Accurate Determination of Loop Iterations for Worst-Case Execution Time Analysis

Mark Bartlett, Iain Bate, *Member, IEEE,* and Dimitar Kazakov

Abstract—Determination of accurate estimates for the Worst-Case Execution Time of a program is essential for guaranteeing the correct temporal behaviour of any Real-Time System. Of particular importance is tightly bounding the number of iterations of loops in the program or excessive undue pessimism can result. This paper presents a novel approach to determining the number of iterations of a loop for such analysis. Program traces are collected and analysed allowing the number of loop executions to be parametrically determined safely and precisely under certain conditions. The approach is mathematically proven to be safe and its practicality is demonstrated on a series of benchmarks.

Index Terms—C.3.d Real-time and embedded systems, C.4 Performance of Systems, D.2.5 Testing and Debugging, I.2.3 Deduction and Theorem Proving and Knowledge Processing

1 Introduction

A large number of embedded systems have constraints not just on their functional behaviour, but on their temporal behaviour. Most commonly, such constraints are deadlines on the time by which action must have been taken or output should be available. In hard real-time systems, the penalty for not meeting these deadlines can be catastrophic, for example in areas such as automotive braking systems, aircraft control systems or nuclear applications.

Ultimately underlying the ability to guarantee that deadlines will be met, is knowledge of the maximum time a task will take to execute in the worst case, the Worst-Case Execution Time (WCET) [1].

Due to theoretical limitations, this quantity cannot be computed accurately in the general case. Rather, in situations in which deadlines must be guaranteed to be met, static analysis is used to produce an estimate of the WCET. A value obtained in this way can be proven to not underestimate the true WCET of the analysed task and hence can be safely used in scheduling the system. However, the estimate will not in general be precise, hence some pessimism is introduced into the estimate, which can lead the designers to make unnecessary (and costly) overprovision of hardware resources.

One particularly acute cause of pessimism in programs can be the overestimation of the number of iterations of a loop that will be taken [2]. Determining such a quantity exactly is in general an undecidable problem, and even slight overestimates will result in adding an amount of pessimism proportional to the execution time of the whole loop body. Furthermore, tighter estimates will lead to better reasoning about the behaviour of caches. The problem is compounded in nested loops, particularly those where the number of executions of an inner loop body is dependent on the iteration of the outer loops currently being performed. In such cases, ignoring these interactions can lead to a massive degree of pessimism. Precision in this aspect will be reflected in tighter WCET estimates. The current state of the art is to manually encode those special cases where dependencies exists [3].

Despite being one of the first problems identified in WCET analysis [4], the determination of loop iterations remains an active area of research [3], [5], [6]. This paper presents and builds on an approach to this problem based on model inference [7]. Rather than deduction of the number of loop executions from the task's code, we propose the *induction* of this information from examples of the task's execution. The number of executions thus predicted is a parametric formula in the program's variables. It is proven that the number of loop iterations can be inferred safely and precisely for a certain class of nested loops, those whose bounds are Presburger expressions [8]. Presburger expressions are a limited subset of general arithmetic expressions which are often used in real-time systems due to their decidability. However, they are sufficiently rich to allow quite complex behaviour to be implemented. For example, Chapman [2] reports on a real aircraft engine controller with around one million lines of code which has loops corresponding to this subset. As the obtained estimates for the number of iterations are safe for this set, they can be used by existing static analysis techniques to yield safe overall WCETs. Indeed, as these formulae are parametric in program variables, they can be used in parametric WCET analysis [9], [10].

The current paper extends this method to deal with the case of loops with bounds not conforming to the Presburger subset. Such an extension allows for the method to be utilised in a wider set of programs. This modified version of the technique is demonstrated to be applicable in this more general case, though safety can

• *M. Bartlett, I. Bate and D. Kazakov are with the Department of Computer Science, University of York, York, YO10 5DD.*
E-mail: {mark.bartlett,iain.bate,dimitar.kazakov}@cs.york.ac.uk

no longer be guaranteed. A treatment is also given for situations in which zero entry loops may exist. The technique is thoroughly evaluated on a range of benchmarks and synthetically generated examples. It is shown to be useful in generating correct and exact formulae for the number of loop iterations. The time needed is shown to be practical for use in real-world applications.

The remainder of this paper is organised as follows. Section 2 presents examples motivating the work. This is then followed by the presentation of related work in Section 3. The idea of model inference for WCET analysis is then presented in Section 4. Section 5 presents a mathematical proof that the technique presented in this paper will correctly identify loop iterations for a certain class of loops. The development of a tool utilising the results of this proof is then described in Section 6. The tool is then evaluated against both synthetic and benchmark examples in Section 7 before final conclusions are shown in Section 8.

2 Motivating Examples

The research presented in this paper can be motivated through two examples taken from the Mälardalen WCET benchmark suite.[1]

Results using further examples from this suite of programs are reported in Section 7. However these two examples illustrate the problem and our solution well, so will be referred to throughout in greater detail.

For the first of these examples, the number of iterations on each entry of inner loops is dependent on the current iteration of the outer loops. For the second, the number of loop executions is a logarithmic function of an input parameter. Both of these issues are potentially problematic in WCET analysis. The following subsections explain these examples in further detail.

2.1 LU Decomposition

The LU decomposition benchmark[2] uses matrix arithmetic to find the solution to a system of n equations in n unknowns. This algorithm is a good example of code in which failing to account for non-rectangular effects in loops can lead to a massive overestimation in the number of times the body of each loop will be executed, and hence, in the overall estimate, WCET will be greatly pessimistic.

For example, the fifth loop is nested inside the fourth loop, both of which are dependent on the loop counter of the outermost loop for the number of executions that should be performed. Neglecting non-rectangular effects, one would estimate that the upper bound on the total number of executions of the body of the fifth loop that can be performed is equal to the product of the maximum number of iterations that each of these three loops can perform. This is equal to n^3. In reality, the total number of executions of this loop is merely $\frac{1}{6}n(n+1)(n+2)$, virtually a sixth of the estimated value for sufficiently large n.

Full results reporting the formulae inferred for each loop (and the amount of pessimism avoided) are reported in Section 7.1. The time required for this inference is also presented in Section 7.3.

2.2 Fast Fourier Transform

The fast Fourier transform benchmark[3] implements an efficient method for converting sampled data into the frequencies from which it was formed. Algorithms based on this are therefore frequently used in signal processing parts of embedded systems.

The specific form of fast Fourier transform used is the Cooley-Tukey algorithm. This completes the conversion process through recursively splitting the problem into two equal parts. For this reason, many of the loops in the benchmark execute a number of times which is either the logarithm to the base 2 of the number of data points, or a function of this value. This presents a challenge as most techniques are geared towards returning polynomial formulae for the number of loop executions.

An extension to our basic system is needed to handle such cases as they fall outside the Presburger subset. This will be explored in Section 6.3. The results of applying this technique to this benchmark are subsequently shown in Section 7.2, while the times required to do so are given in Section 7.3.

3 Related Work

3.1 Worst-Case Execution Time Analysis

As previously noted, the WCET cannot be found exactly in general. Rather there are currently two major approaches to estimating WCET: measurement-based techniques and static analysis.

Measurement-based techniques [1] require the task to be run on many different inputs and from many different initial hardware states. The estimated WCET is then taken to be the worst observed in practice, perhaps with an extra margin of safety added on. Techniques exist to increase the probability that the actual WCET is observed, for example by using coverage criteria [11] or using genetic algorithms to guide test case generation [12]. However short of (frequently infeasible) exhaustive testing, it is impossible to guarantee that the WCET will be encountered during testing. As a result, the estimate thus obtained will likely be close to the real WCET (tight) but could be an underestimate, and therefore deadlines could be missed.

