance metrics of interest in Section 2. Next, we look at the different applications of performance analysis in Section 3. In Section 4 we examine the different components that make this analysis task difficult, and for each we summarize the analysis techniques that are described in existing literature. Finally, in Section 5 we conclude and point out interesting future directions for research.

2 Performance Metrics for Embedded Software

Extreme Case Performance Embedded systems generally interact with the outside world. This may involve measuring sensors and controlling actuators, communicating with other systems, or interacting with users. These tasks may have to be performed at precise times. A system with such timing constraints is called a *real-time system.* For a real-time system, the correctness of the system depends not only on the logical results of computation, but also on the time at which the results are produced. A real-time system can be further classified as either a *hard real-time system* or a *soft real-time system.* A hard real-time system cannot tolerate any missed timing deadlines. An example of a hard real-time system is an automotive engine control unit, which must gather data from sensors, and compute the proper air/fuel mixture and ignition timing for the engine, within a single rotation. In such systems, the response time must comply with the specified timing constraints under all possible conditions. Thus the performance metric of interest here is the *extreme case* performance of the software. Typically, the *worst-case* is of interest, but in some cases, the *best-case* may also be important to ensure that the system does not respond faster than expected.

Probabilistic Performance In a soft real-time system, the timing requirements are less stringent. Occasionally missing a timing deadline is tolerable. An example of a soft real-time system is a cellular phone. During a conversation, it must be able to encode outgoing voice and decode the incoming signal in real-time. Occasional glitches in conversation due to missed deadlines are not desired, but are nevertheless tolerated. In this case, a *probabilistic performance* measure that guarantees a high probability of meeting the time constraints suffices.

Average Case Performance Some embedded systems do not have real-time constraints. In this case, typically the *average case* performance of the system is stated. The performance of the system based on a small set of test-runs is evaluated and it is used to represent the overall performance of the system. Few or no guarantees are made on the variance of the performance. A typical example is a printer, whose average speed is often stated in pages per minute.

The coverage of this paper is somewhat biased towards extreme case performance analysis since this has been the focus of most of the research in this area; this is an indication of its challenging nature.

3 Applications of Performance Analysis

Design Validation The most direct application of performance analysis is design validation, i.e. ensuring that the design meets the specifications. As highlighted in Section 2 these performance specifications may take the form of hard/soft real-time constraints or average case constraints.

Design Decisions and System Optimization Embedded systems generally have a set of tasks that can be implemented either in hardware using ASICs or FPGAs, or in software running on one or more processors. Performance estimates for these tasks on different targets are used to decide this mapping. In the simplest case, with only a single known processor and single hardware resource, this may reduce to deciding the hardware software partition (e.g. [4]). With additional processor resources available, this impacts the selection of processors, and mapping of tasks to different processors. A tighter estimation allows the use of a slower processor without violating any real-time constraints, thus lowering the system cost. Performance estimates may be used to optimize other system parameters such as cache and buffer sizes.

Real-Time Schedulers All real-time schedulers need to use performance bounds for different tasks to guarantee system deadlines. Loose estimates may lead to the inability to guarantee deadlines, or the poor utilization of hardware resources. Real-time scheduling is an area of active research in the real-time community. Surveys for uniprocessor scheduling have been presented by Sha et al. [22] and for multiprocessor scheduling by Shin et al. [24] and Ramamrithan et al. [20].

Compiler Optimization Performance analysis techniques may be used to guide compiler optimizations to improve software performance. As an example, Ghosh et al. [3] use analytical techniques to determine the number of data reference cache misses in loops. This is then used to modify the data layout in memory by either changing the array offset or padding arrays.

4 Analysis Components

Performance analysis must deal with a number of distinct, though not necessarily independent, sub-problems. In this section, we examine these and in each case provide a summary of the techniques proposed in the literature to deal with that aspect of the analysis problem. In most cases, the body of work available is too large to be exhaustively cited, our references are intended to point to representative work.

4.1 Path Analysis

Worst-case analysis is in general undecidable since it is equivalent to the halting problem. To make this problem decidable, the program must meet certain restrictions [19]. These restrictions are:

- all loop statements must have bounded iterations, i.e., they cannot loop forever
- there are no recursive function calls
- there are no dynamic function calls

The execution time of a given program depends on the actual instruction trace (or program path) that is executed. Determining the set of program paths to be considered is a core component of any analysis technique. This can be further broken down into the following sub-components, each of which has been the focus of research attention.

