Parametric Timing Analysis

EMILIO VIVANCOS[1], CHISTOPHER HEALY[2], FRANK MUELLER[3], AND DAVID WHALLEY[4]

[1] *Departamento de Sistemas Informaticos y Computacion, Universidad Politecnica de Valencia, 46022-Valencia, Spain*
e-mail: vivancos@dsic.upv.es, phone: (+34) 96 387-7354

[2] *Computer Science Department, Furman University, Greenville, SC 29613, U.S.A.*
e-mail: chris.healy@furman.edu, phone: (864) 294-2233

[3] *Center for Applied Scientific Computing, Lawrence Livermore National Laboratory, P.O. Box 808, L-561, Livermore, CA 94551*
e-mail: frank.mueller@llnl.gov, phone: (925) 424-3642

[4] *Computer Science Department, Florida State University, Tallahassee, FL 32306-4530, U.S.A.*
e-mail: whalley@cs.fsu.edu, phone: (850) 644-3506

ABSTRACT

Embedded systems often have real-time constraints. Traditional timing analysis statically determines the maximum execution time of a task or a program in a real-time system. These systems typically depend on the worst-case execution time of tasks in order to make static scheduling decisions so that tasks can meet their deadlines. Static determination of worst-case execution times imposes numerous restrictions on real-time programs, which include that the maximum number of iterations of each loop must be known statically. These restrictions can significantly limit the class of programs that would be suitable for a real-time embedded system. This paper describes work-in-progress that uses static timing analysis to aid in making dynamic scheduling decisions. For instance, different algorithms with varying levels of accuracy may be selected based on the algorithm's predicted worst-case execution time and the time allotted for the task. We represent the worst-case execution time of a function or a loop as a formula, where the unknown values affecting the execution time are parameterized. This *parametric timing analysis* produces formulas that can then be quickly evaluated at run-time so dynamic scheduling decisions can be made with little overhead. Benefits of this work include expanding the class of applications that can be used in a real-time system, improving the accuracy of dynamic scheduling decisions, and more effective utilization of system resources.

1. INTRODUCTION

Software is being embedded as a component of an increasing number of critical systems. Often embedded systems have timing constraints that must be met or the system is not considered functional. Computations that are less precise and on-time are often considered better than more precise computations that are late. This type of system is referred to as real time. In order to verify that real-time systems will meet their deadlines, designers require that the worst-case execution time (WCET) of each task in a real-time system be known. The process of automatically and statically determining the WCET of a program or task is called *timing analysis*. Scheduling decisions are based on each task's WCET and the total time in the schedule. Many requirements are often imposed on real-time programs so that the WCET may be predicted. One of these requirements is that the maximum number of iterations of all loops be known statically. Unfortunately, this requirement significantly limits the class of programs that can be performed in a real-time embedded system.

This paper describes how static timing analysis can be used to aid in making dynamic scheduling decisions. The WCET of a function or a loop is represented as a formula, where the values affecting the execution time are parameterized. Such formulas can then be quickly evaluated at run-time so dynamic scheduling decisions can be made when scheduling a task or choosing algorithms within a task. Benefits of this parametric timing analysis include expanding the class of applications that can be used in a real-time system, improving the accuracy of dynamic scheduling decisions, and more effective utilization of system resources.

2. RELATED WORK

Recently, a number of research groups have addressed various issues in the area of predicting the WCET of real-time programs. Conventional methods for static analysis have been extended from unoptimized programs on simple CISC processors [PuK89, Par93, HBW92] to optimized programs on pipelined RISC processors [ZBN93, LBJ94, HWH95], from uncached architectures to instruction caches [AMW94, LMW95, HBL95] and data caches [KMH96, LMW96, WMH97], and from assuming that all paths are feasible to automatically detecting constrains on paths [HeW99, EnE00]. All of these methods obtain discrete values to bound the WCET in a non-parametric fashion.

