Evolutionary Techniques for Parametric WCET Analysis*

Amine Marref

Department of Computer Science
Umm Al-Qura University
Makkah, Saudi Arabia
ajmarref@uqu.edu.sa

Abstract

Estimating the worst-case execution time (WCET) of real-time programs is pivotal in their verification. WCET estimation either yields a numeric value that represents the maximum execution time of the program when executed on a specific hardware platform; or yields a parametric expression in the form of some function of the input which when instantiated with a particular input value, gives a WCET estimation of the program when triggered by this input specifically (on a specific hardware platform). Parametric WCET analysis provides extra accuracy as the WCET estimation can be tuned to particular input values at runtime; and is usually of interest to dynamic-scheduling schemes.

In this paper we use genetic programming as an alternative method to approach the problem of parametric WCET analysis. Parametric expressions are captured automatically by the genetic program based on end-to-end executions of the program under analysis. The technique is complementary to static parametric WCET analysis and more amenable to industrial practice. Experimental evaluation shows that the presented technique computes accurate parametric expression in an almost negligible time.

1998 ACM Subject Classification D.2.5 Testing and Debugging

Keywords and phrases real-time systems, parametric worst-case execution-time analysis, end-to-end testing, genetic programming

Digital Object Identifier 10.4230/OASIcs.WCET.2012.103

1 Introduction

WCET analysis can result in two types of WCET estimations: numeric values or parametric expressions. WCET estimations in the form of numeric values are the predominant in the literature; in this case the WCET analysis returns a single constant value that indicates the maximum number of clock cycles (or other time-measurement units) that the program can potentially spend during its lifetime execution on a specific hardware platform. On the other hand, parametric WCET estimations come in the form of mathematical functions of the inputs of the program.

Parametric WCET estimations are more useful than numeric-form estimations when the scheduling algorithm — that uses the results of the WCET analysis — can utilize the extra context-sensitivity provided by the parametric expression. In this case, the scheduling algorithm can compute a constant WCET value $w_p(v_i)$ for some program p with parametric

* This research is funded by the program of strategic technologies of the national plan for science, technology, and innovation in Saudi Arabia No. 11–INF1705–10.

expression $w_p(v)$ each time p consumes a new input vector $v = v_i$, by instantiating the input v_i inside the parametric expression $w_p(v)$.

Parametric WCET analysis has been the subject of several research works whose most common factor with respect to our work is that they are based on static analysis of program code to determine the parametric expressions. In this paper, we present a novel approach for the problem of parametric WCET estimation based on genetic programming, and is more suitable for end-to-end WCET analysis used in the industry.

This paper is structured as follows. Section 2 introduces genetic programming. Section 3 explains the problem of parametric WCET analysis from the paper's point of view. Section 4 explains the use of genetic programming to approach the problem of parametric WCET analysis. Section 5 describes the experimental evaluation of our approach. Section 6 describes related work in parametric WCET analysis. Section 7 contains concluding remarks and directions for future work.

2 Genetic Programming

Genetic programming (GP) [8] is a bio-inspired computer algorithm that mimics natural evolution of living organisms. It is similar to genetic algorithms with the exception that individuals are computer programs as opposed to vectors of values.

The objective of GP is to evolve a computer program that solves a given problem. In order to do so, a population of computer programs called individuals — that are randomly generated initially — is evolved across a number of generations. The evolution of the population involves the exchange of genetic material between the individuals through cross-over operations, and the alteration of the genetic material of single individuals through mutation operations. A selection strategy is applied to the individuals of a population in a given generation to decide which ones are allowed to proceed to the next generation. Such selection is based on the fitness of the individuals which is a problem-dependent value that specifies the goodness of an individual in solving the problem at hand. The evolution continues until a good-enough individual that solves the problem adequately is found, or until a maximum number of generations is reached.

Each individual in the population is a program represented by its abstract-syntax tree (AST). All non-leaf nodes of the AST represent operators, and leaf nodes represent problem variables or constant values. Crossing-over two programs means taking one or more subtrees from the first program and inserting them into the second program, and taking one or more subtrees from the second program and inserting them into the first program (cross-overs can be single-point or multiple-point). Mutating means changing the content of one or more nodes in the AST.