4.1.1 Branch and Loop Analysis

For straight line code there is exactly one execution path to consider. Complexity creeps in only in the presence of control flow constructs such as branches and loops. These can result in an exponential blowup of the number of possible execution paths and are thus computationally challenging. Researchers have used a variety of different techniques to deal with this depending on the performance metric being considered.

General Heuristics For probabilistic or average case analysis, general heuristics based on "typical" program statistics can be used. Such heuristics include, for example, the observation that most backward branches are taken, and most forward branches are not taken.

Profile Directed Specific statistics can be collected for a given application by considering a sample data set and using profiling information to determine the actual branch decisions and loop counts. Again this can be used only in probabilistic and average case analysis.

```
    if (ok)
S1    i = i*i + 1;  /* i is non-zero! */
    else
S2    i = 0;

    /* ... */

    if (i)
S3    j++;
    else
S4    j = j*j;
```

Figure 1: Different parts of the code are sometimes related.

Symbolic Data Flow Analysis In certain cases it may be possible to determine the conditionals in branch statements and loop iteration statements by symbolic data flow analysis techniques, similar to those used in program verification, e.g. the work by Rustagi and Whalley [21]. However, this has very limited application due to the intractability of the problem.

Extreme Case Selection In worst case (best case) analysis, a straightforward approach is to always assume the worst case (best case) choice is made for each branch and loop. For example, in Shaw's simple timing schema approach [23], for an `if-then-else` statement, the execution times of the true and false statements are compared and the larger one taken for worst case estimation. Consider the example shown in Figure 1. S_1 and S_3 are always executed together, and so are S_2 and S_4. But if the above method is used, statements S_1 and S_4 will be selected for worst case analysis. These two statements are never executed together in practice and the above method results in loose estimation. Such path relationships occur frequently in programs and it is important to provide some mechanism for obtaining this information.

Puschner and Koza [19] as well as Mok et al. [15] extend this approach to allow the programmer to provide simple execution count information of certain statements. This permits non-pessimistic choices locally. This is helpful in specifying the total execution count of the loop body in a nested loop, where the number of loop iteration of the inner loop depends on the loop index of the outer loop. However, this still suffers from the problem that relationships between different parts of the program may not be exploited.

Path Enumeration In order to capture the relationship between different parts of the program, some form of path enumeration may be used. This must be a partial enumeration, since the number of program paths is typically exponential in the size of the program. For extreme case analysis, this partial enumeration must be pessimistic, i.e. it must include paths that bound the extreme case behavior even if they are never actually exercised. In his work Park [18] observed that that all statically feasible execution paths can be expressed by regular expressions. For example, the following equations show the regular expression of the `if-then-else` statement and that of the `while` loop statement with loop bound n respectively.

$$\begin{aligned} \texttt{if } B \texttt{ then } S_1 \texttt{ else } S_2 \quad &: \quad B \cdot (S_1 + S_2) \\ \texttt{while } B \texttt{ do } S \quad &: \quad B \cdot (S \cdot B)^n \end{aligned}$$

In his work, the set of statically feasible execution paths is represented by a regular expression A_p. The user can provide path information by using a script language called IDL (*information description language*), which is subsequently translated into another regular expression denoted as I_p. The intersection of A_p and I_p, denoted as $A_p \cap I_p$, represents all feasible execution paths of the program. The best case and worst case execution paths, and their corresponding execution times can be then determined from the regular expression $A_p \cap I_p$. Typical path information supported by IDL includes:

- two statements are always executed together,
- two statements are mutually exclusive,
- a statement is executed a certain number of times.

The use of IDL is a vast improvement over earlier methods. Simple path relationships can be expressed. However, the main drawback of this approach is that the intersection of A_p and I_p is a complicated and expensive operation. To simplify this operation, (i) the user can only expresses path information in IDL instead of general regular expressions simplifying the format of I_p (ii) pessimistic approximations are used in the intersection operation. These limit the accuracy of path analysis.

Bounding Techniques In the `cinderella` project [12], an alternative attack on the problem is used. Instead of determining the actual set of paths to be considered, feasible paths are determined in terms of bounds on the execution counts of various basic blocks. These are then used in an integer-linear programming formulation to determine the extreme case execution times.