Chapman *et al.* [CBW96] used path expressions to combine a source-oriented parametric approach of WCET analysis with timing annotations, verifying the latter through the former. Bernat *et al.* [BeB00] also proposed using algebraic expressions to represent the WCET of subprograms, where the algebraic expression is parameterized by some of the subprogram's parameters. These approaches are closely related to our work in the sense that we also utilize a parametric approach for WCET prediction. However, our approach differs in a number of ways. Instead of a source-level analysis that cannot easily take architectural details of a processor into account, we perform our analysis within an existing framework of tools operating at a low-level intermediate code, which is equivalent to machine code. The other approaches rely on a computational algebra system, like Mathematica or Maple, to evaluate the algebraic expression representing the WCET with the goal of making more accurate predictions. However, relying on a general computational algebra system to calculate the WCET to make dynamic scheduling decisions in real-time would be unrealistic. We instead emphasize the generation of simple formulas where the WCET can be quickly calculated at run-time. Thus, we can retain the parametric model beyond static timing analysis for a multitude of applications ranging from scheduling of tasks to selection of algorithms within a task.

3. NUMERIC TIMING ANALYSIS

Figure 1 depicts the overall organization of the existing numeric timing analysis environment before it was modified to support parametric analysis. We denote this environment as numeric since the timing analyzer must produce a number representing the WCET of a program (*i.e.*, the number of iterations for each loop must be a number that is known to the timing analyzer). An optimizing compiler [BeD88] was modified to produce control flow and branch constraint information as a side effect of the compilation of a file [AMW94, HSR98, HeW99]. A static cache simulator uses the control flow information to construct a control-flow graph of the program that consists of the call graph and the control flow of each function. The program control-flow graph is then analyzed and a caching categorization for each instruction in the program is produced [AMW94]. A separate categorization is given for each loop level in which the instructions and data references are contained. Next, a timing analyzer uses the control flow and constraint information, caching categorizations, and machine dependent information (e.g. pipeline characteristics) as input to make timing predictions [AMW94, HWH95]. Given a program's control-flow information and instruction caching categorizations along with the processor's instruction set information, the timing analyzer then derives best-case and worst-case estimates for each path, loop, and function within the program. A timing analysis tree is constructed, where each node of the tree corresponds to a loop or function in the function instance graph. Each node is considered a natural loop, where each function is treated as a loop executing a single iteration. The nodes in the timing analysis tree are processed in a bottom-up manner. In other words, the WCET and BCET for a node are not calculated until the times for all of its immediate child nodes are known. This means that the timing analyzer determines the execution time for programs by first analyzing the innermost loops and functions, and proceeding to higher level loops and functions until it reaches `main()`. Finally, a graphical user interface is invoked that allows a user to request predictions for portions of the program [KHR96].

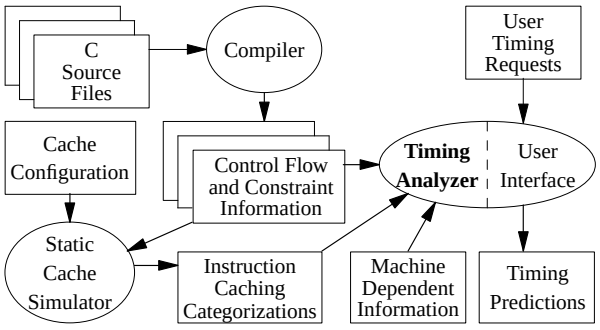

Figure 1: Overview of Timing Analysis Process

Our numeric timing analyzer requires that the number of iterations of each loop be known, that the total number of iterations for a nonrectangular loop nest can be calculated, or that a user provide this information when it is unknown. Figure 2 depicts an abstraction of the loop analysis algorithm that is used. The algorithm repeatedly selects the longest path through the loop until a fixed point is reached (*i.e.*, the caching behavior does not change and the cycles for the worst-case path is guaranteed to remain the same). The maximum number of times that a worst-case path is selected is the minimum of m+1 or n, where m is the number of unique paths through the loop and n is the number of loop iterations. Note that the WCETs for inner loops are predicted before outer loops, an inner loop is treated as being a single node in a path of the next level outer loop, and the control flow is partitioned if the number of paths within a loop exceeds a specified limit [AlY97]. The correctness of the fixed-point algorithm is discussed in more detail elsewhere [AMW94].

```
cycles = curr_iter = 0.
DO
   curr_iter += 1.
   Find the longest path wcpath.
   cycles += wcpath->cycles.
WHILE wcpath's caching behavior can change
      && curr_iter < n
cycles += wcpath->cycles * (n - curr_iter).
```

Figure 2: Numeric Loop Analysis Algorithm

4. PARAMETRIC TIMING ANALYSIS

It would be desirable to support parametric timing predictions when the number of iterations for a loop is unknown until run time. Our goal is to calculate a formula (or closed form) for the WCET for such a loop, where the formula depends on n, the number of iterations for that loop. This formula needs to be relatively inexpensive to calculate since it will be used at run-time to make dynamic decisions concerning the selection of tasks and/or the selection of algorithms within a task. Rather than always passing a numeric value representing the number of cycles of loops and functions up the timing analysis tree, we can instead pass up a symbolic formula when the number of iterations is unknown in a loop representing a node in the subtree.