In contrast, static analysis [1] is based on computing the WCET from models of the program semantics and hardware platform. Typically, this is done using a three stage process, see Fig. 1(a). At the first stage, the basic

1. http://www.mrtc.mdh.se/projects/wcet/benchmarks.html
2. http://www.mrtc.mdh.se/projects/wcet/wcet_bench/ludcmp/
3. http://www.mrtc.mdh.se/projects/wcet/wcet_bench/fft1/

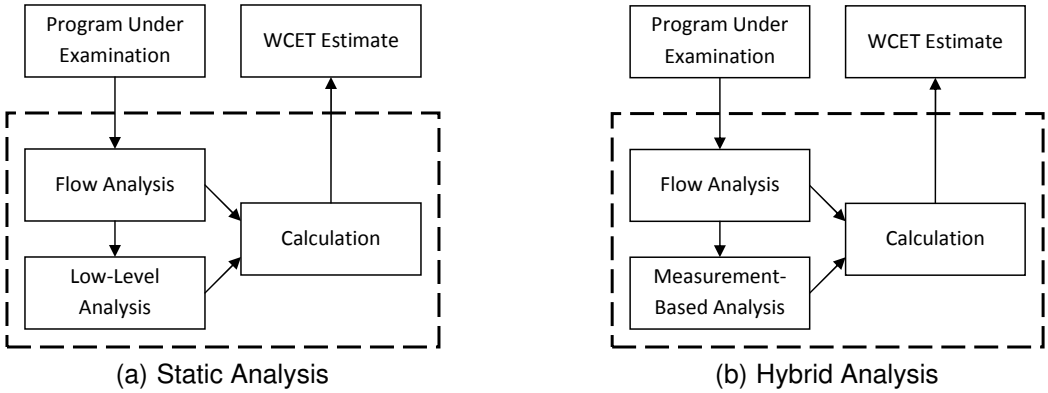


Fig. 1. The 3 stages of static analysis and hybrid analysis

blocks of the program (non-branching sequences of instructions) are identified and the flow of control between these is determined. This information is then used by the low-level analysis to determine the execution time of each basic block on the target hardware. Finally, the flow analysis and low-level analysis are combined to yield a system of equations which is solved to obtain the WCET estimate.

Due to the theoretical impossibility of calculating execution time exactly in general, generalisations are made at each stage of the static analysis that guarantee the calculated value will be at least the real WCET (safe), but may be an overestimate (pessimistic). Having a safe estimate makes the quantity usable in safety-critical systems, but the pessimism can result in under-utilised systems and money wasted on unnecessarily fast hardware.

A third method, hybrid analysis [13], (see Fig. 1(b)) combines the two other approaches. In essence, the low-level analysis stage is replaced with measurement-based analysis of each basic block. This removes the issue of creating good models of the hardware but, depending on the method used in the calculation phase, may either remove the guarantee of safety or lead to extra pessimism in the resulting estimate.

3.2 Loop Iteration Determination

In current approaches to WCET estimation, loop iteration determination is performed by static analysis at the program flow stage. It is then subsequently used in constructing the equations to be solved at the calculation phase.

Computationally, the simplest way to determine loop iterations is to be supplied with the information by the programmer through annotations to the program [14]. This approach was that which was used originally. However, such a method relies on the programmer knowing and supplying accurate information and updating the annotations correctly when the code changes. This can be verified using theorem proving [2], [15], but other issues remain. Needing to use unannotated library functions creates a problem, as does the introduction of code optimisations (most obviously loop-unrolling) by the compiler. For these reasons, automated analysis should be preferred [16].

One approach towards automation is utilised by the aiT tool.[4] This tool is equipped with various manually derived patterns corresponding to particular styles of loop generated by particular compilers. These patterns are then matched against the code under examination to give tight estimates of the number of iterations. In essence, the task has been transformed from manually supplying annotations for each loop, to manually supplying annotations for each common type of loop. While this overcomes the problems noted above, the task is not fully automated in so far as a programmer must still manually derive and supply the patterns needed — not an inconsiderable task. As these are different patterns for each compiler, as well as for different compiler options, the generality of the technique is limited. Furthermore, more complex loops cannot be handled [3].

More recently, several papers in the field have presented approaches based on data flow analysis [3], [5], [6], [17]. The details of these techniques differ, but follow the same general principle. The variables on which loop exits depend are identified, as is the way in which the values of these variables can change during each loop iteration. Given knowledge of the initial states of these variables and what values may trigger exits, safe estimates can then be made of the maximum number of iterations possible.

4 Framework and Problem Domain

4.1 Model Inference Approach to WCET Analysis

The technique to be presented in this paper for determining loop iterations can be seen as a specific application of a more general technique for estimating WCET. This technique is based on inferring models from example program traces, which are then used to augment the existing static analysis approach. This is illustrated in Fig. 2 (cf. Fig. 1). In order to see why such an approach

4. http://www.absint.com/ait/

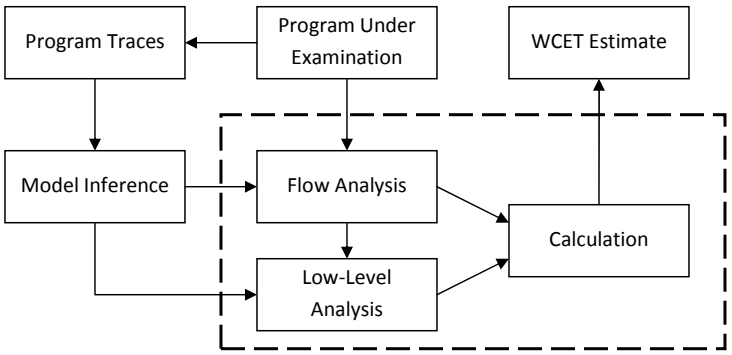


Fig. 2. Model inference based WCET analysis

is needed, it is first necessary to highlight two issues that exist with current static approaches.

Two major problems are present with the existing static analysis approach. Firstly, the pessimism that results from these analyses can be too excessive to make the estimates useful for any real-world application except where safety must be guaranteed with absolute certainty. A decade ago, Chapman [2] found that even for quite simple programs and simple hardware, 30% pessimism was not uncommon and in one case 64% pessimism was observed. More recently, an industrial case study [18] found pessimism of 4–33% when the aiT tool was used to analyse industrial code. This was approximately doubled if hand-coded annotations were not given.

The second problem with static analysis is the difficulty of building models for use by the analysis. At the hardware level, complex components such as pipelines, branch predictors and multi-level caches have been introduced into the processor market, some of which are used in the majority of processors available for embedded systems. These are included to reduce the average-case execution time, though often they are also likely to reduce the worst-case execution time too. However, without an accurate model of how they work, very pessimistic assumptions about their behaviour must be made in WCET analysis. Wilhelm et al. [19] state that "the architecture determines whether a static timing analysis is practically feasible at all and whether the most precise obtainable results are precise enough ... This dependence on the architectural development is of growing concern to the developers of timing-analysis tools and their customers, the developers in industry". For modern processors, the functioning of these components is frequently commercially confidential. Even where this is not the case, the complexity of the these features may make creating appropriate models difficult, such as is the case with out-of-order processors [20]. Where these features are used, static analysis is difficult without considerable effort or manufacturer support.

Similarly, appropriate software models may be difficult to construct. For example, the pattern matching approach to loop bound determination used by the aiT tool requires considerable human reasoning to configure for each compiler used. Ultimately, the user may be required to supply additional information about program flow in the form of annotations to their code in order to ensure tighter estimates than can otherwise be found solely through the automated analysis of the code.

The emerging approach of model inference to assist in static analysis seeks to overcome the second of these two problems and may be useful in reducing the first. The core of the approach is that the construction of models for the static analysis is automated rather than manually performed. The data from which these models are constructed is obtained through observation of the hardware and software in operation. In this approach, there is no need for the programmer of the analysing tool to constantly create new models when new hardware starts being used in systems or to support optimisations made by each individual compiler.

The approach seems very similar to current hybrid approaches, but actually differs in some important aspects. While both techniques rely on adding dynamically obtained data to a static approach, this data is utilised in different ways. In the case of hybrid analysis, the data obtained is simply the runtime of basic blocks (or known sequences of these). This information is then fed into the analysis in a minimally processed way, either in its raw form or only allowing for interblock effects etc. to be determined. In contrast, the model inference approach relies on extracting (potentially non-temporal) information from the execution and then processing it. The results fed into the static analysis are not the data obtained, but rather a model inferred from this data, which may not correspond to what has been observed in any individual run.

