Generalizing Parametric Timing Analysis

ABSTRACT

In the design of real-time and embedded systems, it is important to establish a bound on the worst-case execution time (WCET) of programs to assure via schedulability analysis that deadlines are not missed. Static WCET analysis is performed by a timing analysis tool. This paper describes novel improvements to such a tool, allowing parametric timing analysis to be performed. Parametric timing analyzers receive an upper bound on the number of loop iterations in terms of an expression which is used to create a parametric formula. This parametric formula is later evaluated to determine the WCET based on input values only known at runtime. Effecting a transformation from a numeric to a parametric timing analyzer requires two innovations: 1) a summation solver capable of summing non-constant expressions and 2) a polynomial data structure which can replace integers as the basis for all calculations. Both additions permit other methods of analysis (e.g. caching, pipeline, constraint) to occur simultaneously. Finally, our tool features a user-friendly graphical interface to easily obtain results for any desired program. Combining these techniques allows our tool to statically bound the WCET for a larger class of benchmarks, which more closely match contemporary real-time and embedded application codes and allows users an easy method for procuring results.

Categories and Subject Descriptors

C.3 [SPECIAL-PURPOSE AND APPLICATION-BASED SYSTEMS]: REAL-TIME AND EMBEDDED SYSTEMS

General Terms

Verification, Reliability

Keywords

Worst-case execution time (WCET) analysis, parametric timing analysis

1. INTRODUCTION

Embedded systems, especially those in safety-critical or hard real-time environments, typically require that timing constraints be met. To guarantee these systems will meet deadlines, the worst-case execution time (WCET) of each program must be known. Knowledge of the WCET can also be used to dynamically scale voltage when a scheduler detects future slack time [1, 7, 11, 12]. This facilitates power savings, an especially important aspect in embedded systems. The process of determining the WCET of a program is known as timing analysis.

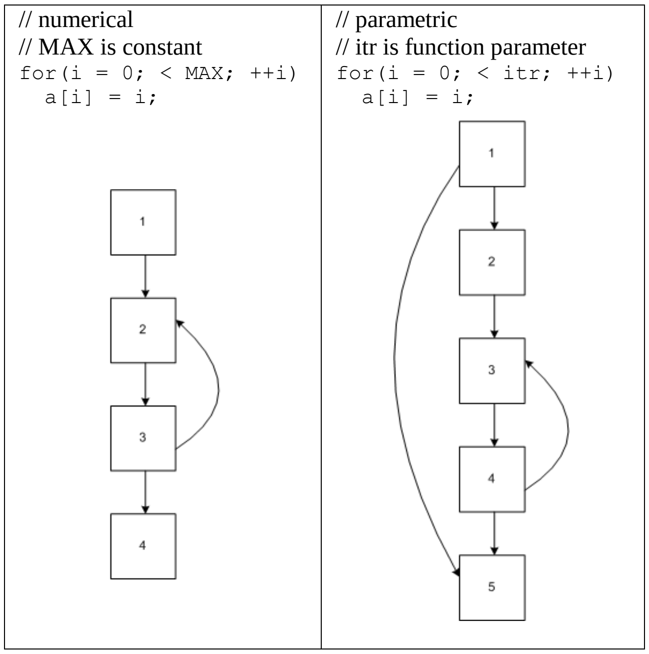
Two different approaches to find the upper bound exist: dynamic and static [3, 8, 17]. Dynamic timing analysis attempts to experimentally determine worst-case input data. This approach is error-prone and usually cannot guarantee that the absolute maximum has been found. In contrast, static timing analyzers examine all paths within a program and individually time each of them by taking into account the architectural components of the specified platform [11, 16]. If the number of paths exceeds a predefined maximum, control flow can be partitioned into synthetic sections that are each handled individually by the timing analyzer [9]. Once the individual paths have been timed, they are coalesced into loops, functions, and finally an entire program [11, 16]. The composition of paths abstracts from concrete program input and assures that a safe upper bound of the WCET is analytically derived. This bound is safe in that it never underestimates the actual WCET.

Static timing analysis has traditionally required loops to contain a constant number of iterations so analyzers may produce constant worst case execution bounds. Such constraints on input programs make this form of timing analysis numerical: the number of loop iterations is constant as is the final result from the timing analyzer. On the other hand, parametric timing analysis allows the number of loop iterations to be unknown at compilation as long as this value may be written as an expression. Such flexibility expands the class of programs which may be analyzed. Instead of providing a constant upper bound for a loop, a symbolic formula is created using the expression representing the number of loop iterations. The formula may be evaluated later to obtain the execution time for any given input [11, 16]. One example of a useful application for parametric timing analysis is image processing applications where the image dimensions are not known *a priori* [18].

We obtained a version of a timing analyzer [5, 9, 11, 16] and enhanced it to support generalized parametric analysis. The conversion required a three-fold process. First, a summation solver capable of summing non-constant numbers of iterations was written. Next, an advanced polynomial data structure was developed to express parametric formulas. Naturally, the polynomial data structure should not significantly slow numerical calculations. Finally, the polynomial class was integrated into the timing analyzer to replace integers in calculations. Constraint analysis – determining which paths in a loop can execute on certain iterations – can still take place with polynomial functionality.

Although these goals appear simple at first, parametric benchmarks present special problems to timing analyzers. Numerical timing analyzers receive a numerical upper bound on the maximum number of loop iterations that is either automatically determined by analyzing the program or is input by the user. Parametric analyzers receive the maximum number of loop iterations as an expression whose value is unknown at compilation (e.g. `for i = 0 .. n-1` contains *n* iterations). Because the expression must be evaluated at runtime, the loop body may not execute even once (e.g. the value of *n* could be `-1`). Such uncertainty means every parametric benchmark implicitly contains control flow even when the numerical version does not.