GP can be used to solve a variety of optimization problems amongst which is symbolic regression that we shall describe here because it is the essence of our approach. To solve a symbolic-regression problem (also known as function-discovery problem), the genetic program (GP[1]) takes as input a set of m observations of values of some variable x, a set of observations of values of some variable y, and tries to identify the function f_0 such that $y = f_0(x)$ is true for all pairs (x, y) in the m observations — and also true outside the m observations. The function f_0 to be determined is a computer program that will be evolved by the GP. The initial population of the GP contains a number of n randomly-generated functions $f_1, f_2, \cdots, f_n$ — each represented as an AST e.g. Figure 1 shows the AST representation of

[1] We use GP to refer to both "genetic programming" and "genetic program".

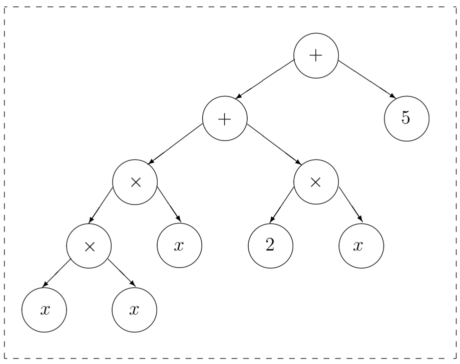

Figure 1 An example abstract-syntax tree corresponding to some function $f_i(x) = x^3 + 2x + 5$.

some function $f_i(x) = x^3 + 2x + 5$ in the GP. The functions will be crossed over and mutated over generations to produce new fitter functions. The fitness of a function f_i is calculated as the sum of differences ($|x - y|$) for all pairs (x, y) in the m observations. At the end, the GP either discovers f_0 during evolution or another function of equal or inferior fitness. Notice that the GP can derive some function f_0' that satisfies $y = f_0'(x)$ for all observed data pairs (x, y) but does not satisfy $y = f_0'(x)$ in the general case i.e. for some unobserved pairs (x, y) the relation $(y = f_0(x) \wedge y \neq f_0'(x))$ holds. This is a classic case of over-fitting and is usually (partially) tackled by dividing the m observations into a training part used during evolution, and a testing part used after evolution to give an indication of how well the derived function f_0' generalizes to new unseen data. Should the derived function not generalise well enough, evolution is restarted with the derived function f_0' injected in the initial population of the new GP run.

3 Parametric WCET Analysis

The objective of parametric WCET analysis is to derive a function that expresses the WCET of some program in terms of its input. These parametric expressions contain constant factors and terms, and variable factors and terms. The constant factors and/or terms in the parametric WCET expression refer to known entities in the analysis such as program-segment execution times and program-segment execution counts. These are derived by (i) the WCET analysis through its flow analysis (execution counts) or processor-behaviour analysis/runtime measurements (execution times), (ii) input by the user (usually as execution counts in the form of loop and recursion bounds, and sometimes as execution times of black-box components or libraries not available for the analysis), or (iii) a mixture of analysis-derived and user-input execution times and execution counts. We use the term program segment to refer to program entities of interest to parametric WCET analysis e.g. a loop body whose number of iterations is controlled by the input and which appears as a term $a_j x_j$ in the parametric expression — where a_j is the execution time of the loop body and x_j is its execution count.

The variable factors and terms in the parametric expression can refer to the whole input, parts of the input, and/or some of the input's properties of interest. For example, in a program that returns the factorial of a nonnegative integer, the whole input i.e. the integer whose factorial is of interest becomes a variable in the parametric expression; and in a program that sorts integers, the size of the input (which is a property of the input) becomes a variable in the parametric expression. Here we use the word input to mean the type of the input as opposed to a single instantiation of it e.g. an input is an array of n integers, is a real number, etc.

Input can influence the execution time of the program in two different ways: by altering the execution times of the program's segments, and/or altering their execution counts. For example, if part of the input is passed as one of the operands of some variable-latency instruction such as multiplication, the WCET estimation will change according to the value passed to the variable-latency instruction. In this case, input affects the execution time of the program segment that contains the variable-latency instruction, and consequently affects the overall program's execution time. If part of the input contributes directly or indirectly to the value of some variable that controls the number of iterations of some loop, the execution time of the program will also change according to the value of this input. In this case, the input affects the number of times a set of program segments executes, and consequently affects the overall program's execution time. Figure 2 shows example scenarios of how input affects the execution time of the program. In line 32, input variable c affects the flow of the program and consequently affects the execution time. Including c in the parametric WCET expression of `foo()` will be in the form of a conditional statement. Formula (1) shows (a very simplified) parametric WCET expression of function `foo()` of Figure 2. The function $f_{mul}(a, b)$ gives the execution time of the multiplication operation in terms of its operands a and b. Formula (1) is the ultimate form of WCET parameterization as it accounts for the diverse ways by which the input affects the execution time.