In general, this approach will suffer from one of the main drawbacks of hybrid analysis, namely the safety of the static analysis approach will be compromised by admitting data obtained from non-exhaustive testing. However, under certain limitations, the models obtained by this approach can be proven to be safe. Even when this outcome cannot be guaranteed, the models may still be useful in soft-real time systems in order to provide potentially unsafe but tighter estimates of WCET. Such

estimates will reflect the limits of typical behaviour observed rather than unusual but possible pathological worst cases.

In addition to the current work presented here on loop bound identification, two other works have been published on the use of model inference for other aspects of WCET analysis. Bate and Kazakov [21] show how the technique could be used to identify which of a set of branch-predictors a processor uses, based on observations of branch predictions and mis-predictions. The model of the identified branch predictor could then be fed into a static analysis, significantly reducing the amount of pessimism from a situation in which no knowledge of the predictor was available.

Lisper and Santos [22] use model inference to create a timing model for the length of execution of each basic block. Rather than use raw data for this timing model, as would be done by hybrid analysis, they infer a WCET for each block, which may be greater than any observed in practice. In a situation in which the timing behaviour of the hardware is quite simple, it is proven that the model inferred will be safe, or even exact, as is the case with the results presented in this paper. However, in contrast to the approach presented here, they present a holistic technique for analysing the execution time of the entire program. Here, our emphasis is on supplementing the existing static approach rather than replacing it.

4.2 Model Inference For Loop Iteration Determination

The task of loop iteration determination requires the number of times each loop is executed to be either determined precisely or safely bounded. Where the number of iterations may vary, the estimated number of iterations should be parametric in the program's variables rather than a simple numeric upper bound. This permits the deduction of tighter WCET estimates at the calculation stage and additionally permits parametric WCET to be undertaken [9], [10]. Furthermore, this allows for the number of loop executions in a procedure to be found as a function of the procedure's inputs, allowing tight estimates from multiple calling contexts.

To apply the technique of model inference to this domain therefore, the task becomes one of inferring a formula for each loop, relating the number of loop executions to the program's variables. Methods for determining which variables should be considered are well established in the field of WCET analysis [23] and hence will not be further discussed in this paper. Similarly, methods for tracking the execution pathway of the program also exist [24]. This latter matter however is returned to in Section 6.1 in order to outline which methods are suitable for obtaining the loop execution counts for this technique.

As the means to gather suitable data already exist in the field of WCET estimation, the novelty of this approach is in inducing the formulae from these data. The details of this are presented in Section 6 alongside a tool constructed for this purpose.

While well-structured loops in general are considered in this paper, particular attention is given initially to the discussion of Presberger loops. These are loops in which the guard conditions are restricted to a particular class of arithmetic expressions over a set of variables [8]. It is shown, in Section 5, that for this class of loops that the models inferred can be guaranteed to be safe and indeed exact. Extensions to this class are considered in subsequent sections, revealing that, while not guaranteed to be safe, formulae for other loop patterns can be inferred correctly under this methodology.

4.3 Presburger Expressions and Loops

Definition 1: The set of Presburger expressions of a set of variables X, $PE(X)$, can be defined as follows

$$PE(X) = \left\{ pe(X) \;\middle|\; pe(X) = \alpha_0 + \sum_{i=1}^{n} \alpha_i x_i \right\}$$

where $X = \{x_1, x_2, \ldots x_n\}$ and $\forall j \in [0, n], \alpha_j \in \mathbb{Q}$ and the notation $pe_i(X)$ is used to denote an arbitrary member of $PE(X)$ throughout the remainder of the paper.

In other words, a Presburger expression over a set of variables is a linear function in those variables in which each numeric constant is a rational number.

This class of expression lacks arbitrary interaction of variables (preventing multiplication of two variables for example) and, as a result, formulae using members of the class are decidable. For this reason, Presburger expressions are often used in control expressions in safety critical programs, in which reasoning over the algorithm must be performed [14], [25]. This paper is initially concerned with such a use of Presburger expressions for controlling the number of loop iterations.

Fig. 3 illustrates the general case of nested Presburger loops. For each loop, the number of iterations is determined by two Presburger expressions, both of which are defined over the set of exogenously supplied variables, V, and any loop counters from outer loop levels. Note that none of the variables in V can have their values changed within the context of the loops, nor may a loop counter's value be changed except by the loop's control expression. We also begin by restricting our treatment to the case where the lower bound is less than or equal to the upper bound.

Previous approaches to determining the number of loop iterations for the Presburger class of nested loops have been proposed which utilise static analysis methods rather than the measurement-based approach adopted here [26], [27], [28]. It can be shown that the upper and lower bounds of each loop in a nest can be used to form a polyhedron. The task of calculating the number of iterations of the nested loop is then equivalent to counting the number of integer points within this shape [26]. However in the general case, even determining whether there are any integer points in the shape

```
for l_1 = pe_1a(V) to pe_1b(V)
  for l_2 = pe_2a(V ∪ {l_1}) to pe_2b(V ∪ {l_1})
    ⋮
    for l_n = pe_na(V ∪ {l_i|i ∈ Z+, i < n}) to
              pe_nb(V ∪ {l_i|i ∈ Z+, i < n})
      Innermost Loop Body
    next
    ⋮
  next
next
```

Fig. 3. Generalised nesting of Presburger loops

is NP-complete [26]. The research in this area therefore concentrates on finding algorithms which are efficient for realistic types of loop nest with reasonably low nesting depth.

However, the issue of obtaining the appropriate loop control variables and determining the upper and lower bounds for each loop is far from trivial. Patterns for common loop types could be matched against but, as previously stated, these must be created for each setting of each compiler that will be used. Automated attempts extract loop iterators based on flow analysis are ongoing [3], [5], [6]. The approach presented here is an alternative technique that bypasses this issue altogether. Rather than perform any deep analysis of the semantics of the code, we instead rely solely on the established methods used by static analysis programs to identify the location of loops in the code and extract variables *possibly* influencing the number of iterations. Admission of excess variables which actually do not affect the number of loop iterations do not adversely affect this technique (except for in the number of traces needed). Nor is it necessary to determine upper and lower bound conditions for each loop.

5 Proof of Sufficient Observations

This section presents a mathematical proof that the number of iterations of a set of nested Presburger loops can be uniquely determined from a finite and quantified number of example observations of its behaviour. Specifically, it is shown that the number of iterations is a polynomial function in the set of exogenously defined variables. From this, it follows that the number of observations needed to uniquely determine this polynomial can be determined as a function of the depth of loop nesting and the number of exogenous variables.

The proof proceeds in three stages. Firstly, it will be shown that the number of executions of a nested loop body can be written as a polynomial of known maximum degree (Section 5.1). The number of terms in this polynomial will then be demonstrated (Section 5.2). Finally, the observations necessary to uniquely determine the single correct function from the class of all functions of this form will then be shown (Section 5.3). By the combination of these stages, it follows that the resulting observations from the final stage are sufficient to correctly learn the number of loop body executions.

5.1 Number of Executions of a Nested Loop

Before deriving the functional form for the number of executions of a nested loop, it is necessary to derive a lemma that the later proof is conditional on.

Lemma 1: $\forall pe(V) \in PE(V)$, $\sum_{l=0}^{pe(V)} l^n$ can be written as a polynomial of maximum degree $n+1$ over the set of variables V.

Proof: By Faulhaber's Formula [29]

$$\sum_{l=0}^{p} l^n = \sum_{k=1}^{n+1} c_k p^k$$

for some numeric coefficients, c_k. Substituting $p = pe(V)$ and expanding yields,

$$\sum_{l=0}^{pe(V)} l^n = c_1 pe(V) + c_2 pe(V)^2 + \ldots + c_{n+1} pe(V)^{n+1}$$

As each c_k is a polynomial of degree 0 and $pe(V)$ is a polynomial of maximum degree 1 in V, this sum can be rewritten as the sum of $n+1$ polynomials in V, with the greatest possible degree being $n+1$. Trivially, this can be shown to simplify to a single polynomial in V of maximum degree $n+1$. □

Theorem 1: The total number of executions of the body of the innermost loop of a nested set of Presburger loops, as shown in Fig. 3, in which a loop iterator's upper value is never exceeded by its lower value, can be expressed as a polynomial of maximum degree n over the set V, where n is the number of nested loops and V is the set of exogenous variables.