Figure 3 shows an abstraction of the revised loop analysis algorithm. The algorithm iterates until a fixed point is reached where the caching behavior of the WCET path does not change. The base cycles that are applied before the final worst-case path time is known are saved. Equation 1 shows that the WCET of a loop depends on these base cycles and the WCET path time, which are both constants, and the number of iterations, which is unknown until run time.

```
cycles = curr_iter = 0.
DO
   curr_iter += 1.
   Find the longest path wcpath.
   cycles += wcpath->cycles.
WHILE wcpath's caching behavior can change
base_cycles =
   cycles - (wcpath->cycles * curr_iter).
```

Figure 3: Parametric Loop Analysis Algorithm

$$WCET\_loop\_time = WCET\_path\_time * n + base\_cycles \quad (1)$$

If the actual number of iterations exceeds the number of iterations that are required to reach the point where the parametric WCET loop time is calculated, then the parametric result should be comparable to that produced by the numeric timing analyzer when there is a known number of iterations. If the actual number of iterations is less than the number of iterations that are required to reach this point, then there could be an overestimation. For instance, consider the situation as depicted in Table 1. The formula representing the parametric *WCET_loop_time* would 11**n*+16 since the *WCET_path_time* after the caching behavior does not change is 11 cycles and the base cycles is 16 (20-11 + 18-11). If the loop iterates only once, then the predicted WCET would be 27 cycles, while the actual WCET would be 20 cycles. We could have modified the formula to address dealing with the special cases for having fewer iterations. However, the resulting formula would be more complex and the additional cost of

making these checks at run time would likely outweigh the benefit obtained for a more accurate prediction. Furthermore, if the number of iterations is small, then the loop will likely have little impact on the total WCET. If the number of iterations is large, then there should be very little sacrifice in the accuracy of the predictions.

How the Path Is Evaluated	Path 1	Path 2
Initial Execution Including Cache Misses	20	18
After Caching Behavior Does Not Change	11	9

Table 1: WCET Path Information for a Hypothetical Loop

The timing analyzer processes inner loops before outer loops. A nested inner loop is represented as a single block when processing a path in the outer loop. Rather than representing the inner loop with a number of cycles, we instead represent the loop with a symbolic formula when the number of iterations is not statically known. The time for the outer loop path is simply the symbolic sum of the cycles associated with the rest of the path and the formula representing the inner loop.

The analysis becomes more complicated when one or more paths in a loop contain a nested loop with a parametric WCET. Consider the following example shown in Figure 4. There are two loops, where an inner loop (block 4) is nested in the outer loop (blocks 2, 3, 4, and 5). Assume the number of iterations for the inner loop is symbolic. The loop analysis algorithm requires that the timing analyzer find the longest path in the outer loop, which may depend on the number of iterations of the inner loop. The minimum number of iterations for a loop is one since our definition for the number of loop iterations is the number of times that the loop header (entry block of the loop) is executed. If the WCET for path A(**2**→**3**→**5**) is less than the WCET for path B(**2**→**4**→**5**) when the number of iterations of the inner loop is one, then path B is chosen. Otherwise, a *max()* function must be used to represent the parametric WCET of the outer loop. For instance, equation 2 shows that the maximum of the two paths should be selected. Note that the WCET of these paths is after the caching behavior reaches a steady state and that the base cycles are the extra cycles before both of these these paths reach that state. The first value passed to *max* for the example in Figure 4 would be numeric and the second would be symbolic.