$$w_{foo} = \begin{cases} n + m + f_{mul}(x, x) + 1 & \text{if } c = \mathit{TRUE} \\ n + f_{mul}(x, x) + 2 & \text{if } c = \mathit{FALSE} \end{cases} \tag{1}$$

In the parametric-WCET literature, the derived parametric expressions are in the form of polynomial functions where the variables refer to input parts or properties that influence loop iteration numbers. The derived parametric expressions are usually expressed conditionally over subdomains of the input space. In the literature, if the derived parametric WCET of some program p has the form $w_p = a_k x^k + a_{k-1} x^{k-1} + \cdots + a_0$, it means that the program have parts that have execution times $a_i, i \in [0..k]$ and are repeated $x^i, i \in [0..k]$ times where x is a part or a property of the input to program p. In these types of expressions, x is a variable that will control the iteration number of one or more loops in the program under analysis. In this paper, we will focus on deriving parametric WCET expressions where the final expression is polynomial. Deriving more complex parametric expressions that account for variable-latency instructions or those that are expressed conditionally over input space is left for future work.

4 Parametric WCET Analysis using Genetic Programming

Parametric WCET analysis requires knowledge about the input parts or properties that affect the execution time of the program — and which will appear in the final parametric expression. We assume in this paper that knowledge about the exact parts or properties of the input that influence the WCET is available to us through some third-party analysis or user-input; and we exclusively consider those that affect loop bounds and/or recursion depths as we have discussed. We will refer to the input parts and properties that appear in the final parametric expression as parameters.

The problem of deriving parametric WCET expressions reduces to a problem of symbolic regression. Let p be the program under analysis. Program p is executed for a number of m

```
// The input to this program is composed of the following fields:
//   1. two integer arrays,
//   2. an integer variable,
//   3. and a boolean variable.
private static int foo (Integer[] A, Integer[] B, int x, boolean c)
{
        // The size of the array (n/m) is a property of the input.
        // Assume that execution time of the following segment is 1.
        int n = A.length;
        int m = B.length;

        // This is a part of the program whose execution time depends on
        // some property of the input. This is a case where the input
        // affects execution counts.
        for(int i=0; i<n; i++)
        {
                // Do something.
                // Assume that execution time of this body is 1.
        }

        // This is a part of the program whose execution time depends on
        // some part of the input. This is a case where the input affects
        // execution times directly assuming multiplication is variable-
        // latency instruction in the target hardware.
        // Assume that execution time of multiplication of a and b is
        // f_{mul}(a,b).
        int y = x*x;

        // This is a part of the program whose execution time depends on
        // some part of the input. This is a case where the input affects
        // execution counts in a conditional manner.
        if(c)
        {
                for(int i=0; i<m; i++)
                {
                        // Do something.
                        // Assume that execution time of this body is 1.
                }
        }
        else
        {
                // Do something.
                // Assume that execution time of this part is 1.
        }
        return 0;
}
```

Figure 2 Some function `foo()` that illustrates different scenarios of how input influences the execution time of a program.

inputs: in each run $i \in [1..m]$ the values $x_{1,i}, x_{2,i}, \cdots, x_{n,i}$ of all n parameters $x_1, x_2, \cdots, x_n$ are recorded together with p's end-to-end execution time t_i. After all runs have completed, we obtain a m-by-$(n+1)$ matrix of real numbers. The objective then is to find the function w_p that satisfies $t_i = w_p(X_i), i \in [1..m]$ where X is a vector defined as $X =< x_1, x_2, \cdots, x_n >$ and X_i is the value of X in run i i.e. $X_i =< x_{1,i}, x_{2,i}, \cdots, x_{n,i} >$. Let X^k be the vector that contains the parameters $x_1, x_2, \cdots, x_n$ raised to the power k i.e. $X^k = x_1^k, x_2^k, \cdots, x_n^k$.