Proof: The proof is by induction in the number of nested loops.

For n = 1, by program semantics, the number of loop body executions is

$$\sum_{l_1=pe_{1a}(V)}^{pe_{1b}(V)} 1 \quad = \quad pe_{1b}(V) - pe_{1a}(V) + 1$$

which, from the definition of a Presburger expression, is a polynomial in V of degree 1.

For $n > 1$, the number of innermost loop executions is

$$\sum_{l_1=pe_{1a}(V)}^{pe_{1b}(V)} \sum_{l_2=pe_{2a}(V\cup\{l_1\})}^{pe_{2b}(V\cup\{l_1\})} \cdots \sum_{l_n=pe_{na}(V\cup\{l_i|i\in\mathbb{Z}^+,i<n\})}^{pe_{nb}(V\cup\{l_i|i\in\mathbb{Z}^+,i<n\})} 1$$

Assuming inductively that the theorem holds for the $n-1$ inner loops, this is equal to

$$\sum_{l_1=pe_{1a}(V)}^{pe_{1b}(V)} f_{n-1}(V \cup \{l_1\})$$

where the notation $f_N(X)$ is used throughout to indicate a polynomial of degree N over X.

This can be expanded to yield (for some constant α)

$$\sum_{l_1=pe_{1a}(V)}^{pe_{1b}(V)} [f_{n-1}(V) + f_{n-2}(V) \times l_1 + \ldots + f_1(V) \times l_1^{n-2} + \alpha l_1^{n-1}]$$

Expanding out the summation operator, this yields a sum of n terms, each of which is the product of a maximum degree k polynomial in V and the term $\sum_{l_1=pe_{1a}(V)}^{pe_{1b}(V)} l_1^{n-1-k}$. From Lemma 1 it follows that this latter term can be rewritten as a polynomial of maximum degree $n-k$ in V, therefore the above equation is equivalent to the sum of n terms each of which is a polynomial in V of maximum degree n. This can simply be shown to be equal to a single polynomial in V of maximum degree n.

Hence, the theorem is true for n nested loops if it is true for $n-1$ nested loops. As it is true for $n=1$, it is therefore true $\forall n \in \mathbb{Z}^+$. □

5.2 Terms in the Expansion

In the previous section, it was shown that the number of loop iterations could be rewritten as a polynomial of maximum degree n, where n was the depth of loop nesting. It is now necessary to establish the number of terms in such a function, for which it will later be shown that coefficients can be straight-forwardly derived.

Theorem 2: The number of terms in the canonical form of a polynomial of degree n in V is

$$\binom{n+|V|}{|V|}$$

Proof: The result is a consequence of the number of ways of choosing n items from $|V|+1$ with replacement when order is not important.

Any polynomial of degree n can be written as the product of n polynomials of degree 1. Hence each of the terms in the degree n polynomial is formed by taking one variable term or the constant from each degree 1 polynomial and multiplying them together. As there are $|V|+1$ terms in each degree 1 polynomial and n of these polynomials, this corresponds to choosing from $|V|+1$ items n times. Because the same term can be chosen from multiple degree 1 polynomials and multiplication is commutative, this is equivalent to choosing $|V|+1$ from n with replacement and with order unimportant. From combinatorics, this result is known to be equal to $\binom{n+(|V|+1)-1}{(|V|+1)-1}$ which simplifies to $\binom{n+|V|}{|V|}$. □

5.3 Sufficient Observations for Discrimination

There are an infinite number of polynomials of degree n over the set of variables V, each differing only in the coefficients of the terms and the value of the constant term. Therefore, in order to uniquely distinguish any one of these polynomials from all others of the same form, it is sufficient to establish these numeric values. It has been shown that the number of loop body executions is a function of this form, subject to certain constraints on conditionals controlling the loops. In order to determine exactly which function, the numeric values must be found.

By observing examples of the execution of loops it is possible to determine the values of all relevant exogenous parameters and also the number of innermost loop executions which occurred. The ways in which this observation may be done are discussed in Section 6.1. Substituting the information thus obtained from a single observation in to the polynomial functional form derived previously yields a linear equation in the (as yet) unknown coefficients of the polynomial equation. From the previous theorems, it follows that there are $\binom{n+|V|}{|V|}$ of these coefficients.

Using Gaussian elimination, it is possible to solve a system of linear equations in p unknowns when the system contains p such equations, none of which are collinear. It therefore follows that the number of examples needed to uniquely distinguish a function of the form derived above from all others of the same form is $\binom{n+|V|}{|V|}$.

Taken in conjunction with theorems 1 and 2, it follows then that from the observation of $\binom{n+|V|}{|V|}$ examples of the loop nest being executed, an exact function describing the number of iterations for any input can be determined. This subject only to the restrictions that the loop nest is of the correct form, zero-entry loops are not possible and that the equations to be solved by the Gaussian elimination algorithm corresponding to two example executions are not collinear.

6 Inferring Loop Iterations from Program Traces

A tool has been implemented to infer loop iterations from program trace data using the method described in the previous section.

This tool is based on Inductive Logic Programming (ILP) [30], a type of machine learning in which the data to be learned from, the theory that is learned and the space of possible hypotheses to consider are all expressed in first-order logic. As a mature field, we believe machine learning may contain many methods that are useful in taking the area of WCET analysis forward. Many aspects of WCET tasks are incredibly well-suited to the strengths of machine learning in general and ILP in particular; the data to learn from are typically noiseless, deterministic, discrete and available in whatever quantity is required. Many of the open questions in machine learning research relate to the cases in which data do not possess these properties. For data of the type encountered in WCET analysis, standard off-the-shelf tools and techniques already exist.

Specifically in the current paper, the tool is implemented in the learning environment of Aleph [31] using Progol syntax. The present version of the tool is an

Input: A set of pairs of variable values and the associated observed iteration count
Output: A polynomial formula relating iterations to variable values

for $i = 1$ to Max_Depth
  Try to use Gaussian elimination to find a polynomial of degree i that is consistent with the input data
next
for $j = 1$ to Max_Depth
  Test data against the degree j polynomial calculated earlier
  if Degree j polynomial fits data exactly **then**
    return Degree j polynomial
  end if
next
return No polynomial found

Fig. 4. The loop iteration formula inferring algorithm

evolution of that which was described in detail in [32] (and first presented in an early form in [33]). The current version produces the same results as the previously described version, differing only in finding the formula more quickly due to two changes. Firstly, the underlying algorithms have been rewritten to be more efficient using techniques such as tail recursion, difference lists and dynamic programming, none of which have changed the functional operation of the code. Secondly, the equation search space has been tuned slightly to achieve the same results while considering fewer theories. An overview of the workings of the tool now follows at a level sufficient to replicate its functionality; readers interested in lower level implementation details and improvements made are referred to [32]. An outline of the algorithm behind the tool is shown in Fig. 4.

Input to the tool consists of a list of Progol predicates of the form `target(A,B)` where `A` is a list of values for program variables which, it is believed, affect the loop iteration count, and `B` is the observed number of loop iterations when the loop has been executed with the given values in `A`. At present, this input must be compiled into this form manually. In future versions of the tool, it is envisaged that the relevant variables will be found through flow analysis of the program code (or some representation of it, such as a control flow graph) and that the iteration count will be extracted from an appropriate program trace. Methods for automating both of these aspects already exist (for example [23], [34]) making integration primarily an engineering task.