Figure 1 shows the blocks in a numerical and parametric version of the same benchmark. Note that the parameterized loop (body) may never execute (since there is a conditional check to ensure that the initial value is less than the limit). In the presence of an unknown number of iterations, parametric timing analyzers must present a solution encompassing all possibilities. These include the case where the loop body does not execute. This inconclusiveness complicates the process of timing analysis.



```
// numerical
// MAX is constant
for(i = 0; < MAX; ++i)
  a[i] = i;
```

```
// parametric
// itr is function parameter
for(i = 0; < itr; ++i)
  a[i] = i;
```

Figure 1: Blocks in a numerical and parametric benchmark.

Because the creation of a "true" parametric timing analyzer involves both a summation solver capable of symbolically evaluating loop iterations and an improved polynomial data structure, it was logical to split work into two separate phases. The first phase developed the symbolic summation solver. The second segment of work focused on implementing and integrating the polynomial data structure.

Previous work (described in Section 2) in timing analysis has been centered on numerical analysis where the maximum number of iterations for loops must be known at compilation. When loops are non-rectangular in nature, averaging techniques are used to guarantee tight WCET bounds (Section 2.2). Section 4 discusses the tool *Emtadel* which was written to handle the calculation of the number of iterations of nested triangular loops. This tool replaces Ctadel, a general-purpose algebraic simplifier [15]. Section 5 provides the details of the polynomial classes which were written to achieve increased performance during timing analysis calculations. We highlight the algorithms used in parametric timing analysis in Section 6 before discussing our experimental platform and results in Section 7.

In this paper, we describe novel techniques for bounding the WCET of parameterized programs. First, we describe a procedure for calculating the number of iterations of nested loops in terms of loop invariant parameters. Secondly, an existing timing tool was enhanced so parametric formulas rather than numeric quantities could be used in all calculations. Thirdly, a graphical user interface allows immediate feedback on the results of a desired program. This interface allows a large number of testing parameters – including hardware platform, cache configuration, and the number of loop iterations – to be modified. The final result is a static timing analyzer capable of determining WCET bounds for programs which are likely to be encountered in today's real-time embedded systems.

2. PREVIOUS WORK

2.1 Timing Analysis

The numerical timing analyzer that we obtained is illustrated in Figure 2. A modified C compiler (VPO) emits control-flow and branch constraint information during compilation [2]. This information is used by a static cache simulator so the caching categorization (e.g. always miss, always hit, first miss, first hit) of each instruction may be determined. The timing analyzer combines the control-flow, constraint, caching, and machine dependent (e.g. pipeline) information to reach an accurate upper bound of the WCET.

As the other studies of static timing analysis stipulate [5, 11, 16], a number of other conditions must be met. First, recursive programs are not allowed. Secondly, there can be no indirect calls. Finally, loops must be structured which for this purpose means they have a single entry point.

2.2 Bounding Loop Iterations

A stand-alone software package was used to calculate integral solutions for nested triangular loops. Triangular means that the number of iterations of the inner loop depends on the induction variable of the outer loop. Consider the code fragment

```
for (x = 0; x < 3; ++x)          | .
   for (y = 0; y <= x; ++y)      | . .
     statement;                  | . . .
```

The dots to the right indicate the number of iterations of the inner loop for each value of x. Because the number of iterations is non-constant, the minimum, maximum, and average number of iterations must be calculated. The software package could handle any level of loop nesting and could determine if the number of loop iterations is zero even when it is not immediately apparent by examining the original loops.

For the example presented above, the equivalent summation would be

$$total\_iterations = \sum_{x=0}^{2}(\sum_{y=0}^{x}(1)) \qquad (1)$$

which equals 6. The minimum number of iterations of the inner loop is 1, the maximum is 3, and the average is 2. The average number of iterations of the inner loop is calculated by dividing the sum (6) by the total number of iterations of the outer loop (3). The concept of representing the number of loop iterations as a summation was motivated by the work of Sakellariou [13].

Averaging the number of iterations considers the possibility of the inner loop being *zero-trip* or *partially zero-trip*. A zero-trip loop derives its name from the fact that a summation whose lower bound exceeds its upper bound evaluates to zero. Hence, a zero-trip loop does not execute the loop body. A partially zero-trip loop fails to execute the loop body on some iterations. If the condition of the inner loop in the example presented above was $y < x$ (instead of $y \le x$), the inner loop would be partially zero-trip. On the first iteration of the outer loop, x would be zero and the inner loop would not execute. On the second and later iterations, the inner loop would execute. Hence, the inner loop would be partially zero-trip in the modified example. Using the average number of iterations for triangular loops achieves tight WCET bounds.

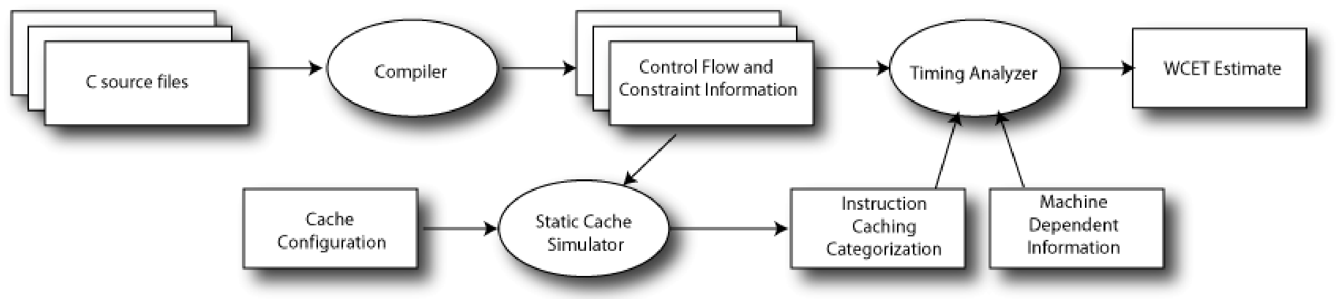

C source files
Compiler
Control Flow and Constraint Information
Timing Analyzer
WCET Estimate
Cache Configuration
Static Cache Simulator
Instruction Caching Categorization
Machine Dependent Information

Figure 2: Static Timing Analysis Process

Using rational numbers in the calculations allows even more accuracy when averaging the number of iterations of triangular loop nests. In the previous example, changing the conditional expression of the outer loop to $x \leq 3$ (instead of $x < 3$) would mean that the total number of iterations would be 10. Dividing by the total number of iterations of just the outer loop (4) provides the average number of iterations (2 ½). Representing the number of iterations as an integral quantity would mean that the inner loop's average number of iterations would be 3. Hence, a rational representation further tightens WCET bounds.