The parametric expression w_p is just a function that expresses the execution time of the program p in terms of its parameters. It becomes a parametric *WCET* expression once the m runs exercise different program segments in their worst-case execution scenarios. This means that if the expression w_p has the form $w_p = A_k X^k + A_{k-1} X^{k-1} + \cdots + A_0$ where $A_i, i \in [0..k]$ is a vector of constants, then w_p is WCET expression if the values in the vectors $A_k, A_{k-1}, \cdots, A_0$ — which correspond to program-segment execution times — are maximum. This depends on the quality of testing which is a common issue in all measurement-based and end-to-end WCET analyses. Here we will focus on using the GP's symbolic regression to learn a parametric expression as opposed to trying to prove that the factors $A_k, A_{k-1}, \cdots, A_0$ have their maximum-possible values — which is a separate problem to solve outside this paper.

In order to discover the parametric expression, the GP is informed with the types of operators and terminals (variables and constants) that can potentially appear in the derived parametric expression. As we have discussed before, the operators will appear in the non-leaf nodes and the terminals will appear in the leaf nodes of the AST representation of the parametric expression. In our case, we have limited our attention to polynomials; which by their mathematical definition allow addition and subtraction of terms, each term can be composed by multiplying variables by variables or variables by constants, raising a variable to the power of a constant, or dividing a variable by a constant. Therefore the set of operators to consider is {addition, subtraction, multiplication, division}.

In addition to this, depending on implementation, the GP can benefit from knowledge about the shape, depth, and size of the ASTs that represent the target parametric expressions. A binary AST is usually the choice for symbolic regression problems where the target expression is a polynomial. The depth of the binary AST will influence the order of the polynomial. For example, the parametric expression of Figure 1 requires an AST of depth 5 (assuming root depth is 0) for a polynomial of order 3.

Notice that two runs of the GP (based on the same data set) that are performed to derive the parametric expression w_p of some program p are not guaranteed to derive the same parametric expression i.e. they will potentially result in two expressions w_p and w'_p such that $w_p \neq w'_p$. However, if the two runs of the GP are given the same resources (time, processing power, memory, etc.) it is unlikely that the structure of the parametric expressions will be different; but their constant factors are likely to differ slightly. The reason for this is that GP is a search-based method that is based on some element of randomness in the genetic operations such as cross over. The GP converges towards a solution of some fitness — after a number of generations — which is often almost the same in different runs of the GP on the same problem instance — as long as the GP has access to the same resources in these different runs. Here, “almost the same” is seen in the form of parametric expressions that have the same AST but have slightly-different constant factors.

5 Evaluation

To evaluate our approach, we use the Mälardalen WCET benchmarks [10] which have historically been used to evaluate parametric WCET approaches in the literature. The benchmarks have been modified to make them amenable to end-to-end testing — basically by allowing the function `main()` to take input arguments, processing them using the function `atoi()`, and passing them to the function of interest (e.g. `factorial`, `insertion_sort`, etc.). The benchmarks we used are described in Table 1 and they include all benchmarks used in [3, 16, 9, 11, 15, 6, 2, 5, 1] except those not available in [10]. The benchmark `janne_complex` has been augmented with a new parameter c that substitutes the constant value 30 used in the outermost loop — to make it more interesting to analyse.

Each benchmark program in Table 1 is compiled for the ARM architecture using a *gcc* cross compiler and executed on Simplescalar [4] using the configuration shown in Figure 3. The reason behind using Simplescalar instead of actual hardware is that we don’t have access to actual hardware and accompanying execution-time measurement equipment. The reason behind using the configuration of Figure 3 is to top the most complex hardware platform used in parametric WCET analysis by [2] where the authors use MPC565 for evaluating their work.

Table 1 The benchmark programs used in the evaluation.