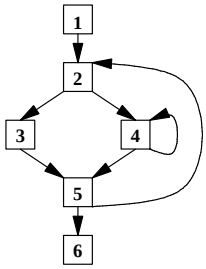

Figure 4: Example of an Outer Loop with Multiple Paths

$$WCET\_loop\_time = max(WCET\_path\_A\_time, WCET\_path\_B\_time) * n + base\_cycles \quad (2)$$

We have developed a preliminary implementation of the parametric timing analyzer. As in the numeric timing analyzer, certain restrictions are imposed. First, recursive programs are not allowed since cycles in the call graph would complicate the generation of unique function instances. Second, indirect calls are not allowed since an explicit call graph must be generated statically. Finally, loops must be structured, which means there must be only a single entry point in the loop. Unlike the numeric timing analyzer, the parametric timing analyzer does not need to know a constant value for the maximum number of iterations of each loop. However, the compiler must be able to generate a symbolic expression that represents the number of loop iterations to be executed at run-time.

Table 2 describes a small set of test programs that were used to contrast the results with numeric and parametric timing analysis. These test programs were all relatively simple since this preliminary implementation does not yet address the case where the worst-case path within a loop cannot be selected due to two or more of the loop paths having a symbolic WCET.

Program	Description
Matcnt	Counts and sums the nonnegative elements of an integer matrix.
Matmul	Multiplies 2 integer matrices and stores the result in a third integer matrix.
Stats	Calculates the sum, mean, variance, and standard deviation and the linear correlation coefficient between two vectors of numbers.
Summinmax	Sum the minimum and maximum of the corresponding elements of two integer vectors.
Sumnegpos	Sums the negative, positive, and all elements of an integer vector.

Table 2: Test Programs

Table 3 shows the results of predicting execution time using the two types of techniques. For these programs we predicted pipeline and instruction cache performance. *Formula* is the formula returned by the parametric timing analyzer and it represents the parameterized predicted execution time of the program. In order to evaluate the accuracy of the parametric timing analysis approach, we made each of the loops in these test programs iterate the same number of times. Thus, *n Iters* represents the number of loop iterations for each loop in the program and *n* represents that value in the formulas. The power of *n* represents the loop nesting level and the factor represents the cycles spent at that level. Note that most of the programs had multiple loops at each nesting level. For example, $160n^2$ represents that 160 cycles is the sum of the cycles that would occur in a single iteration of all the loops at nesting level 2 in the program. If the number of iterations of two different loops in a loop nest differ, then the formula would reflect this as a multiplication of these factors. For instance, if the matrix in *Matcnt* had *m* rows and *n* columns, where $m \neq n$, then the formula would be $(160n+267)m+857$. The *Observed Cycles* were obtained by using an integrated pipeline and instruction cache simulator and this value represents the cycles in the execution using worst-case input data. The *Numeric Analysis* represents the results using the previous version of the timing analyzer, where the number of iterations of each loop must be a number that is known to the timing analyzer. The *Parametric Analysis* represents cycles where the number of iterations was unknown until run time. The *Estimated Cycles* and *Ratio* represent the predicted number of cycles by the timing analyzer and ratio to the *Observed Cycles*. The estimated parametric cycles were obtained by plugging the number of iterations into the formula returned by the

Program	Formula	n Iters	Observed Cycles	Numeric Analysis		Parametric Analysis	
				Estimated Cycles	Ratio	Estimated Cycles	Ratio
Matcnt	$160n^2+267n+857$	1	1,206	1,207	1.001	1,284	1.065
		10	18,954	19,523	1.030	19,527	1.030
		100	1,622,034	1,627,553	1.003	1,627,557	1.003
Matmul	$33n^3+310n^2+530n+851$	1	1,432	1,460	1.020	1,724	1.204
		10	62,182	70,137	1.128	70,151	1.128
		100	33,725,782	36,153,837	1.072	36,153,851	1.072
Stats	$1049n+1959$	1	2,885	2,912	1.009	3,008	1.043
		10	12,290	12,449	1.013	12,449	1.013
		100	106,340	106,859	1.005	106,859	1.005
Summinmax	$17n+123$	1	139	140	1.007	140	1.007
		10	283	293	1.035	293	1.035
		100	1,723	1,823	1.058	1,823	1.058
Sumnegpos	$13n+94$	1	104	107	1.029	107	1.029
		10	203	224	1.103	224	1.103
		100	1,193	1,394	1.168	1,394	1.168

Table 3: Results

parametric timing analyzer. The results show that the parametric analysis is less accurate when loops have a single iteration. This is due to the caching behavior of the WCET path not reaching a fixed point, as described in Figure 3. However, the parametric analysis is almost as accurate as the numeric analysis when the number of iterations increases.