In the case of parametric loops, a naïve approach would derive the total time of the loop through a formula resembling

```
total_time =
   iteration time * iterations
```

where the number of iterations is a variable. Nested parametric loops requiring the number of iterations to be averaged would create a problem. The inner loop's final cycle count would be a function of the outer loop's induction variable instead of the variable passed into the function. For example, the code fragment (where *z* is a function parameter)

```
for (x = 0; x < z; ++x)    // loop 1
  for (y = 0; y < x; ++y) // loop 2
    sum += (double) (x * x - y * y);
```

would produce the following output:[1]

```
          TOTAL TIME
LOOP 2    51x + 4
```

The inner loop in this case should report its time in terms of the function parameter *z*. Only a summation solver capable of calculating the number of iterations symbolically could correct this naïve approach. Then, the formula for the average number of iterations could be used which would allow the same algorithm to solve both integral and polynomial cases.

3. RELATED WORK

Lisper presented a procedure to bound the WCET for parametric programs [10]. His approach is based on *abstract interpretation* of the program's execution and *integer linear programming* to compute the WCET bound. In general, both of these aspects are computationally intensive. The approach has not been fully implemented so its accuracy has not been determined.

Colin and Bernat proposed a scope-tree representation of a program in which control-flow constraints are expressed as annotations to the tree [4]. Execution frequencies of blocks are stored, and these can be parametric as well as numeric values. Control-flow constraints can also be specified. The tree can then be traversed to evaluate the WCET. Unfortunately, experimental results of the implementation are not provided.

Sandberg et al. have used program slicing to derive control-flow information [14]. Program slicing determines subsets of a program which can affect other sections of the program. This allows segments which do not affect program flow to be disregarded during control-flow analysis. Consequently, the space and time complexity required during control-flow analysis is greatly reduced. Being able to handling control-flow analysis automatically greatly diminishes the need for writing control-flow information in manual annotations which is painstaking and error-prone work.

4. COMPUTING ITERATIONS

The VPO C compiler (see Figure 2) creates a control-flow information file during compilation [2]. The timing analyzer obtains loop specific information from this control-flow file. This data includes the iterating (induction) variable and iteration information consisting of the minimum and maximum number of iterations (provided these are numeric quantities) or the initial, limit, and increment value of the induction variable [6]. If a parametric loop is nested within another parametric loop, the outer loop's iteration information is also reported. This allows a group of summations representing the loop structure to be generated. The program Emtadel handles the evaluation of the loop summations.

The popular numerical and symbolic computation system Mathematica allows C programmers to interact with its extensive library of functions. Although the features of this software exceed those of a summation solver written in C, two drawbacks exist. First, the portability of the timing analyzer would be compromised. The timing analyzer could only run on systems which had Mathematica installed, and changes to the Mathematica installation would necessitate updates to the environment variables used to access Mathematica functions. Additionally, computer algebra systems do not provide guards on the solutions which are returned. In the example presented above, *z* must be greater than zero. If such conditions are not met, the summation should be zero but in certain cases, substituting a value for *z* directly into the formula returned could result in a negative answer.

To illustrate calculating the average number of iterations for triangular loop nests, we first consider the control-flow information generated by the compiler. Loop information consists of the induction register, beginning value, limiting register (or value), relational operator, and information about the parent loop

[1] The coefficient of *x*, 51, is the number of cycles required to complete one iteration of loop 2. The constant term, 4, is the pipeline fill time encountered on the first iteration.

[6]. Using a summation grammar, the timing analyzer can create the summation command `sum(1, loop2 = 0 .. loop1 - 1 by loop1 = 0 .. z - 1)` for the nested loops presented above. Note that the loops appear in reverse order when compared to a handwritten summation. A symbolic summation solver calculates the total number of iterations nearly identically to the way the summation is solved by hand. This allows any number of non-induction variables to be present in a summation.

The command (`sum(1, ...)`) generated by the timing analyzer is passed to the new summation solver Emtadel. Emtadel parses this string to retrieve the information required for evaluation (induction variable and the upper and lower bounds). At this point, Emtadel's internal representation closely resembles the equivalent handwritten summation

$$total\_iterations = \sum_{loop1=0}^{z-1}\Big(\sum_{loop2=0}^{loop1-1}(1)\Big) \qquad (2)$$

As alluded to previously, Emtadel solves summations by using the Bernoulli formulas. For example,

$$\sum_{i=0}^{n} i = \frac{n(n+1)}{2} \qquad (3)$$

$$\sum_{i=0}^{n} i^2 = \frac{n(n+1)(2n+1)}{6} \qquad (4)$$

The innermost summation – in this case, the summation with induction variable `loop2` – is always solved first. The resulting intermediate solution is `loop1`. This answer is only applicable under the following condition: `loop1` must be greater than zero, the lower bound of the summation. If this condition is not upheld, the equation is invalid, for the loop would be zero-trip. After this intermediate solution is obtained, Emtadel solves the outer summation. The solution to this summation is identical to the solution of

$$total\_iterations = \sum_{loop1=0}^{z-1}(loop1) \qquad (5)$$

where the previous result has been substituted into the original summation. The final answer is

$$total\_iterations = \frac{1}{2}z^2 - \frac{1}{2}z \qquad (6)$$

Calculating the average number of iterations of the inner loop (in terms of the outer loop) is now trivial – dividing by the outer loop's total number of iterations produces

$$inner\_iterations = \frac{1}{2}z - \frac{1}{2} \qquad (7)$$

Since a general example has already been explained, another summation will demonstrate the complexity of calculating symbolic solutions when loops may be zero-trip. One example that requires a condition to be placed on the final answer is

```
for (i = 1; i <= z; i++)
  for (j = 7; j <= i; j++)
    for (k = 5; k <= i; k++)
      sum++;
```

where z is once again a function parameter. Emtadel receives the command `sum(1, k=5 .. i by j=7 .. i by i=1 .. z)` and generates the equivalent sigma notation

$$total\_iterations = \sum_{i=1}^{z}\Big(\sum_{j=7}^{i}\Big(\sum_{k=5}^{i}(1)\Big)\Big) \qquad (8)$$

The innermost summation will evaluate to zero on all iterations where $i < 5$. Similarly, the middle summation will also be partially zero-trip.