Benchmark	Description	Parameters and Ranges
bsort100	Bubble Sort (Triangular Loop)	Size n of array to be sorted $n \in [1..100]$
cnt	Matrix Count	Size n of side of square matrix $n \in [1..30]$
crc	Cyclic Redundancy Check	Size n of input string $n \in [1..200]$
fac	Factorial	Integer n $n \in [1..100]$
fir	Finite Impulse Responder	Variables in_len, $coef\_len$, and $scale$ $in\_len, coef\_len, scale \in [1..700]$
insertsort	Insertion Sort	Size n of array to be sorted $n \in [1..100]$
janne_complex	Nested Loop Program	Variables a, b, and c $a, b, c \in [1..100]$
matmult	Matrix Multiplication	Size n of side of square matrix $n \in [1..20]$
sqrt	Square Root Computation by Taylor Series	Integer n $n \in [1..100]$
st	Statistics Program	Size n of arrays to be processed $n \in [1..300]$

```
# Pipeline                      -res:imult 1                    -cache:il2 none
-fetch:ifqsize 4                -res:memport 2                  -cache:dl2 none
-decode:width 1                 -res:fpalu 1
-issue:width 1                  -res:fpmult 1                   # Branch Predictor
-issue:inorder false                                            -bpred taken
-issue:wrongpath true           # Cache
-lsq:size 2                     -cache:il1 il1:128:16:2:1       # Default settings are used
-res:ialu 1                     -cache:dl1 none                 # for everything else.
```

Figure 3 Configuration of the Simplescalar architecture used in the experiments.

The experimentation setup is straightforward. Each benchmark program p is executed $m = 10,000$ times using randomly generated inputs. The number m was chosen to be at the same time large enough to allow more input diversity, and small enough not to affect the GP's runtime too severely — since the fitness function computes a sum of differences of complexity $\Theta(m)$. In each run, the value X_i of the parameters of interest X and the end-to-end execution time $t_i, i \in [1..m]$ of p are captured by reading the value sim_cycle generated by Simplescalar at the end of each execution of the program under analysis. It is worth noting that the end-to-end execution time corresponds to the entire program, not just the function of interest and consequently the parametric timing expression corresponds to the function `main()` — including the use of the function `atoi()` and all initializations such as array initialization; and the function of interest. Measuring the end-to-end execution time of the actual function of interest inside the benchmark program reduces to parsing the Simplescalar trace (generated via `-ptrace`). This adds an unnecessary overhead to

Table 2 The results of the experimental evaluation.

Benchmark p	**Expression** w_p **by** ***Eureqa***	**Error** e_p	**Time to find** w_p
bsort100	$35n^2 + 917n + 5.55e4$	± 0.20	2 minutes
cnt	$988n^2 + 100n + 5.82e4$	± 0.22	2 minutes
crc	$180n + 2.04e5$	± 0.20	1.5 minute
fac	$52n + 5.60e4$	± 0.0012	10 seconds
fir (*)	$1706 in\_len + 4.45e4$	± 0.57	10 minutes
insertsort	$13n^2 + 970n + 5.54e4$	± 0.28	2.5 minutes
janne_complex	$20.78c - 17.46a + 5.80e4$	± 0.05	10 seconds
matmult	$189n^3 + 1753n^2 + 74n + 6.24e4$	± 0.02	4 minutes
sqrt	$31n + 5.76e4$	± 0.13	5 seconds
st	$2302n + 5.93e4$	± 0.26	2.5 minutes

the experiment because the parametric expression can be derived regardless of whether the end-to-end execution times are measured at the function level, or the whole-program level.

After the m executions, we pass the resulting m-by-$(n+1)$ data matrix M_p of data to the GP to perform symbolic regression. We use the rows $[1..0.9m]$ for training i.e. evolution, and the remaining $[0.9m+1..m]$ for testing i.e. measuring the error of the derived parametric expression when applied to unseen data. For the GP, we used both *Eureqa Formulize v0.96* (a standalone application based on [12] and freely downloadable from [7]) and *gptips v1.0* (a library for MATLAB based on [14] and freely downloadable from [13]). The results we show in this paper are those obtained by *Eureqa* because it runs faster than the MATLAB-based *gptips*, and it also comes with the paid option of using a cloud cluster to accelerate computation. However we did not use the cloud cluster in our experiments, but it would be interesting to use it for future work. It is worth noting that *gptips* is open-source and allows more customization of the GP such as introducing specialized functions to use in symbolic regression, and also user-crafted genetic operators. Such customization ought to accelerate evolution, but we have not tested this. The experiments took place in a desktop computer of the specifications *Intel(R) Core(TM)2 Duo CPU @2.80 GHz* running *32-bit Windows 7 Professional* with *4Gb of RAM.*