5. USING PARAMETRIC WCET PREDICTIONS

One may ask how can a symbolic formula representing a parametric WCET for a loop or function actually be used. A scheduler can dynamically adjust priorities based on a parametric estimate of the WCET of a task. In addition, it would be beneficial to make dynamic decisions within a real-time task based on an estimate of the remaining execution time. Multiple versions of algorithms could be developed that solve the same problem with different degrees of precision and execution time. Parametric WCET prediction can potentially allow the selection of an algorithm with higher precision due to tighter WCET predictions. Likewise, parametric WCET analysis may be used to determine an appropriate number of iterations for algorithms that progressively improve the precision of calculations (*i.e.*, imprecise computations). The functions providing the parametric WCET can be provided as call-backs to the operating system. Once the parameters, such as the number of loop iterations, have been supplied by the application, the scheduler may inspect the time budget to determine how to schedule the remainder of a task's execution.

If code within a task is generated to represent the symbolic WCET formula during the parametric timing analysis, then it would seem that this code would affect the previously determined timing predictions due to changing the addresses of the instructions and potentially the caching behavior. Rather than inserting the code for the symbolic WCET formula directly into the source code at the point it is used, we decided to invoke a function that evaluates the symbolic WCET formula and we append these functions to the end of the program. We illustrate the approach with an example given in Figure 5, where the timing analysis is accomplished in stages as the parametric formulas are generated and later evaluated. In this example function 4 is generated by the timing analyzer to calculate the WCET of loop 3, whose number of iterations is unknown until run time.

(1) A call to a function is inserted that returns the WCET for a specified loop or a function based on a parameter indicating the number of loop iterations that becomes known at run time. The instructions that are associated with the call and that use the return value after the call are generated during the initial compilation. For instance, in Figure 5(a) function #1 calls function #4 to obtain the WCET of loop #3, which contains a symbolic number of iterations.

(2) The timing analyzer generates the source code for the called function into a separate file when processing the specified loop or function whose time needs to be calculated at run time. For instance, Figure 5(b) shows that after loop #3 has been parametrically analyzed, the code for function #4 has been generated. Note that the timing analysis tree representing the loops and functions in the program is processed in a bottom up fashion. The code in function 1 invoking the generated function is not evaluated until after the generated function is produced. The static cache simulator can initially assume that a call to an unknown function invalidates the entire cache. Figure 6 shows an example of the source code of such a generated function. This function returns the number of cycles associated with the formula given in Table 3 for the *Matcnt* program.

(3) The generated function is compiled and is placed at the end of the executable. The formula representing the symbolic WCET need not be simplified by the timing analyzer. Most optimizing compilers perform constant folding, strength reduction, and other optimizations that will automatically simplify the symbolic WCET produced by the timing analyzer. By placing the generated function after the rest of the program, the other instruction addresses in the program are unaffected. Likewise, the predicted caching behavior of the previously analyzed functions and loops are also unaffected since we are processing the tree in a bottom-up order.

(4) The timing analyzer is reinvoked to resume the analysis of the program, which includes calculating the WCET of the generated function and the code invoking such a function. For instance, Figure 5(c) shows that the generated function #4 has been numerically analyzed and Figure 5(d) shows that function #1 has been parametrically analyzed, which includes the numeric WCET required for executing function #4.