The innermost summation must be evaluated first. The intermediate solution in this case is $i - 4$ if $i \geq 5$. The condition ($i \geq 5$) allows Emtadel to track (partially) zero-trip sums. To calculate the middle summation, previous conditions must first be considered. In this case, the variable i must be greater than or equal to 5, but the current summation demands that i be greater than or equal to 7 since this summation's zero-trips never include an inner summation's sum. Therefore, the condition $i \geq 5$ is updated to $i \geq 7$. Emtadel uses the appropriate Bernoulli formula on each individual term. The intermediate solution is $i^2 - 10i + 24$ if $i \geq 7$. When the outer summation is reached, the lower bound is replaced by the conditional demand $i \geq 7$. The new lower bound is 7, and the summation only iterates from $i = 7$ to z. Hence, any (partially) zero-trip loops have been taken into account through the use of the conditional expression. The final answer is

$$iterations = \frac{1}{3}z^3 - \frac{9}{2}z^2 + \frac{115}{6}z - 25 \qquad (9)$$

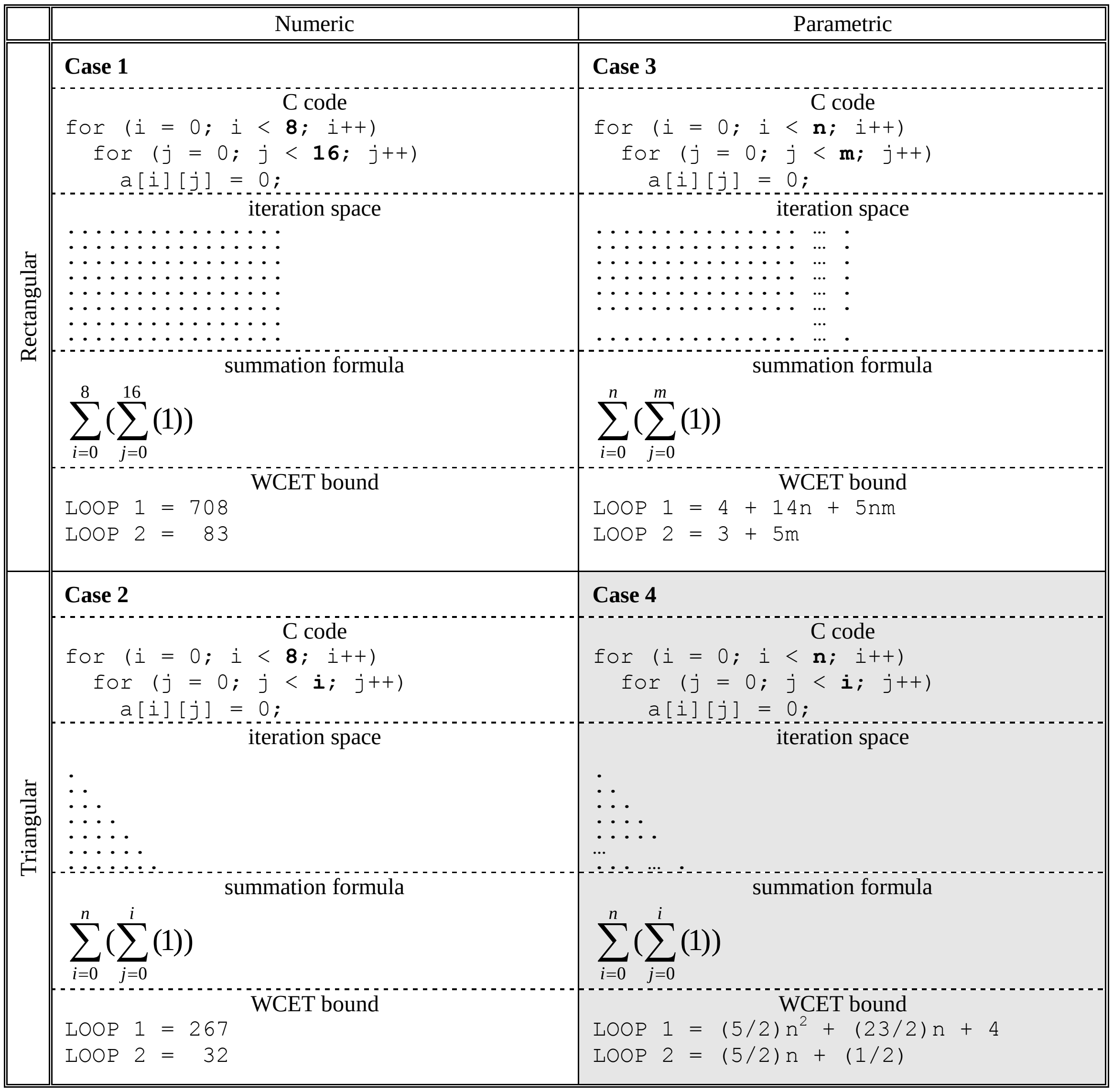
if $z \geq 7$. If z is less than 7, the summation evaluates to zero.

This symbolic approach allows the timing analyzer to input loop variable information directly instead of having to form equations independently. The improvement replaces a significant portion of code contained within the timing analyzer and improves encapsulation. Additionally, any condition(s) placed on the final solution allows the timing analyzer itself to place conditions on the validity of its final answer.