The tool attempts to fit various equations to this data set through the choice of suitable coefficients. Specifically, it tries to find a coherent polynomial in the given program variables using Gaussian elimination, as described in the previous chapter. Beginning with a degree 1 polynomial, the tool constructs candidate polynomials up to a specified degree. From the previous chapter, it follows that this is equivalent to looking in turn for a formula which could be produced by an unnested loop through to one that could be produced by a nesting with equal depth to the highest order polynomial considered. In the current implementation, a depth of up to 8 loops is considered, which should prove sufficient for realistically encountered code. Amending the tool to consider a greater depth can be achieved trivially by the alteration of constants. However, the greater the maximum depth to consider, the longer the tool will take for analysis. A depth of 8 levels is chosen in order to cover almost all loops likely to be encountered in practice without placing excessive overhead on the time to infer from the more common simpler loops.

Having attempted to find polynomials consistent with the observed behaviour, the program selects the one of lowest degree which is fully consistent with the input data. Assuming that the loops are of the required form and that sufficient observations were available (which can be checked using the formula derived in Section 5) then this can be shown to be the lowest degree polynomial to give the correct number of iterations for all inputs; no simpler polynomial could be correct or it would be chosen in preference and the given degree polynomial must be right by the previously established proof.

It is worth noting that the search through polynomials of varying degrees could be avoided if the method were to be used in conjunction with a static analyser. In this case, examination of the control flow graph would reveal the loop nesting depth for each loop body. Given this information, the maximum degree of the polynomial can be deduced as shown in the earlier proof. Coefficients could then be computed for this polynomial without the search for the relevant degree which is carried out in the current tool.

Output from the tool consists of a Prolog predicate representing the polynomial function found to predict the number of loop iterations. This is then transformed into the equivalent polynomial in a more natural form, such as would be found in a manual annotation of program code. Adding this annotation back to the code under examination or passing it to a static analyser is a trivial step to be performed in integrating the tool in to a larger WCET analyser.

Results obtained with this tool are presented in Section 7.

6.1 Instrumenting Code

In order to infer the correct formula, it is necessary to obtain data from program runs. By necessity, this entails some manner of instrumentation. There are multiple techniques that can be and are used in measurement-based analysis to examine both the functional and temporal behaviour of systems. The various methods (including logic analyzers, in-circuit emulators and injecting instrumentation code) are well known, as are the

trade-offs that exist between altering the behaviour of code and simplicity of collection [24].

For the presented approach, the specific data that must be gathered consists of the value of several variables at the beginning of the loops and the number of executions of each loop. As there is no necessity to deal with the timing behaviour of the code, rather only control flow information, very temporally intrusive instrumentation can be used, providing the flow is not altered, and the techniques by which the data can be collected are very broad. Furthermore, the number of loop iterations performed is a property of the code and is independent of the target hardware. Therefore simulated execution can be used to give results that remain valid on the actual hardware platform.

In the results presented in this paper, instrumentation is achieved through addition of extra code to the program under examination. This outputs the values of the relevant variables and tracks the number of entries to each loop body. In general, such instrumentation may be too intrusive and alter the timing behaviour of the code, but is acceptable for this approach for the reasons noted above.

For the sake of simplicity, the analysis reported in this paper is performed at the source code level, and hence the code is instrumented at this level too. This makes the results obtained on the benchmark cases comparable with results obtained by other techniques; for an analysis at an intermediate or compiled level, the actual code examined may differ between approaches due to the use of different compilers. Validation of the results obtained is also simpler. However, as it is ultimately compiled code that is executed, analysis at the source code level would not be as useful in a real world setting. In some cases, the structure of the source code does not correspond to that of the compiled code. In particular for the current application, the number of loop executions may vary due to compiler optimisations such as loop unrolling. In any case, as long as the compilation process does not introduce unstructured flow or non-Presburger expressions, the presented technique will work equally well at the object code level.

6.2 Zero-Entry Loops

The proof previously presented, that the formulae can be correctly inferred from a given number of observations, is contingent on the lower bound of each loop being less than or equal to its upper bound. However, this condition is not always met in real code, for example Fig. 5 shows a loop where the upper bound may be less than the lower bound depending on the value of the variable X. Where X is greater than 10, an implementation of such code in most programming languages would result in the body of the loop never executing. In some languages, it is possible that this could result in the body being executed either once or infinitely, but only the more usual case will be considered here. In order for

for $l_1 = X$ to 10
  Loop Body
next

Fig. 5. A loop which may have zero entries of the body

the method presented to be generally applicable, code with such zero-entry loops must be permitted. Note that we need only deal with those loops for which only some inputs cause the zero-entry behaviour to be exhibited; if the loop cannot be entered for any input value (for example if it can be proved that X would always be greater than 10 when the snippet in Fig. 5 is encountered) then it is impossible to generate examples from which to induce a formula.

The problem which occurs where zero-entry loops occur is that the function for the number of loop executions is not smooth as in the simpler case where upper bounds are not less than lower bounds. Instead, they must be expressed as a piecewise function of the input variables, where the number of intervals grows exponentially with the depth of nesting.

As our approach is unable to infer such functions, a method to detect these situations is instead implemented. Through appropriate instrumentation of the code, it is possible to detect when a zero-entry situation has occurred; the head of the loop will be seen to be evaluated and anything other than the loop body will execute subsequently.

Two options are available if this case occurs. Firstly, the fact that this situation can happen may be just be noted and the user informed. No formula will be generated, but the tool detects the limitation and therefore does not generate an incorrect formula. Secondly, the cases in which the zero-entry loops are observed may be discarded and a formula learned as normal from the other observations. Calculation of roots of the inferred formulae at each level of the loop nesting can then be used to reveal the range for which the inferred formulae are positive and hence for which these formulae are applicable. This may be useful if it can be assumed or proven that the usual mode of execution is within this range, for example if X can be proven to be less than 10 in Fig. 5.

6.3 Beyond Polynomial Functions

As formulated, the inference engine is only capable of learning formulae which are polynomial expressions of the input variables. This has been shown to be sufficient where the components of the loop guards are Presburger expressions. In fact, it remains sufficient for loop guards that are any polynomial expression, though the degree of the resulting expression may be higher than for an equivalent number of nested Presburger loops.

However, the number of executions of some loops are not polynomial expressions in the exogenous variables. Most commonly, such loops are encountered in divide

and conquer algorithms, which split the problem to solve in half and hence have a total number of executions based on the base 2 logarithm of the problem size. The fast Fourier transform previous discussed in Section 2.2 is one such algorithm.

In order to allow the technique presented to function on such programs, there are two ways in which alterations may be made. Firstly, the range of functions composing the functional hypothesis space could be increased to allow the inference engine to explore a greater range of functional forms, including logarithms, exponentiation, etc. However, this would entail significantly rewriting the inference engine. It would also increase the number of functions to consider significantly resulting in a longer time taken to learn the correct function, even in the cases where the functional form was eventually found to be solely polynomial. Finally, it can be shown through theoretical considerations that if the range of learnable functional forms was increased to allow any possible function then for any observed dataset, there would be an infinite number of formulae consistent with it and no way to choose between them.

Therefore, a second, alternative technique has been adopted to deal with these non-polynomial cases. In this approach, the inference engine is not altered, but rather the inputs given are enriched with additional inputs, each derived from one or more of the default exogenous variables. For example, rather than provide two variables as input for induction, four variables may be supplied, the standard input variables and their respective logarithms. The resulting formula would then consist of a polynomial in which terms were either exogenous variables or logarithms of the exogenous variables. While still polynomial in the input variables supplied, the fact some input variables are based on others means that the expression is not polynomial in the base exogenous variables.

Augmenting the input to the inference engine with all conceivable (or even likely) derived inputs would massively increase the size of the input vector. In Section 7.4, it will be shown that increasing this size results in a corresponding exponential increase in the time needed for induction. Therefore, adding large amounts of additional inputs is an infeasible technique. However, an alternative exists.

Rather than add all derived inputs at once, a series of runs of the inference engine can instead be made. On the first run, the unaltered inputs are used. On subsequent runs, inputs are augmented with a single type of derived inputs (for example, logarithms). Once all runs with only a single type of derived input have been conducted, runs with pairs of derived inputs are performed. Then three types of additional input and so on. Execution of these runs is halted as soon as a formula consistent with the observed data is induced. Such a scheme implements a bias towards finding the simplest consistent formula possible, but still allows a more complex formula to be found. Additionally, the simplest formulae are found without any additional search to the basic Presburger case considered before and therefore requiring no extra time. While such an approach requires multiple runs of the inference engine, this can still be undertaken in an automated way.