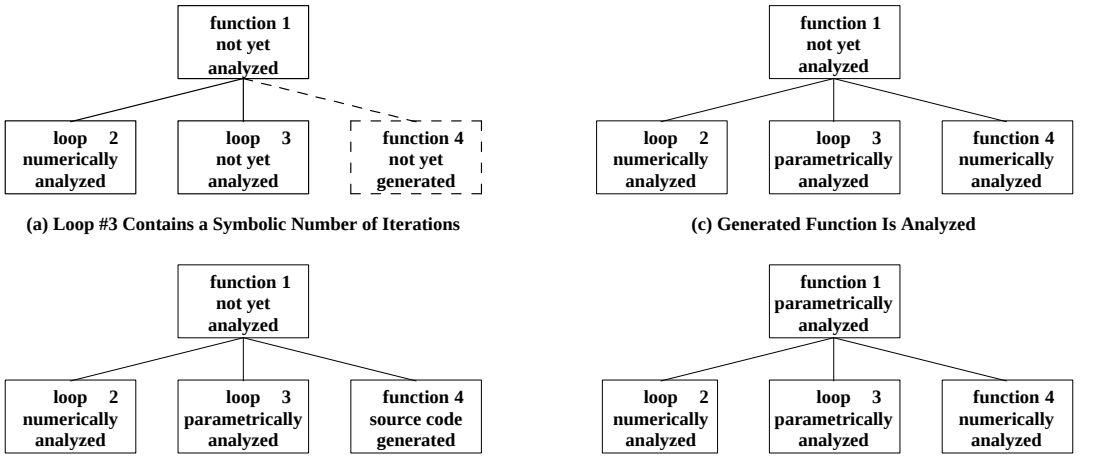

Figure 5: Example of Using Parametric Timing Predictions

```
int WCET_loop(int n)
{
   return (160*n + 267)*n+857;
}
```

Figure 6: Example of a Generated Function Calculating the WCET of a Loop

In summary, this approach allows for the timing analysis to be accomplished in stages. Parametric formulas are produced when needed and source code functions representing these formulas are generated. These generated functions are then compiled, analyzed, and revised timing predictions are made. This process continues until a formula can be produced for the entire program or task.

6. FUTURE WORK

There are several types of problems that we have yet to address in our parametric timing analysis. Sometimes the number of iterations executed by a loop can vary since it may depend on the value of an outer loop counter variable. Our approach to resolve this in the numeric timing analysis was to formulate a nonrectangular loop nest in terms of summations and solve the resulting equation [HSR00]. If the range of all of the loop variables in the nonrectangular loop nest cannot be bounded, then the summations can still be formulated and an equation in a closed form can still often be generated when performing parametric timing analysis.

Likewise constraints on branches can limit the paths that can be taken in a program. Automatically detecting these constraints can result in tighter WCET predictions [HeW99]. We need to formulate an equation when such branch constraints are present and the number of loop iterations is only known at run time.

Finally, we need to implement the approach described in Section 5. This will involve modifying the timing analyzer to recognize the names of special function calls indicating that they should return the WCET of functions or loops. We will also have to change the environment shown in Figure 1 to repeatedly invoke the compiler, static simulator, and timing analyzer as new functions are generated and then subsequently analyzed.

7. CONCLUSIONS

We have described how static timing analysis can be used to aid in making dynamic scheduling decisions. We have shown how a symbolic formula can be produced to represent the WCET for a loop when the number of loop iterations cannot be determined until run-time. While more complex cases should be addressed, preliminary results for some simple programs were given indicating that the approach for parametric timing analysis is almost as accurate as a completely numeric approach when the number of iterations in a loop exceeds the number of iterations required to reach a point where the WCET caching behavior is in a steady state. We also outlined how these parametric WCETs could be used without affecting the timing predictions of the loops and functions that were previously analyzed. The benefits of parametric timing analysis include expanding the class of applications that can be used in a real-time system, improving the accuracy of dynamic scheduling decisions, and more effective utilization of system resources.

ACKNOWLEDGEMENTS

This research was supported in part by the Spanish CICYI grant TAP98-0333-C03-01 and the National Science Foundation grants EIA-9806525, CCR-9904943, and EIA-0072043. The work performed by Frank Mueller was under the auspices of the U.S. Department of Energy by University of California Lawrence Livermore National Laboratory under contract No. W-7405-Eng-48.

8. REFERENCES

[AlY97] N. Al-Yaqoubi, *Reducing Timing Analysis Complexity by Partitioning Control Flow*, Masters Project, Florida State University, Tallahassee, FL (1997).

[AMW94] R. Arnold, F. Mueller, D. Whalley, and M. Harmon, "Bounding Worst-Case Instruction Cache Performance," *Proceedings of the Fifteenth IEEE Real-Time Systems Symposium*, pp. 172-181 (December 1994).

[BeD88] M. E. Benitez and J. W. Davidson, "A Portable Global Optimizer and Linker," *Proceedings of the SIGPLAN '88 Symposium on Programming Language Design and Implementation*, pp. 329-338 (June

1988).

[BeB00] G. Bernat and A. Burns, “An Approach to Symbolic Worst-Case Execution Time Analysis,” *25th IFAC Workshop on Real-Time Programming*, (May, 2000).