The timing analyzer uses Emtadel when the number of iterations of a loop must be averaged. This case only occurs in triangular loop nests like the one presented above and cases 2 and 4 in Figure 3. Emtadel will not be used to calculate the number of iterations of nested numeric or parametric loops. Examples demonstrating all these different types of loops are in Figure 3. The WCET bounds shown in the figure were determined by using our enhanced parametric timing analyzer. To reiterate, Emtadel is called only when nested loops are triangular in nature.

5. ANALYSIS USING SYMBOLIC FORMULAS

The summation solver Emtadel requires a polynomial data structure to calculate and represent the symbolic solution of a summation. The timing analyzer also uses polynomials to compute parametric results. We adopt the following definition of polynomials: a polynomial contains one or more terms combined using addition. A term consists of zero or more variables each raised to a power. Each term is multiplied by either a constant (any rational number) or a *polynomial chain*. A polynomial chain allows the timing analyzer to express a parametric formula as the maximum (or minimum) of two or more polynomials. This functionality is required when the largest (or smallest) of a group of polynomials cannot be known until the values of variables are substituted into the expressions. The major fields in our implementation are presented in Table 1.



	Numeric	Parametric
Rectangular	**Case 1**	**Case 3**

	C code
	`for (i = 0; i < 8; i++)`
	`  for (j = 0; j < 16; j++)`
	`    a[i][j] = 0;`
		C code
		`for (i = 0; i < n; i++)`
		`  for (j = 0; j < m; j++)`
		`    a[i][j] = 0;`

	iteration space	iteration space
	summation formula $\sum_{i=0}^{8}(\sum_{j=0}^{16}(1))$	summation formula $\sum_{i=0}^{n}(\sum_{j=0}^{m}(1))$

	WCET bound
	`LOOP 1 = 708`
	`LOOP 2 =  83`
		WCET bound
		`LOOP 1 = 4 + 14n + 5nm`
		`LOOP 2 = 3 + 5m`

Triangular	**Case 2**	**Case 4**

	C code
	`for (i = 0; i < 8; i++)`
	`  for (j = 0; j < i; j++)`
	`    a[i][j] = 0;`
		C code
		`for (i = 0; i < n; i++)`
		`  for (j = 0; j < i; j++)`
		`    a[i][j] = 0;`

	iteration space	iteration space
	summation formula $\sum_{i=0}^{n}(\sum_{j=0}^{i}(1))$	summation formula $\sum_{i=0}^{n}(\sum_{j=0}^{i}(1))$

	WCET bound
	`LOOP 1 = 267`
	`LOOP 2 =  32`
		WCET bound
		`LOOP 1 =` $(5/2)n^2 + (23/2)n + 4$
		`LOOP 2 =` $(5/2)n + (1/2)$

Figure 3: Examples of numeric and parametric loop nests.

Table 1: Major classes and fields in the polynomial data structure

Class	Fields
`Polynomial`	`vector<Term> terms`

`Term`	`vector<VariablePower> vps`
	`Rational coefficient`
	`PolynomialChain coefficientPolynomial`
	`bool useCoefficientPolynomial`

`Polynomial Chain`	`vector<Polynomial> polynomials`
	`int operation // MIN or MAX`

`Variable Power`	`String variable`
	`Rational power`

Since polynomial expressions encompass integral expressions, it made sense to move the underlying structure of the timing analyzer away from integers and to polynomials. The existing polynomial data structure presented two obstacles that had to be overcome before this could occur. First, the comparison functionality was limited to equality. Two polynomials were considered equal only if they were exact copies of each other. Altering the order of terms or variables within terms would result in polynomials being considered unequal. The timing analyzer naturally requires all of the inequality operators for appropriate control flow. The second limitation of the polynomial data structure was that all mathematical operations require special function calls. None of these functions considered polynomial chains. When a branch is found in a parametric program, the timing analyzer must return a Max(…) expression representing the WCET of the timing node. The mathematical functions would perform the operation on the first polynomial and ignore other polynomials in the Max expression.

The PolynomialChain class allows both minimum and maximum expressions. One reason for this functionality is completeness – the polynomial class as a whole was designed to be stand-alone. But the primary reason is actually the timing analysis algorithm. Although one normally thinks of WCET as a maximum of several possible values (see Table 3), intermediate steps require calculating a minimum. Thus, it is possible to find a Min(…) expression as part of the WCET.

The new polynomial class would have the following requirements. It would provide comparable performance for operations involving constants. While some overhead is unavoidable, representing an integer as a polynomial must not noticeably degrade performance. Mathematical operations on polynomials would also have to be more efficient than the previous implementation. Finally, all mathematical and commonly used operators would be overloaded so the class could easily take the place of integers.

As the polynomial class was written, classes from C++ standard template library – namely vectors (see Table 1) – were used to maximize performance. The result is an improved data structure which achieves the aforementioned goals. The additional overhead of the polynomial class does not degrade the performance of the timing analyzer. Even timing the benchmark Matmult (see Table 2 and Table 3) which has fifteen timing nodes still completes in less than three seconds on a Sun Ultra 10 workstation. [2] Polynomials may be simplified after each operation which means successive operations complete more quickly.

The inequality functions do present a practical limitation when non-constant expressions are compared. Comparison between two polynomials is modeled after comparison in Java: there is a comparison function which returns an integer: zero if the polynomials are equivalent, a negative number if this polynomial is less than the specified one, and a positive number otherwise. The result of some comparisons (e.g. those between polynomials with multiple variables) cannot be determined. When this is the case, the comparison function throws a logic exception. The overloaded operators will catch this exception and return false. This means the inequality operators no longer represent a closed relation.

For example, let `poly1` and `poly2` be polynomials containing the values x and y. When the function `poly1.compareTo(poly2)` is called, a logic exception is thrown because the answer is uncertain without knowing the values of x and y. Because the overloaded operators catch the exception, `poly1 <= poly2` is false as is `poly1 > poly2`. If the calling code relies on this comparison to guide program flow, both conditions must be checked, and a final (else) case may need to be added for comparisons which cannot be determined.