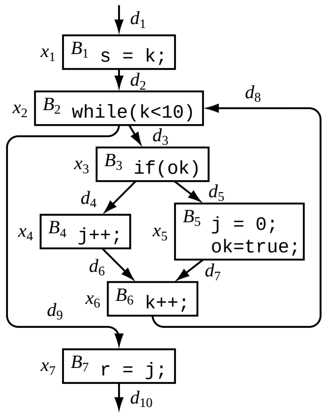
Let x_i be the execution count of a basic block B_i, and c_i be the execution time of the basic block. If there are N basic blocks in the program, then the total execution time of the program is given as:

$$\text{Total execution time} = \sum_{i}^{N} c_i x_i. \qquad (1)$$

The possible values of x_i's are constrained by the program structure and the possible values of the program variables. These are expressed as linear constraints divided into two parts: (i) *structural constraints*, which are derived automatically from the program's control flow graph (CFG) [1], and (b) *functionality constraints*, which are provided by the user to specify loop bounds and other path information. The construction of these constraints is illustrated by an example shown in Fig. 2, in which a conditional statement is nested inside a `while` loop. Fig. 2(b) shows the CFG. A basic block execution count, x_i, is associated with each node. Each edge in the CFG is labeled with a variable d_i which serves both as a label for that edge and as a count of the the number of times that the program control passes through that edge. Analysis of the CFG is equivalent to a standard network-flow problem. Structural constraints can be derived from the CFG from the fact that, for each node B_i, its execution count is equal to the number of times that the control enters the node (inflow), and is also equal to the number of times that the control exits the node (outflow). The structural constraints do not provide any loop bound information. This information can be provided by the user as a functionality constraint. In this example, we note that since `k` is positive before it enters the loop, the loop body will be executed between 0 and 10 times each time the loop is entered. The constraints to specify this information are:$0x_1 \leq x_3 \leq 10x_1$. The functionality constraints can also

```
/* k >= 0 */
s = k;
while (k < 10) {
  if (ok)
    j++;
  else {
    j = 0;
    ok = true;
  }
  k++;
}
r = j;
```

(a) Code



(b) Control flow graph

Figure 2: An example showing how the structural and functionality constraints are constructed.

be used to specify other constraints on execution paths. For example, we observe that the `else` statement (B_5) can be executed at most once inside the loop. This information can be specified as: $x_5 \leq 1x_1$. These constraints form the input to an integer linear programming formulation. More complicated path information can also be specified, and this mechanism has been shown to be more powerful than Park's IDL [18] in describing path information [11].

4.1.2 Interrupt Analysis

Somewhat related to branch and loop analysis is the issue of interrupt analysis for preemptive execution, since it also alters the flow of control. Limited work here has focussed on studying the cache behavior of preemptive execution in multi-tasking systems, i.e., determining the worst-case points for interrupts during program execution with respect to impact on the state of the cache [10]. However, in the experimental study, there is little variation in the execution time with varying interrupt points. This points to the use of simple analysis techniques in practice that do not depend on locating the interrupt points.

4.2 Utilization of System Resources

Significant variations in program execution time can result from varying uses of system resources. The way the program references memory or occupies pipeline resources can have significant impact on overall program and system performance. In this section we discuss analysis techniques for estimating or bounding the utilization of different types of systems resources.

4.2.1 Microarchitectural Resources

Some of the key performance factors in systems today revolve around the program's utilization of the processor's microarchitectural resources. Early works [15, 19, 18] in this area assumed a very simple microarchitecture, such as the Motorola M68000 microprocessor, where the instruction execution times are assumed to be constant and independent of each other. The effects of pipelines and cache memories are not considered. This simplifies the determination of estimated bound as microarchitecture modeling can be performed independently before program path analysis.

With fairly complex superscalar pipelines becoming more common even in embedded processors, simple processor models often no longer suffice for estimating or bounding program performance. To be useful, processor performance models must consider the utilization of individual functional units and the issue rate down the processor pipelines. Pipelines are relatively easy to model and they have been studied extensively. Specific cases have been studied by Bharrat and Jeffay [2] (Cypress CY7C611 SPARC microprocessor), Zhang et al. [27] (Intel 80C188 processor's two stage pipeline) and Li et al. [12] (Intel i960KB). Narasimhan and Nilsen [17] and Harmon [5] present various retargetable pipeline modeling methods. The above pipeline modeling methods model the pipeline states within a short straight line sequence of instructions, such as a basic block [1]. Hur et al. [8] (MIPS R3000) and Healy et al. [6] (Micro-SPARC I) have considered pipeline effects across the branches and loops. It is generally felt that analysis using limited code length sequences is fairly accurate.

4.2.2 Memory Behavior

With processors speeds improving at a much faster rate than memory speeds, the relative importance of memory behavior on program performance has increased significantly in recent years. In response to this, researchers have explored several ways of producing estimates, or bounds, on program memory performance.