This method assumes that the likelihood of a formula being correct is positively correlated with the simplicity of its terms. Such a bias conforms to Occam's razor. However, if more information is known about the likely functional forms in a given domain, then this could be used to alter the order in which derived inputs are tried in order to decrease the average number of runs needed.

7 Results

This section presents results of using the tool outlined in the previous sections. Except where necessary and noted, only the basic version of the tool is utilised, i.e. inputs consist of only the exogenous variables and not any variables derived from these and the possibility of zero-entry loops is not considered. As previously stated, the results presented are based on analysis at the source code level. Times reported are the times for inference and do not include time to gather the required observations.

The results of evaluating the tool against the two motivating examples shown in Section 2 are presented. This is then followed by an analysis of the time taken to infer these results and those for another benchmark from the Mälardalen WCET benchmark suite. Finally, the scalability of the tool is explored through experiments on a range of synthetic examples. The use of synthetic examples allows the number of variables and depth of nesting to be controlled and set, making a systematic investigation possible.

7.1 LU Decomposition Benchmark

Table 1 shows the formulae for the number of executions of each loop in the LU decomposition program introduced in Section 2.1. The first formulae are those that would be obtained using a simple method that just multiplied the maximum number of iterations for each level together, that is to say, a method that can accurately infer the maximum iteration of each loop but which is unable to account for the non-rectangular effects. For loop 3, it has been assumed that the flow analysis would be able to correctly identify the iterations for which the if condition was not satisfied; if not, the overestimation for this loop would be even greater. The formulae inferred by our method are then given, as are the actual number of iterations that the code produces.

As can be seen, the number of iterations using the inference method always match the actual number of iterations. As would be expected, the simple method produces a formula which is always greater or equal to the correct one, for any given n. This overestimation is also shown as a function of n in table 1.

This result confirms the mathematically proven results from the earlier section and also illustrates the need for

TABLE 1
Formulae for the number of entries of each loop in the LU decomposition algorithm. Formulae are shown for the predicted number of iterations under the simple bound scheme, the number inferred by the learning algorithm, the actual number that occur and the number of overestimated iterations of the simple bounds versus the actual number. The number inferred always matches the actual number.

Loop Number	Simple Bounds	Inferred Iterations	Actual Iterations	Overestimation
1	n	n	n	0
2	n^2	$\frac{1}{2}n(n+1)$	$\frac{1}{2}n(n+1)$	$\frac{1}{2}n(n-1)$
3	$(n-1)^3$	$\frac{1}{6}n(n+1)(n-1)$	$\frac{1}{6}n(n+1)(n-1)$	$\frac{1}{6}(n-1)(n-2)(5n-3)$
4	n^2	$\frac{1}{2}n(n+1)$	$\frac{1}{2}n(n+1)$	$\frac{1}{2}n(n-1)$
5	n^3	$\frac{1}{6}n(n+1)(n+2)$	$\frac{1}{6}n(n+1)(n+2)$	$\frac{1}{6}n(n-1)(5n+2)$
6	n	n	n	0
7	n^2	$\frac{1}{2}n(n+1)$	$\frac{1}{2}n(n+1)$	$\frac{1}{2}n(n-1)$
8	n	n	n	0
9	n^2	$\frac{1}{2}n(n+1)$	$\frac{1}{2}n(n+1)$	$\frac{1}{2}n(n-1)$

a method that accommodates non-rectangular effects in loops.

Evidence that there is a necessity to deal with non-rectangular effects, using a technique such as this, is provided in table 2. Here, the overestimates shown in the previous table are enumerated for various values of n to show how much pessimism this can result in. The effect of the pessimism is obviously greater for the more deeply nested loops. It is notable however that even for relatively low values of n, the amount of pessimism experienced grows very rapidly. For example, even in a 10×10 matrix, over 350% pessimism will be present in the estimate for the number of executions of instructions inside loop 5.

7.2 Fast Fourier Transform Benchmark

The fast Fourier transform benchmark (see Section 2.2) was introduced to highlight the issue of dealing with loops with non-Presburger loop guards. Specifically, several of the loops in this algorithm execute a number of times related to the logarithm base 2 of the input variable.

Table 3 shows the results of running the inference tool on observations of this benchmark. Two results are reported for each loop, the inferred formula when the tool is used in its basic form and the inferred formula when the input was enriched with the logarithms as described in Section 6.3.

The table shows that several of the loops had formulae that were expressible without the need for logarithms. While there was no practical need to infer from observations of these loops again using logarithms, such results are reported for completeness. In all cases, the same formula was still inferred as was expected, illustrating that providing additional unneeded variables does not interfere with the tool's functioning.

For the remaining loops, the basic inference process failed to compute formulae for the number of iterations. As the correct formulae were not able to be considered by this inference process, this was the correct behaviour. However, when the input was increased to include both the input variable and its logarithm, formulae for all loops were correctly learned. It should be noted that these expressions are no longer polynomials of the input variable n, but are polynomial in n and $logn$.

These results validate the applicability of the technique, suitably modified, in domains where the loop guards are not Presburger expressions.

7.3 Benchmark Timing Results

Having demonstrated the usefulness of the technique in the previous sections, the practicality is now assessed. Specifically, the time required for the tool to infer formulae is measured to evaluate its potential for real world use.

In addition to the two benchmark examples previously explored in detail, the Mälardalen benchmark suite was

TABLE 2
Overestimation in the number of entries to each loop of the LU decomposition algorithm if non-rectangular effects are neglected. Overestimation is shown for various values of n, the number of rows of the matrix to be decomposed.

n	Loop number								
	1	2	3	4	5	6	7	8	9
1	0%	0%	—	0%	0%	0%	0%	0%	0%
10	0%	82%	342%	82%	355%	0%	82%	0%	82%
100	0%	98%	482%	98%	482%	0%	98%	0%	98%
1000	0%	100%	498%	100%	498%	0%	100%	0%	100%
10000	0%	100%	500%	100%	500%	0%	100%	0%	100%

TABLE 3
Inferred formulae for fast Fourier transform function with and without logarithms.

Loop number	Without Logarithms	With Logarithms
1	Not found	$logn$
2	Not found	$logn$
3	$n-1$	$n-1$
4	Not found	$\frac{1}{2}nlogn$
5	$n-1$	$n-1$
6	Not found	$n-logn-1$
7	n	n

examined to identify any other benchmarks suitable for analysis by the tool. In order to be analysable by the tool, the benchmark needed to contain loops in which there was only a single, non-data-dependent exit point. Additionally, only benchmarks containing nested loops with non-rectangular effects were considered due to the trivial formulae associated with all appropriate unnested loops identified. These criteria led to the identification of only one additional benchmark, that for matrix inversion. For some of the loops in the matrix inversion benchmark, the loops were not of a form which could be analysed by the tool (for example, one loop had a return statement within its body) nevertheless, analysis was attempted on all loops.

The times taken for the analysis are reported in Table 4. Each formula found by the tool has been checked against the code and found to be exact and therefore safe. For the loops in which formulae could not be induced, each was found to violate some aspect of the requirements for analysability by the tool, such as multiple exit points. Inference times for each of the loops were remarkably consistent at between 0.28 and 0.38 seconds each. Such times would certainly be acceptable in real-world applications.

7.4 Scalability of the Approach

While testing on the established benchmarks in the field is useful, doing so fails to establish the general applicability of the technique. It is widely acknowledged that the benchmark suite is very limited and not necessarily reflective of code that requires analysis in the real-world. For example, in the benchmarks studied, the maximum depth of loop nesting encountered was three, while only one or two variables were ever found to affect the iteration count. Unfortunately, real-world code is not generally available either. Where access to such code is possible, confidentiality prevents the code that a tool is evaluated on being placed in the public domain or described in sufficient detail to permit repeatability of the findings. Without access to similar real-world code, it is impossible to compare one's own methods against published techniques.