6. PARAMETRIC TIMING ANALYSIS

Parametric timing analysis utilizes two distinct segments of analysis to achieve tight WCET bounds. The first considers branch constraints of individual paths within a loop. The second handles the timing of the entire loop and enforces the constraint information reported by the first.

6.1 Branch Constraint Detection

The timing analyzer automatically performs branch constraint detection using the control-flow information reported by the VPO C compiler [2, 5]. Branch constraint detection in parametric benchmarks falls into three separate categories: infeasible paths, effect-based constraints, and the minimum/maximum number of iterations. The amount of automatic detection in parametric benchmarks is limited by the control-flow information whereas the amount of control-flow information reported for numerical benchmarks is considerably more and allows additional analysis.

The first part of branch constraint detection finds infeasible paths within the loop. These infeasible paths are disregarded during later path and loop analysis. The second considers effect-based constraints. Effect-based constraints check to see which paths can or cannot follow each other. The final stage of propagating branch constraints in parametric benchmarks assigns a minimum and maximum number of iterations for each path. The loop analysis algorithm limits the number of times a path may be executed based on these cumulative results [5].

6.2 WCET Loop Analysis

The paths of a loop or function are each independently evaluated. Once the WCET has been established for each path, the longest path is selected and used in the algorithm abstracted below [6].

```
// process loop iterations
iters = 0;
while (iters < N)
{
  // process iters while longest path has first miss/hit
  do
  {
    iters = iters + 1;
    wcpath = find the longest path;
    cycles =
        cycles + wcpath->cycles;
  } while (iters < N && caching
      behavior of wcpath can change);

  // process remaining iterations of longest path
  if (iters < N)
  {
    iters_to_do = remaining iterations
        of longest path;
    cycles = cycles +
        (iters_to_do * wcpath->cycles);
    iters = iters + iters_to_do;
  }
}
```

Such an algorithm accounts for updates in the instruction cache on different iterations of a loop. When the outermost while loop terminates, the variable `cycles` contains the WCET of the loop. If parametric timing analysis is being performed, the WCET of the loop will be a symbolic formula instead of a constant.

Each loop or function is considered a node in the timing tree. Nodes are processed in a bottom up manner meaning the timing bounds of child nodes must be established before the node itself may be evaluated. A child node is considered a single block which is inserted into the parent node [11, 16].

7. EXPERIMENTAL PLATFORM

A graphical automated tester has been designed and implemented to check the results of the timing analyzer against a hardware simulator. This program allows the simulated number of cycles to be compared with the estimation produced by the timing analyzer: both the simulator and the timing analyzer are run automatically from the automated tester without receiving any

[2] The Sun Ultra 10 workstation used during the test contains an UltraSPARC II*i* processor running at 440 MHz and 256 MB of memory.

additional user input. A ratio of the estimated cycles and the simulated cycles provides immediate feedback on the percent difference. The automated tester immediately flags as incorrect benchmarks for which the timing analyzer produces either an overestimation greater than a specified percentage or any underestimation. This highly flexible program makes testing the timing analyzer easier and can identify timing nodes where the timing analyzer's output is incorrect. Such functionality facilitates quick debugging when a mistake in the timing analyzer is found.

The graphical tester allows users to vary multiple input values to the timing analyzer and hardware simulator. A user may choose any benchmark and the architecture[3] (e.g. SPARC) for running the test. The cache configuration – including the number of cache lines, cache associatively, the number of bytes per line, and the miss penalty – may be adjusted. The user may select appropriate discrete values (powers of two) for the number of lines, associativity, and the number of bytes per line of cache. The miss penalty may range from zero to fifty where zero represents all cache hits. Finally, parametric benchmarks allow different numbers of loop iterations to be specified. The number of iterations is used as input to the simulator and to evaluate the symbolic formula produced by the timing analyzer. In the case of nested parametric loops, the outermost loop executes for the specified number of iterations, and all inner loops – regardless of nesting level – execute for double the number. This ensures both loops are being timed accurately. A user may also choose to run all of the benchmarks at once using an identical cache configuration. Attempting to hand check the timing analyzer for all of the different dimensions of analysis is impractical. The automated tester couples the accuracy of a hardware simulator with the ease of using a GUI application.

The database tab lists all of the benchmarks that have been run since starting the automated tester. A glance at the table can quickly verify the accuracy of the timing analyzer and identify any benchmarks which may require minor changes to the timing analyzer. Running the same benchmark with different parameters will result in multiple instances of the benchmark in the listing. The exact parameters used to run each benchmark may be viewed by selecting the appropriate instance of the benchmark. If desired, a database may be saved for future reference or reopened from a previous session.

The simulator tab contains a list of each benchmark which has been simulated. The majority of time spent checking a benchmark comes from simulation. The process is sped by saving the information reported by the simulator when each benchmark is run. Asking the automated tester to run a benchmark which has previously been simulated results in rerunning the timing analyzer and using the saved simulator data. This saved information may be deleted for one or all of the benchmarks – this results in simulating each benchmark again.

Even when the timing analyzer does not produce satisfactory results, the automated tester can produce a detailed analysis of the benchmark. Detailed analysis provides the simulated and estimated number of cycles for every node in the timing tree. This quickly focuses attention on the problematic loop or function.

Figure 4 is a screen shot of the automated tester. On the left side of the window is the listing of benchmarks. The names of the benchmarks come from a text file which is opened using the Load List button. Underneath the Load List button is a check box labeled "Build section cycles." This check box is used to generate detailed analysis from both the hardware simulator and the timing analyzer. The right side allows the user to modify the aforementioned testing parameters using sliders and drop down list boxes. On some architectures (e.g. ones without cache), the cache specifications will be ignored by both the timing analyzer and hardware simulator. The Build button runs the currently selected benchmark (in the left pane), and the Build All button runs all of the benchmarks using the specified cache configuration. The text box at the bottom contains the results of the timing analyzer (estimated cycles) and hardware simulator (simulated cycles) and the ratio of the two. The ratio provides the percent difference of the two. In the case of a parametric benchmark, the symbolic WCET is displayed as well as a cycle count which is obtained by evaluating the symbolic formula with the number of iterations specified by the user. The polynomial evaluation is automatically performed by the automated tester at runtime.