Simulation One of the most common mechanisms for evaluating program memory behavior on different platforms is via simulation. Here, traces of memory referencing behavior are fed into a simulator of a particular memory hierarchy organization. Timing information and other cache statistics are tracked and can be used to guide program performance tuning [9, 14] or architectural choices.

Renewal Theory Models While simulation can provide detailed views of program memory behavior, it suffers from the drawback that it can be quite slow. To avoid this, researchers have considered techniques for generating performance statistics based on samples of memory reference traces. Since the cache state is unknown at the beginning of each sample, renewal theory models have been developed for analyzing memory behavior based on this partial information [26]. For each cache set, Wood et al.'s model breaks time in "live time", where the block will be referenced again before it is replaced, and "dead time", where the block will not be referenced again before it is replaced. They show that the expected miss rate for each block is equal to the dead time divided by the total time. This result can be used to improve the accuracy of sampled cache performance simulations.

Locality Analysis Locality analysis is widely used by compiler writers to offer good estimates of memory behavior in loop-oriented scientific codes [25]. The analysis relies on computing a set of reuse vectors that summarize how the loop's array accesses reference (and re-reference) memory locations and cache lines. Subsequent work has built on reuse analysis to guide memory prefetching algorithms [16].

CM Equations Further building on ideas from locality analysis, recent research has developed the *cache-miss* (CM) equations, which give a detailed representation of the cache misses in loop-oriented scientific code [3]. Linear Diophantine equations are generated to summarize each loop's memory behavior. Mathematical techniques for manipulating

Diophantine equations allow us to compute the number of possible solutions, where each solution corresponds to a potential cache miss. These equations provide a general framework to guide code optimizations, such as array padding or data offsetting, for improving cache performance.

Bounding Techniques The `cinderella` project integrates the bounding techniques used in path analysis with cache modeling. A brief summary of direct-mapped instruction cache modeling is provided here, details on other cache types may be found in [13].

An *l-block* is defined as a contiguous sequence of instructions within the same basic block that are mapped to the same cache set in the i-cache, and thus have identical hit/miss behavior. Suppose a basic block B_i is partitioned into n_i l-blocks, denoted as $B_{i.1}$, $B_{i.2}$, ..., $B_{i.n_i}$. The hit and miss execution times of the l-block, are represented by $c_{i.j}^{hit}$ and $c_{i.j}^{miss}$ respectively. Let $x_{i.j}^{hit}$ and $x_{i.j}^{miss}$ be integer variables that represent an l-block $B_{i.j}$'s hit and miss counts, then the total execution time of the program can be defined as:

$$\text{Total execution time} = \sum_{i}^{N} \sum_{j}^{n_i} (c_{i.j}^{hit} x_{i.j}^{hit} + c_{i.j}^{miss} x_{i.j}^{miss}). \quad (2)$$

Since l-block $B_{i.j}$ is inside the basic block B_i, its total execution count is equal to x_i. Hence

$$x_i = x_{i.j}^{hit} + x_{i.j}^{miss}, \qquad j = 1, 2, \ldots, n_i \quad (3)$$

Equation (3) links the new cost function (2) with the program structural constraints and the program functionality constraints, both of which remain unchanged. In addition, the cache activities can now be specified in terms of the new variables $x_{i.j}^{hit}$'s and $x_{i.j}^{miss}$'s.

For any two l-blocks mapped to the same cache set, they *conflict* with each other if their address tags [7] are different. Otherwise, they are said to be *non-conflicting*. When a cache set contains two or more conflicting l-blocks, the hit/miss counts of all the l-blocks mapped to this set will be affected by the sequence in which these l-blocks are executed (and not by the execution of any other l-blocks) This leads to the abstraction of the control flow of the l-blocks mapped to that particular cache set in the form of a *cache conflict graph*.

A cache conflict graph (CCG) is constructed for every cache set containing two or more conflicting l-blocks. It contains a start node 's', an end node 'e', and a node '$B_{k.l}$' for every l-block $B_{k.l}$ mapped to the same cache set. The start node represents the start of the program, and the end node represents the end of the program. For every node '$B_{k.l}$', a directed edge is drawn from node $B_{k.l}$ to node $B_{m.n}$ if there exists a path in the CFG from basic block B_k to basic block B_m *without* passing through any other l-blocks of the same cache set.