[CBW96] R. Chapman, A. Burns, and A. Wellings, “Combining Static Worst-Case Timing Analysis and Program Proof,” *Real-Time Systems* **11**(2) pp. 145-171 (1996).

[EnE00] J. Engblom and A. Ermedahl, “Modeling Complex Flows for Worst-Case Execution Time Analysis,” *Proceedings of the IEEE Real-Time Systems Symposium*, (November 2000).

[HBW92] M. G. Harmon, T. P. Baker, and D. B. Whalley, “A Retargetable Technique for Predicting Execution Time,” *Proceedings of the Thirteenth IEEE Real-Time Systems Symposium*, pp. 68-77 (December 1992).

[HSR00] C. Healy, M. Sjödin, V. Rustagi, D. Whalley, and R. van Engelen, “Supporting Timing Analysis by Automatic Bounding of Loop Iterations,” *Real-Time Systems*, pp. 121-148 (May 2000).

[HSR98] C. A. Healy, M. Sjödin, V. Rustagi, and D. B. Whalley, “Bounding Loop Iterations for Timing Analysis,” *Proceedings of the IEEE Real-Time Technology and Applications Symposium*, pp. 12-21 (June 1998).

[HeW99] C. A. Healy and D. B. Whalley, “Tighter Timing Predictions by Automatic Detection and Exploitation of Value-Dependent Constraints,” *Proceedings of the IEEE Real-Time Technology and Applications Symposium*, pp. 79-88 (June 1999).

[HWH95] C. A. Healy, D. B. Whalley, and M. G. Harmon, “Integrating the Timing Analysis of Pipelining and Instruction Caching,” *Proceedings of the Sixteenth IEEE Real-Time Systems Symposium*, pp. 288-297 (December 1995).

[HBL95] Y. Hur, Y. H. Bae, S. S. Lim, S. K. Kim, B. D. Rhee, S. L. Min, C. Y. Park, H. Shin, and C. S. Kim, “Worst Case Timing Analysis of RISC Processors: R3000/R3010 Case Study,” *Proceedings of the Sixteenth IEEE Real-Time Systems Symposium*, pp. 308-321 (December 1995).

[KMH96] S. Kim, S. L. Min, and R. Ha, “Efficient Worst-Case Timing Analysis of Data Caching,” *Proceedings of the IEEE Real-Time Technology and Applications Symposium*, (June 1996).

[KHR96] L. Ko, C. Healy, E. Ratliff, R. Arnold, D. Whalley, and M. Harmon, “Supporting the Specification and Analysis of Timing Constraints,” *Proceedings of the IEEE Real-Time Technology and Applications Symposium*, pp. 170-178 (June 1996).

[LMW96] Y. Li, S. Malik, and A. Wolfe, “Cache Modeling for Real-Time Software: Beyond Direct Mapped Instruction Caches,” *Proceedings of the IEEE Real-Time Systems Symposium*, pp. 254-263 (December 1996).

[LMW95] Y. S. Li, S. Malik, and A. Wolfe, “Efficient Microarchitecture Modeling and Path Analysis for Real-Time Software,” *Proceedings of the Sixteenth IEEE Real-Time Systems Symposium*, pp. 298-307 (December 1995).

[LBJ94] 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 Technique for RISC Processors,” *Proceedings of the Fifteenth IEEE Real-Time Systems Symposium*, pp. 97-108 (December 1994).

[Par93] C. Y. Park, “Predicting Program Execution Times by Analyzing Static and Dynamic Program Paths,” *Real-Time Systems* **5**(1) pp. 31-61 (March 1993).

[PuK89] P. Puschner and C. Koza, “Calculating the Maximum Execution Time of Real-Time Programs,” *Real-Time Systems* **1**(2) pp. 159-176 (September 1989).

[WMH97] R. T. White, F. Mueller, C. A. Healy, D. B. Whalley, and M. G. Harmon, “Timing Analysis for Data Caches and Set-Associative Caches,” *Proceedings of the IEEE Real-Time Technology and Applications Symposium*, pp. 192-202 (June 1997).

[ZBN93] N. Zhang, A. Burns, and M. Nicholson, “Pipelined Processors and Worst Case Execution Times,” *Real-Time Systems* **5**(4) pp. 319-343 (October 1993).