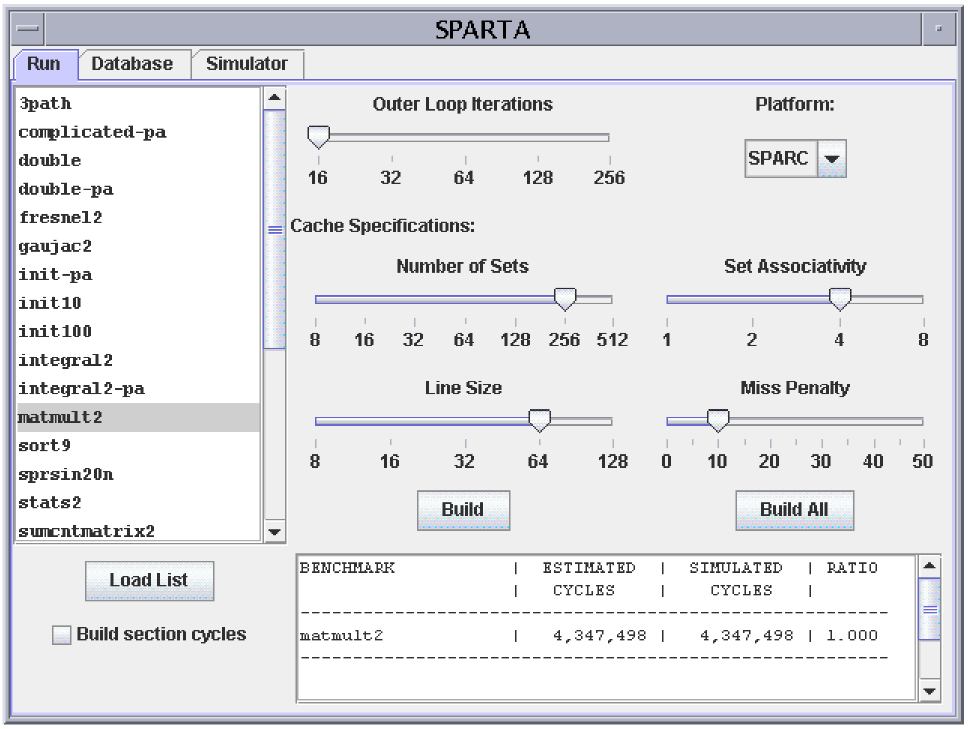
Table 2 lists a number of test programs which were chosen to demonstrate the effectiveness of Emtadel and the enhanced timing analyzer.

Table 2: Parametric Test Programs

Program	Description
Integral	Evaluates a double integral over a trapezoidal region.
Matmult	Multiplies two square matrices and stores the result in a third matrix.
Sprsin	Converts an integer matrix into row-indexed sparse storage mode.
Sumcntmatrix	Sums and counts the number of positive and negative elements in a matrix.
Summatrix	Adds two square matrices and stores the result in a third matrix.
Summinmax	Sums the minimum and maximum of the corresponding elements of two integer vectors.

The parametric benchmarks used are identical to numeric versions except that the limit values of loops are parameters passed into the function. The main function receives a different command line argument for each different loop limit. The argument is a left shift amount which means 2^0, 2^1, …, 2^9 are all valid numbers of maximum iterations. This approach introduces the minimum amount of extra code into the benchmark's source. We chose these numbers for convenience in testing; the timing analyzer is not limited to benchmarks which contain an unknown number of iterations which are powers of two.

[3] Different architectures necessitate a separate hardware simulator for each platform. Benchmarks are compiled for the specified architecture and run with the appropriate simulator.



3path
complicated-pa
double
double-pa
fresnel2
gaujac2
init-pa
init10
init100
integral2
integral2-pa
matmult2
sort9
sprsin20n
stats2
sumcntmatrix2

Outer Loop Iterations: 16 (16, 32, 64, 128, 256)

Platform: SPARC

Cache Specifications:

Number of Sets: 256 (8, 16, 32, 64, 128, 256, 512)

Set Associativity: 4 (1, 2, 4, 8)

Line Size: 64 (8, 16, 32, 64, 128)

Miss Penalty: 10 (0, 10, 20, 30, 40, 50)

```
BENCHMARK              |  ESTIMATED  |  SIMULATED   | RATIO
                       |   CYCLES    |    CYCLES    |
------------------------------------------------------------
matmult2               |  4,347,498  |   4,347,498  | 1.000
------------------------------------------------------------
```

Figure 4: Screenshot of Automated Testing Application

Table 3 shows the results of the benchmark programs. The column Estimated Cycles gives the execution time predicted by the timing analyzer. Observed Cycles was obtained by using the integrated instruction cache and pipeline simulator which received worst case input data.

As mentioned in the introduction, all of the benchmarks listed implicitly contain control-flow information because they have been parameterized. The timing analyzer performs additional control-flow constraint analysis on all benchmarks as previous described in [5]. Had this analysis not been performed, the benchmark Summinmax (which contains an infeasible path) would have an additional overestimation of 5 percent. Although multiple paths exist in each timing node, only the benchmark Integral reports its final answer as the Max of two distinct polynomials. In the intermediate steps, the other benchmarks also calculated a symbolic solution using Max(…) expressions. Nonetheless, the polynomial data structure was able to determine in these other cases that one of the polynomials was always larger than the other. Hence, the smaller polynomial expression has been subsumed by the larger.