Table 2 summarizes the findings. The GP is run for each benchmark program's data matrix for 10 minutes. Notice that the GP can be left running indefinitely over some data matrix M_p until an optimal symbolic expression w_p is obtained. We argue that 10 minutes is a reasonable time to leave the GP running for the relatively-small problems dealt with in this work — which (i) have small numbers n of independent variables that affect the size of the ASTs generated by the GP, (ii) have small numbers of observations m that affect the cost of the fitness operation, and (iii) have a known polynomial structure and consequently the GP's search space is pruned because only relevant operators are used during evolution. The best solution after 10 minutes is the one shown in Table 2. The error e_p in the parametric expression w_p is also computed and it is the average of the sum of $|w_p(X_i) - t_i|$ in the $0.1m$ unseen runs. So if the error in the parametric expression is $e_p = 0.01$, it means that the average value for the difference $|w_p(X_i) - t_i|$ per unseen run is 0.01. The recorded execution time of the GP in Table 2 is the first instant in time in which the best solution is found. For example, for the benchmark program `bsort100`, the derived expression that is shown in Table 2 has been found after 2 minutes, and did not improve over the remaining 8 minutes during the 10-minute execution of the GP.

The sources of the errors e_p in the table come from the execution-time variations imposed by the hardware architecture. For example, let program p contain a flat loop — and no other loop — that iterates x times where x is some input variable to p; then its parametric expression has the shape $w_p = ax + b$ where a is the execution time of the body of the loop, and b is the execution time of everything else outside the loop. In this case, it is perfectly possible to witness the following scenarios during execution: (i) when $x = 2$, $a = 100$; (ii) when $x = 3$, $a = 95$; and (iii) when $x = 4$, $a = 110$. This could happen if x controls some conditional statement before the execution of the loop, and hence alters the execution history prior to the execution the loop which leads to different values of the execution time a of the loop's body. In cases like this, it is very hard if not impossible for the GP to derive an expression w_p with error $e_p = 0$ because of the complex program-hardware interaction that creates what can be referred to as "noise" in the data matrix M_p.

The GP was unable to find an accurate parametric expression for the benchmark program `fir` with the three parameters specified in Table 1 (error had average magnitude $e_{fir} = 52$) because of the complex interactions between these variables and their effect on program flow — which cannot be captured by a polynomial. The set of parameters was reduced to one element namely *in_len* while the other two parameters were fixed to their original values 35 and 285 in [10]. The one-parameter expression is the one shown in Table 2 which still has the worst accuracy.

Other than that, the GP was able to approximate the parametric expressions of the benchmark programs with great accuracy despite the hardware exhibiting variations in execution times due to the presence of out-of-order execution, cache, and branch prediction. The expressions were derived in a record time which never exceeded few seconds/minutes per benchmark program. The shape of the parametric expressions corresponds to loops and loop nests in the benchmark programs, and their ASTs are isomorphic with the ASTs derived for the same programs using static parametric WCET in the literature. The obvious differences are in the constant terms and factors which correspond to execution times — which are expected to be different because (i) the hardware platform is different, and (ii) end-to-end testing is used instead of static analysis.

6 Related Work

A relatively recent review on WCET analysis is provided by [17] where different analysis methods, calculation techniques, and available tools are described and compared. In this section we shall exclusively review parametric WCET analysis which has been the topic of research in [3, 16, 9, 11, 15, 6, 2, 5, 1].

Bernat and Burns [3] present a tree-based approach for parametric timing analysis where they derive algebraic expressions for parts of the code, link them together according to the tree structure to build larger expression, and then use software tools such as Maple to simplify the parametric expressions. The technique is manual as no tool was implemented for it, it takes information about the parameters affecting the WCET through a system of code annotation, and uses a timing model supplied by the user or a third-party analysis.

Vivancos et al. [16] use a path-based approach where parametric expressions are derived using fixed-point caching behaviour. The technique is not described in great detail because of the focus on applications of parametric timing analysis. However, the authors do mention limitations in their technique namely the ability to only handle well-structured non-recursive code, and inability to handle cases where there are nested parametric expressions inside loop nests.

Van Engelen et al. [15] introduce a method for parametric WCET analysis using Newton-

Gregory formulae that handles rectangular and non-rectangular loop nests of seemingly arbitrary structures. Unfortunately, the work is purely theoretical and lacks evaluation on actual benchmark programs with the exception of small code snippet. The work here is included in the review for the sake of completeness only.