Like the control flow graph, some linear constraints can be derived from the CCG. For each edge from node $B_{i.j}$ to node $B_{u.v}$, the variable $p_{(i.j,u.v)}$ counts the number of times that the control passes through that edge. At each node $B_{i.j}$, the sum of control flow going into the node must be equal to the sum of control flow leaving the node, and it must also be equal to the total execution count of l-block $B_{i.j}$. Therefore, two constraints are constructed at each node $B_{i.j}$:

$$x_i = \sum_{u.v} p_{(u.v,i.j)} = \sum_{u.v} p_{(i.j,u.v)}, \quad (4)$$

where '$u.v$' may also include the start node 's' and the end node 'e'. Note that due to the existence of x_i's, this set of constraints is linked to the structural and functionality constraints.

The program is executed once, so at the start node:

$$\sum_{u.v} p_{(s,u.v)} = 1. \quad (5)$$

The number of cache hits is determined by:

$$p_{(i.j,i.j)} \leq x_{i.j}^{hit} \leq p_{(s,i.j)} + p_{(i.j,i.j)}. \quad (6)$$

Equations (2) through (6) are the cache constraints for a direct mapped i-cache. These constraints, together with (3), the structural constraints and the functionality constraints, are used by the ILP solver with the goal of maximizing the cost function (2).

To first order, program path analysis primarily focuses on understanding embedded software characteristics, while microarchitecture utilization primarily has a hardware focus. Despite this hardware/software division, there is a strong coupling between these two areas. This is especially true for extreme-case analysis. For example, you need to know the worst-case execution path to determine the pipeline utilization and cache misses. On the other hand, since they impact the execution time, you need to know the cache misses and pipeline stalls in order to determine the worst case path. This mutual dependency makes this analysis problem hard. The majority of research has focussed on either modeling the microarchitecture, or on determining worst case paths. Most efforts fail to handle both the tasks well. The `cinderella` project [12] does manage to deal with both aspects using a single uniform ILP based bounding technique.

4.3 Input Characterization

Input data has a significant impact on software performance. However, it is an area that has had little study. For extreme-case performance the input space is implicitly assumed to be the entire set of possible inputs. However, not all of these inputs may be generated by the environment. This is analogous to the case of *don't cares* in logic design. How do we specify these don't cares in a succinct manner? More importantly, how can we then use them in analysis?

In probabilistic analysis we need to specify the distribution of inputs. Again, there are no easy mechanisms for describing this. Typically this reduces to a large set of input samples. The analysis then reduces to analyzing the behavior for each individual sample and then summarizing the results. More desirable and efficient would be a succinct description of the input space and a single analysis step capable of exploiting this information.

The situation in average case analysis is probably the most acceptable. Using a small set of typical inputs works well as long as these are truly representative of the input data. However, determining representative data may be non-trivial for complex functions, e.g. how would you determine a representative set of pictures for a JPEG compression program?

It is clear that in all these cases input characterization is tied to the ability of analysis techniques to use this information – that is where we need to seek solutions for these problems.

5 Conclusions

Overall, this paper has explored the key issues in static timing analysis of embedded software. This area has brought up a rich variety of research problems spanning stochastic analysis, compiler techniques, and hardware modeling. While current techniques offer several effective alternatives for estimating and bounding program performance, there are several significant avenues for future work. In particular, more accurately considering the effects of interrupts, varying input data, and newer dynamic pipelines, will all greatly extend the application of these static performance tools.

REFERENCES

[1] A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers Principles, Techniques, and Tools.* Addison-Wesley, 1986. ISBN 0-201-10194-7.

[2] S. J. Bharrat and K. Jeffay. Predicting worst case execution times on a pipelined RISC processor. Technical report, Department of Computer Science, University of North Carolina at Chapel Hill, April 1994. TR94-072.

[3] S. Ghosh, M. Martonosi, and S. Malik. Cache Miss Equations: An Analytical Representation of Cache Misses. Proc. 1997 International Conference on Supercomputing, July 1997.

[4] R. K. Gupta and G. D. Micheli. Hardware-software cosynthesis for digital systems. *IEEE Design and Test of Computers*, pages 29–41, September 1993.

[5] M. G. Harmon. *Predicting Execution Time on Contemporary Computer Architectures.* PhD thesis, The Florida State University, April 1991.

[6] C. A. Healy, D. B. Whalley, and M. G. Harmon. Integrating the timing analysis of pipelining and instruction caching. In *Proceedings of 16th IEEE Real-Time Systems Symposium*, pages 288–297, December 1995.