Table 3: Results for Parametric Test Programs

Program	Formula	n iterations	Observed Cycles	Estimated Cycles	Ratio
Integral	Max((153/2)n^2 – n + (193/2), (153/2)n^2 + (49/2)n + 90)	16	20,026	20,066	1.002
		128	1,256,562	1,256,602	1.000
Matmult	31n^3 + 186n^2 + 61n + 361	16	175,399	175,929	1.003
		64	8,891,095	8,892,585	1.000
Sprsin	36n^2 + 33n + 185	16	9,702	9,929	1.023
		128	592,774	594,233	1.002
Sumcntmatrix	106n^2 + 36n + 298	16	27,869	28,010	1.005
		128	1,741,357	1,741,610	1.000
Summatrix	93n^2 + 35n + 199	16	24,486	24,567	1.003
		128	1,528,310	1,528,391	1.000
Summinmax	16n + 73	16	318	329	1.035

		128	2,110	2,121	1.005

8. FUTURE WORK

Only the first five Bernoulli formulas (through i^5) have been entered into Emtadel. This limit is not a serious as it may first appear, for the loops in the summation command would all have to be triangular to cause a problem. Although generalized ways to calculate the Bernoulli numbers and formulas exist, the difficulty in converting these automatically into a sequence of polynomial operations make such a solution both unwieldy and prohibitive. As mentioned in the introduction, when the number of paths contained within a timing node exceeds a defined maximum, control flow can be partitioned and synthetic timing nodes created [9]. When the timing analyzer encounters five nested triangular loops, it could create a synthetic section which would be handled similarly to a function call. Such an approach would overcome the lack of a generalized Bernoulli formula.

A polynomial should be able to accept boundary conditions, and these could aid in inequality operations. The loops encountered by the timing analyzer would not be executed in the presence of a negative variable. If a loop iterates from zero to a negative number (and the loop variable has a positive increment), the loop is zero-trip. Such bounds checks allow fewer Max(…) expressions to be generated by the timing analyzer.

9. CONCLUSION

The contributions of this paper are threefold. First, we describe a generalized procedure for computing summations that represent the number of iterations of nested loops. This approach can handle numeric as well as multi-variate quantities and express the number of iterations in terms of loop invariant parameters. Second, we significantly enhanced an existing timing tool so that it represents both the number of iterations as well as the WCET of code segments in terms of generalized polynomial expressions rather than simply a numeric number of cycles. We then enhanced its loop analysis algorithm to take advantage of our new polynomial representation and accurate loop iteration computations. The result is that we are now able to statically bound the WCET for a larger class of benchmarks. Finally, our tool features a user-friendly interface which a user can utilize to quickly gain feedback on the results of a desired benchmark. Using the GUI, the user can vary the parameters of benchmark testing according to a variety of dimensions, including the choice of hardware platform and cache configuration, as well as specifying the number of loop iterations for parametric benchmarks.

10. ACKNOWLEGEMENTS

This work was supported in part by NSF grants. [The grant numbers have been removed to facilitate the double-blind review.]

11. REFERENCES

[1] Aydin, H., Melhem, R., Mosse, D., and Mejia-Alvarez, P., “Power-Aware Scheduling for Periodic Real-Time Tasks,” *IEEE Transactions on Computers*, *53*, *5* (May 2004), pp. 584 – 600.

[2] Benitez, M.E., and Davidson, J.W., “A Portable Global Optimizer and Linker,” *Proceedings of the SIGPLAN ’88 Symposium on Programming Language Design and Implementation*, June 1988, pp. 77 – 98.

[3] Bernat, G., Colin, A., and Petters, S. M., “WCET Analysis of Probabilistic hard Real-Time Systems,” *Proceedings of the 23rd IEEE Real-Time Systems Symposium*, December 2002, pp. 279 – 288.

[4] Colin, A., Bernat, G., “Scope-tree: a Program Representation for Symbolic Worst-Case Execution Time Analysis,” *Proceedings of the 14th Euromicro Conference on Real-Time Systems*, June 2002, pp. 50 – 59.

[5] *Previous work by authors.*

[6] *Previous work by authors.*

[7] Kang, D., Crago, S., and Suh, J., “A Fast Resource Synthesis Technique for Energy-Efficient Real-Time Systems,” *Proceedings of the 23rd IEEE Real-Time Systems Symposium*, December 2002, pp. 225 – 234.

[8] Kirner, R., Wenzel, I., Rieder, B., and Puschner, P., “Using Measurements as a Complement to Static Worst-Case Execution Time Analysis,” *Intelligent Systems at the Service of Mankind, 2*, December 2005.

[9] *Previous work by authors.*

[10] Lisper, B., “Fully Automatic, Parametric Worst-Case Execution Time Analysis,” *Proceedings of the 3rd International Workshop on WCET Analysis*, July 2003, pp. 99 – 102.

[11] *Previous work by authors.*

[12] Pillai, P., and Shin, K., “Real-Time Dynamic Voltage Scaling for Low-Power Embedded Operating Systems,” *Proceedings of the 18th ACM symposium on Operating Systems Principles*, 2001, pp. 89 – 102.

[13] Sakellariou, R., “Symbolic Evaluation of Sums for Parallelising Compilers,” *Proceedings of the 15th IMACS World Congress on Scientific Computation, Modeling and Applied Mathematics*, 1997.

[14] Sandberg, C., Ermedahl, A., Gustafsson, J., and Lisper, B., “Faster WCET Flow Analysis by Program Slicing,” *Proceedings of the 2006 ACM SIGPLAN/SIGBED Conference on Language, Compilers, and Tool Support for Embedded Systems*, June 2006, pp. 103 – 112.

[15] Van Engelen, R., Wolters, L., and Cats, G., “Ctadel: A Generator of Multi-Platform High Performance Codes for PDE-based Scientific Applications,” *Proceedings of the 10th ACM International Conference on Supercomputing*, May 1996, pp. 86 – 93.

[16] *Previous work by authors.*

[17] Wegener, J., and Mueller, F., “A Comparison of Static Analysis and Evolutionary Testing for the Verification of Timing Constraints,” *Real-Time Systems, 21, 3* (November 2001), pp. 241 – 268.

[18] Zinner, C., and Kubinger, W., “ROS-DMA: a DMA Double Buffering Method for Embedded Image Processing with Resource Optimized Slicing,” *Proceedings of the 12th IEEE Real-Time and Embedded Technology and Applications Symposium*, April 2006, pp. 361 – 372.