Coffman et al. [6] use a summation solver called *Emtadel* that automatically derives the number of times the body of a traingular loop nest executes; in the form of a polynomial expression involving the use of min/max operators. The use of Emtadel comes as the answer to the problem of nested parametric loops the authors encountered in [16]. The main problem with this approach is that there is no reference to where to download Emtadel or similar software to reproduce the results obtained by the approach.

Altmeyer et al. [2] present an automatic technique for performing parametric WCET analysis of executable code based on dependency analysis between program variables and parameters. The technique identifies the parameters automatically by performing a simple read/write analysis of memory cells and CPU registers, and using the variables that are read from before written to as the analysis parameters. The dependency analysis is used to form relationships between the variables that control loop iterations and the parameters of the program.

Althaus et al. [1] present a parametric-analysis technique that exploits the usually-regular structure of code written for critical real-time applications — in particular where loops have single entries. They present an algorithm of polynomial-time complexity in practice and exponential-complexity in theory which works on executable code and derives the parametric expressions of loops. For code that does not satisfy the single-entry loop property, they present transformation techniques to force the property which unfortunately adds to the complexity of the approach.

Lisper et al. [9, 5] use polyhedral flow analysis together with symbolic integer-linear programming to derive parametric WCET expressions. The analysis is very accurate but at the cost of prohibitive complexity: the derived parametric-WCET expressions are too complicated to be evaluated instantly during runtime — unless manual simplifications are applied to them. In addition to this, the analysis requires extensive resources as it failed for some problem instances.

The technique we present is different from all previous techniques in the sense that it is based on end-to-end runtime measurements as opposed to static analysis, and uses off-the-shelf GP tools which gives it the advantage of amenability to industrial use. Table 3 shows a comparison of our technique with the techniques in the literature using the following metrics (columns in Table 3).

- **Input.** This metric refers to the way by which the parametric analysis identifies the variables that appear in the parametric expression i.e. they are derived automatically, semi-automatically, or manually.
- **Automation.** This metric refers to whether or not the parametric analysis (excluding input information) is fully-automatic, semi-automatic, or manual.
- **Operators.** This metric refers to the operators supported in the final parametric expression namely arithmetic operators ($+$, $-$, $\times$, $\div$), the conditional operator, and the max/min operators.
- **Limitations.** This metric refers to the main drawbacks of the method that might hinder its use in practice.

Table 3 Comparison with parametric WCET analysis techniques.

Technique	**Input**	**Automation**	**Operators**	**Limitations**
Ours	Manually-derived (or user-provided).	Fully-automatic	Arithmetic	Does not come with safety guarantees.
Bernat and Burns [3]	Manually-derived: user-provided.	Manual	Arithmetic, conditional.	No implementation is available.
Vivancos et al. [16]	Semi-automatic: the analysis looks for all parametric loops only and reads their induction variables.	Fully-automatic	Arithmetic	Well-structured non recursive code only. No support for nested-parametric loops. Not clear how variable bounds on loop-induction variables are related to program input.
Coffman et al. [6]	Semi-automatic: the analysis looks for all parametric loops only and reads their induction variables.	Fully-automatic	Arithmetic, max/min.	Approach based on a software package that is unfindable on the web.
Altmeyer et al. [2]	Semi-automatic: only constant-offset dependencies can be derived automatically.	Fully-automatic	Arithmetic, conditional.	User input needed for to resolve non constant-offset dependencies, such input is very unlikely considering it is based on disassembled binary code.
Lisper et al. [9, 5]	Manual	Semi-automatic	Arithmetic, max/min.	Technique does not scale well, and fails for some problem instances. Parametric expressions are overly-complex which hinders their use in dynamic scheduling — unless manually simplified.
Althaus et al. [1]	Not clear.	Fully-automatic	Arithmetic, conditional.	Complexity polynomial in practice but can grow to exponential in theory.

7 Conclusions and Future Work

In this paper we have shown how to perform parametric timing analysis using genetic programming. End-to-end execution times of the program are recorded together with values of parameters that trigger them, and are then input to the genetic program which performs symbolic regression to discover the parametric expressions. The technique has been successful in deriving accurate parametric expressions.