Due to the limited number of programs which are analysable and interesting in the benchmark suite, and the lack of suitable real-world code, testing of this tool is best performed using a test harness. This harness takes a number of variables and a depth of loop nesting as input and returns a randomly generated program with the corresponding characteristics as output. In order to ensure that the formulae for the number of loop iterations is of the maximum possible degree and complexity, the start and end values for every loop counter is a Presburger expression of all exogenous variables and all outer loop counters. This testing methodology allows for a more systematic test of the tool. By altering the two inputs to the harness, it is possible to establish how well the technique scales as these factors vary.

The test harness is allied to a simulator for the code. This takes the generated programs and runs them with various values of the program's input parameters, outputting the information required by the tool for its inference. Specifically, it generates a file containing a list of binary Prolog predicates recording the input values given and the observed number of innermost loop executions. Sufficiently many predicates are generated for each program to allow the iteration count to be uniquely determined as described in Section 5.

In Section 5, it was shown that, for the type of loop considered here, a specific number of observations would be sufficient for the number of iterations to be determined. Namely $\binom{n+|V|}{|V|}$ examples of the loop's behaviour must be observed. As a first step in examining the scalability of the technique, this function is shown in Fig. 6. It can be seen that the number of examples that must be generated is nearly exponential in both the maximum depth of nesting and in the number of variables considered.

However, for real-world code nesting of the level graphed here is rarely encountered [2]. In order to find practical application, it is unlikely that a tool would ever need to deal with more than 8 levels of nesting, and seldom with more than four. At these depths, a few thousand observations are likely to be more than sufficient, depending on the number of variables that it is necessary to consider. Program slicing [23] may also be used to further reduce the number of variables to consider, and hence observations needed. Such quantities of data can be practically generated in a reasonable timeframe to make the technique usable.

The second issue of scalability relates to the time that is needed for inference once the data has been collected. In order to assess this, the test harness was used to generate random programs from which data was collected and inference conducted. The resulting times taken for inference are shown in Fig. 7. All times have been collected from a PC with an Intel T2300 1.66GHz processor with 1GB of RAM running Microsoft Windows Vista. As the loops generated in the sample programs were maximally complex in terms of interaction effects between nested loops, the values shown in this graph represent an upper bound on the time needed to learn in each situation in the general case.

The time taken for the inference can be seen to increase

TABLE 4
Times taken for inference of benchmark programs. For loops marked with a cross, (×), formulae could not be inferred by the tool.

Benchmark	Loop	Time (s)
LU Decomposition	1	0.375
	2	0.281
	3	0.297
	4	0.328
	5	0.343
	6	0.296
	7	0.281
	8	0.296
	9	0.328
Fast Fourier Transform	1	0.359
	2	0.343
	3	0.359
	4	0.328
	5	0.328
	6	0.327
	7	0.327
Matrix Inversion	1	0.390
	2	0.312
	3 (×)	0.375
	4	0.359
	5	0.343
	6 (×)	0.327
	7	0.312
	8 (×)	0.358
	9 (×)	0.343
	10	0.374

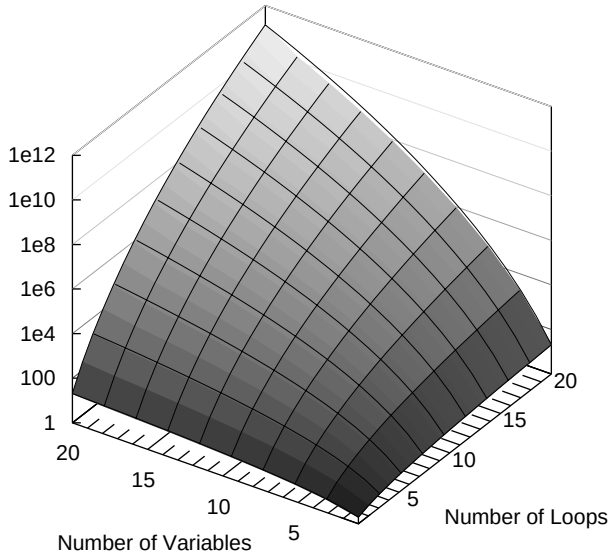


Fig. 6. Sufficient observations to infer the number of loop iterations for a given number of variables and depth of loop nesting.

greatly with the number of variables, but appears almost independent of the depth of loop nesting. The invariance with depth is due to the tool used to perform the inference process. This first generates the set of best fitting polynomials up to the maximum degree to consider and then subsequently tests to see which of these is the lowest order that fits. Given that generating potential functions takes substantially longer than testing these, an almost equal amount of work is done regardless of the complexity of the function returned. Altering the tool to test the functions as it generates them would presumably lead to the time taken and depth of nesting becoming positively correlated.

In contrast, increasing the number of variables has a large effect on the time taken for inference. This is in line with what might be intuitively expected. Increasing the number of variables leads to a larger number of equations being needed during the Gaussian elimination process and a larger number of terms in each of them. This also appears to mirror the increase in the number of examples needed for inference, see Fig. 6.

As was also noted for the number of observations that must be generated, the values obtained here appear acceptable for real-world applications especially for the range of parameter values likely to be encountered in practice.

In order to test the iteration formulae learned for correctness, each formula was tested against a further set of examples of the loops' behaviour, obtained using the same test harness as was used to generate the examples for inference. The test data was formed by limiting each variable to the range 0 to 10, and exhaustively exercising the loop. This resulted in 10, 100, 1000 or 10000 observations, depending on the number of variables under examination. In every case, the formulae were found to be consistent with the test data. This was to be expected given the proof of the validity of the method in Section 5 and simply confirms the correct implementation of the tool.

One final aspect of these results that could be reported is the WCET estimate that could be obtained if these iteration formulae were included in a full analysis. However, this is not done here. Doing so would only obfuscate the pertinent issues with extraneous information. The generated code is simply a shell of loops into which any code might be placed. The approach could be made to show arbitrarily good improvement over a simpler method of deriving loop iterations by inserting increasingly large amounts of code inside the innermost loop or by choosing hardware in which branch misprediction was particularly detrimental. Any such form of evaluation therefore becomes meaningless and hence is not reported here.

8 Conclusions

This paper has presented a dynamic technique for the determination of the number of iterations of a loop, or nested loops, for use in calculating tight estimates of Worst-Case Execution Time. As has been shown throughout, the benefits from deriving tighter estimates

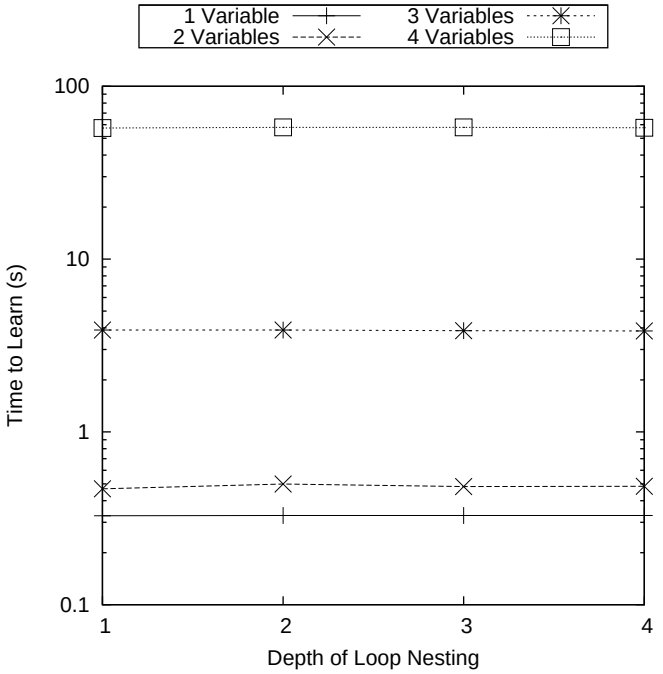


Fig. 7. Time taken to infer loop iteration formula for given number of variables and depth of loop nesting.

in non-rectangular loops can be very significant. The approach presented differs from those currently used in that it is closer to a black box technique. Whereas other automated techniques rely heavily on analysing the code used, the new technique presented merely requires that the loops within the program can be identified as such and that the values of given variables may be obtained. This should make it more easily portable to new languages and overcome issues of compiler specific analysis that blight other techniques. The method has been proven to produce exact loop counts for well-structured code in which the loop guards are Presburger expressions.