[7] J. L. Hennessy and D. A. Patterson. *Computer Architecture: A Quantitative Approach, Second Edition.* Morgan Kaufmann Publishers, Inc., 1996. ISBN 1-55860-329-8.

[8] Y. Hur, Y. H. Bae, S.-S. Lim, S.-K. Kim, B.-D. Rhee, S. L. Min, C. Y. Park, M. Lee, H. Shin, and C. S. Kim. Worst case timing analysis of RISC processors: R3000/R3010 case study. In *Proceedings of 16th IEEE Real-Time Systems Symposium*, pages 308–319, December 1995.

[9] A. R. Lebeck and D. A. Wood. Cache Profiling and the SPEC Benchmarks: A Case Study. *IEEE Computer*, pages 15–26, Oct. 1994.

[10] C.-G. Lee, J. Hahn, Y.-M. Seo, S. L. Min, R. Ha, S. Hong, C. Y. Park, M. Lee, and C. S. Kim. Analysis of cache-related preemption delay in fixed-priority preemptive scheduling. In *Proceedings of the IEEE Real-Time Systems Symposium*, pages 264–274, December 1996.

[11] Y.-T. S. Li. *Performance Analysis of Embedded Software.* PhD thesis, Princeton University, 1997. In preparation.

[12] Y.-T. S. Li, S. Malik, and A. Wolfe. Efficient microarchitecture modeling and path analysis for real-time software. In *Proceedings of 16th IEEE Real-Time Systems Symposium*, pages 298–307, December 1995.

[13] Y.-T. S. Li, S. Malik, and A. Wolfe. Cache modeling for real-time software: Beyond direct mapped instruction caches. In *Proceedings of 17th IEEE Real-Time Systems Symposium*, pages 254–263, December 1996.

[14] M. Martonosi, A. Gupta, and T. Anderson. Tuning Memory Performance in Sequential and Parallel Programs. *IEEE Computer*, pages 32–40, Apr. 1995.

[15] A. K. Mok, P. Amerasinghe, M. Chen, and K. Tantisirivat. Evaluating tight execution time bounds of programs by annotations. In *Proceedings of the 6th IEEE Workshop on Real-Time Operating Systems and Software*, pages 74–80, May 1989.

[16] T. C. Mowry, M. S. Lam, and A. Gupta. Design and Evaluation of a Compiler Algorithm for Prefetching. In *Proc. Fifth Intl. Conf. on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 62–73, Oct. 1992.

[17] K. Narasimhan and K. Nilsen. Portable execution time analysis for RISC processors. In *Proceedings of ACM PLDI Workshop on Language, Compiler, and Tool Support for Real-Time Systems*, pages L1–L10, June 1994.

[18] C. Y. Park. *Predicting Deterministic Execution Times of Real-Time Programs.* PhD thesis, University of Washington, Seattle 98195, August 1992.

[19] P. Puschner and C. Koza. Calculating the maximum execution time of real-time programs. *The Journal of Real-Time Systems*, 1(2):160–176, September 1989.

[20] K. Ramamritham, J. A. Stankovic, and P.-F. Shiah. Efficient scheduling algorithms for real-time multiprocessor systems. *IEEE Transactions on Parallel and Distributed Systems*, pages 184–194, April 1990.

[21] V. Rustagi and D. B. Whalley. Calculating minimum and maximum loop iterations. Technical report, Computer Science Department, Florida State University, May 1994.

[22] L. Sha, R. Rajkumar, and S. S. Sathaye. Generalized rate-monotonic scheduling theory: A framework for developing real-time systems. In *Proceedings of the IEEE*, pages 68–82, January 1994.

[23] A. C. Shaw. Reasoning about time in higher-level language software. *IEEE Transactions on Software Engineering*, 15(7):875–889, July 1989.

[24] K. G. Shin and P. Ramanathan. Real-time computing: A new discipline of computer science and engineering. In *Proceedings of the IEEE*, pages 6–24, January 1994.

[25] M. E. Wolf and M. S. Lam. A Data Locality Optimization Algorithm. In *Proc. SIGPLAN '91 Conf. on Programming Language Design and Implementation*, pages 30–44, June 1991.

[26] D. A. Wood, M. D. Hill, and R. E. Kessler. A Model for Estimating Trace-Sample Miss Ratios. In *Proc. ACM SIGMETRICS Conf. on Measurement and Modeling of Computer Systems*, pages 79–89, June 1991.

[27] N. Zhang, A. Burns, and M. Nicholson. Pipelined processors and worst-case execution times. *Journal of Real-Time Systems*, October 1993.