Genetic programming can be used in conjunction with static-analysis methods for parametric timing analysis to validate their results, and should be of great interest to industry where end-to-end measurements are the way forward to performing WCET estimation. The technique can be explored further in the the following directions (to list but a few): (i) investigate the use of the method on more substantial case studies, (ii) augment the proposed technique by automatic identification of parameters that affect the WCET, (iii) apply the method on programs that run on actual hardware, and (iv) incorporate variable-latency instructions in the parametric expressions.

References

1 E. Althaus S. Altmeyer and R. Naujoks. Precise and efficient parametric path analysis. *SIGPLAN Not.*, 46(5):141–150.

2 S. Altmeyer, C. Humbert, B. Lisper, and R. Wilhelm. Parametric timing analysis for complex architectures. In *Proceedings of the 2008* 14th *IEEE International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA'08)*, pages 367–376, August 2008.

3 G. Bernat and A. Burns. An approach to symbolic worst-case execution time analysis. In *Proceedings of the 25th Workshop on Real-Time Programming, Palma, Spain*, June 2000.

4 D.C. Burger and T.M. Austin. The simplescalar tool set, version 2.0. Technical Report CS-TR-1997-1342, University of Wisconsin, Madison, 1997.

5 S. Bygde, A. Ermedahl, and B. Lisper. An Efficient Algorithm for Parametric WCET Calculation. In Patrick Kellenberger, editor, *The* 15th *IEEE International Conference on Embedded and Real-Time Computing Systems and Applications, RTCSA 2009*, pages 13–21. IEEE Computer Society, August 2009.

6 J. Coffman, C. Healy, F. Müeller, and D. Whalley. Generalizing parametric timing analysis. *ACM SIGPLAN Notices*, 42(7):152–154, 2007.

7 Cornell Creative Machines Lab. Eureqa. *http://creativemachines.cornell.edu/eureqa*, April 2012.

8 J.R. Koza. *Genetic Programming: On the Programming of Computers by Means of Natural Selection (Complex Adaptive Systems).* A Bradford Book, 1 edition, December 1992.

9 B. Lisper. Fully Automatic, Parametric Worst-Case Execution Time Analysis. In *Proceedings of the 3rd International Workshop on Worst-Case Execution Time (WCET) Analysis*, pages 77–80, Porto, July 2003.

10 Mälardalen WCET Research Group. WCET project/benchmarks. *http://www.mrtc.mdh.se/projects/wcet/benchmarks.html*, April 2012.

11 S. Mohan, F. Müeller, W. Hawkins, M. Root, C.A. Healy, and D.B. Whalley. Parascale: Exploiting parametric timing analysis for real-time schedulers and dynamic voltage scaling. In *26th IEEE International Real-Time Systems Symposium (RTSS'05)*, pages 233–242, 2005.

12 M. Schmidt and H. Lipson. Distilling free-form natural laws from experimental data. *Science*, 324(5923):81–85, 3 April 2009.

13 D.P. Searson. GPTIPS: Genetic Programming & Symbolic Regression for MATLAB. *http://sites.google.com/site/gptips4matlab/*, April 2012.

14 D.P. Searson, D.E. Leahy, and M.J. Willis. GPTIPS : An Open Source Genetic Programming Toolbox For Multigene Symbolic Regression. In *Proceedings of the International Multiconference of Engineers and Computer Scientists 2010 (IMECS 2010)*, volume 1, pages 77–80, Hong Kong, 17-19 March 2010.

15 R.A. van Engelen, K.A. Gallivan, and B. Walsh. Parametric timing estimation with Newton-Gregory formulae: Research Articles. *Concurrency and Computation: Practice and Experience*, 18(11):1435–1463, September 2006.

16 E. Vivancos, C. Healy, F. Müeller, and D. Whalley. Parametric timing analysis. *SIGPLAN Not.*, 36(8):88–93, 2001.

17 R. Wilhelm, J. Engblom, A. Ermedahl, N. Holsti, S. Thesing, D. Whalley, G. Bernat, C. Ferdinand, R. Heckmann, T. Mitra, F. Müeller, I. Puaut, P. Puschner, J. Staschulat, and P. Stenström. The Worst-Case Execution-Time Problem–Overview of Methods and Survey of Tools. *ACM Transations on Embedded Computing Systems*, 7(3):1–53, 2008.