Specific to this paper has been the demonstration that the technique can still be of use in situations in which loop guards do not meet these strict criteria. The method has been shown to be adaptable to provide potentially useful estimates in these cases, but these can no longer be guaranteed to be safe. Two causes of violations of the loop conditions have been considered. Firstly, the applicability of the technique was shown in situations in which non-Presburger conditions were present. Secondly, methods of identifying and coping with zero-entry loops were presented.

The results presented have shown the applicability of the technique to both standard benchmarks and more complex manufactured examples. Inference of the correct formulae for nests of loops up to four levels deep and with up to four variables has been shown to require tens of seconds on a typical desktop computer. For the standard WCET benchmarks studied, the computation time is even lower.

REFERENCES

[1] R. Wilhelm, J. Engblom, A. Ermedahl, N. Holsti, S. Thesing, D. Whalley, G. Bernat, C. Ferdinand, R. Heckmann, T. Mitra, F. Mueller, I. Puaut, P. Puschner, J. Staschulat, and P. Stenström, "The worst-case execution-time problem — Overview of methods and survey of tools," *Transactions on Embedded Computing Systems*, vol. 7, no. 3, pp. 1–53, 2008.

[2] R. Chapman, "Static timing analysis and program proof," Ph.D. dissertation, University of York, UK, 1995.

[3] C. Cullmann and F. Martin, "Data-flow based detection of loop bounds," in *7th International Workshop on Worst Case Execution Time (WCET) Analysis*, C. Rochange, Ed. Dagstuhl, Germany: Internationales Begegnungs und Forschungszentrum für Informatik (IBFI), Schloss Dagstuhl, Germany, 2007.

[4] C. Park and A. Shaw, "A source level tool for predicting deterministic execution times of programs," Department of Computer Science and Engineering, University of Washington, USA, Tech. Rep. 89-09-02, 1989.

[5] J. Gustafsson, A. Ermedahl, C. Sandberg, and B. Lisper, "Automatic derivation of loop bounds and infeasible paths for WCET analysis using abstract execution," in *RTSS '06: Proceedings of the 27th IEEE International Real-Time Systems Symposium*. Washington, DC, USA: IEEE Computer Society, 2006, pp. 57–66.

[6] M. de Michiel, A. Bonenfant, H. Casse, and P. Sainrat, "Static loop bound analysis of C programs based on flow analysis and abstract interpretation," in *14th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA '08)*, 2008, pp. 161–166.

[7] M. Bartlett, I. Bate, and D. Kazakov, "Guaranteed loop bound identification from program traces for WCET," in *Proceedings of the 15th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS 2009)*, San Francisco, CA, United States, April 2009, pp. 287–294.

[8] R. Stansifer, "Presburger's article on integer arithmetic: Remarks and translation," *Technical Report TR84-639, Department of Computer Science, Cornell University*, 1984.

[9] E. Vivancos, C. Healy, F. Mueller, and D. Whalley, "Parametric timing analysis," in *Proceedings of the ACM SIGPLAN Workshop on Languages, Compilers and Tools for Embedded Systems (LCTES'01)*, J. Fenwick and C. Norris, Eds., Snowbird, Utah, 2001, pp. 88–93.

[10] S. Bygde, A. Ermedahl, and B. Lisper, "An efficient algorithm for parametric WCET calculation," in *Proceedings of the 16th International Conference on Real-Time Computing Systems and Applications (RTCSA'09)*, P. Kellenberger, Ed. Beijing, China: IEEE Computer Society, 2009, pp. 13–21.

[11] E. J. Weyuker, "Axiomatizing software test data adequacy," *IEEE Transactions on Software Engineering*, vol. 12, no. 12, pp. 1128–1138, 1986.

[12] J. Wegener, H. Sthamer, B. F. Jones, and D. E. Eyres, "Testing real-time systems using genetic algorithms," *Software Quality Journal*, vol. 6, pp. 127–135, 1997.

[13] R. Kirner, I. Wenzel, B. Rieder, and P. Puschner, "Using measurements as a complement to static worst-case execution time analysis," in *Intelligent Systems at the Service of Mankind*. UBooks Verlag, 2005, vol. 2.

[14] P. Puschner and C. Koza, "Calculating the maximum execution time of real-time programs," *Real-Time Systems*, vol. 1, no. 2, pp. 159–176, 1989.

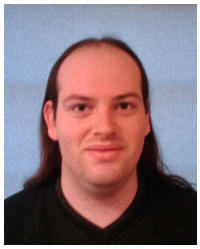
[15] C. Y. Park, "Predicting program execution times by analyzing static and dynamic program paths," *Real-Time Systems*, vol. 5, no. 1, pp. 31–62, 1993.

[16] A. Ermedahl and J. Gustafsson, "Deriving annotations for tight calculation of execution time," in *Euro-Par '97: Proceedings of the Third International Euro-Par Conference on Parallel Processing*. London, UK: Springer-Verlag, 1997, pp. 1298–1307.

[17] C. Healy, V. Rustagi, D. Whalley, and R. Van Engelen, "Supporting timing analysis by automatic bounding of loop iterations," *Real-Time Systems*, vol. 18, pp. 121–148, 2000.

[18] D. Sehlberg, A. Ermedahl, J. Gustafsson, B. Lisper, and S. Wiegratz, "Static WCET analysis of real-time task-oriented code in vehicle control systems," in *ISOLA '06: Proceedings of the Second International Symposium on Leveraging Applications of Formal Methods, Verification and Validation*. Washington, DC, USA: IEEE Computer Society, 2006, pp. 212–219.

[19] R. Wilhelm, D. Grund, J. Reineke, M. Schlickling, M. Pister, and C. Ferdinand, "Memory hierarchies, pipelines, and buses for future architectures in time-critical embedded systems," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 28, pp. 966–978, 2009.
[20] X. Li, A. Roychoudhury, and T. Mitra, "Modeling out-of-order processors for software timing analysis," in *Proceedings of the 25th IEEE International Real-Time Systems Symposium*, 2004, pp. 92–103.
[21] D. Kazakov and I. Bate, "New directions in worst-case execution time analysis," in *Proceeding of the 2008 IEEE World Congress on Computational Intelligence*, 2008.
[22] B. Lisper and M. Santos, "Model identification for WCET analysis," in *Proceedings of the 15th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS 2009)*. IEEE, April 2009.
[23] A. Ermedahl, C. Sandberg, J. Gustafsson, S. Bygde, and B. Lisper, "Loop bound analysis based on a combination of program slicing, abstract interpretation, and invariant analysis," in *7th International Workshop on Worst Case Execution Time (WCET) Analysis*, C. Rochange, Ed. Dagstuhl, Germany: Internationales Begegnungs und Forschungszentrum für Informatik (IBFI), Schloss Dagstuhl, Germany, 2007.
[24] N. Wilde and D. Knudson, "Understanding embedded software through instrumentation: Preliminary results from a survey of techniques," Technical Report, Department of Computer Science, University of Florida, 1999.
[25] E. Kligerman and A. D. Stoyenko, "Real-time Euclid: A language for reliable real-time systems," *IEEE Transactions on Software Engineering*, vol. 12, no. 9, pp. 941–949, 1986.
[26] N. Tawbi, "Estimation of nested loops execution time by integer arithmetic in convex polyhedra," in *Proceedings of the 8th International Symposium on Parallel Processing*. Washington, DC, USA: IEEE Computer Society, 1994, pp. 217–221.
[27] W. Pugh, "Counting solutions to Presburger formulas: How and why," in *SIGPLAN Conference on Programming Language Design and Implementation*, 1994, pp. 121–134.
[28] P. Clauss, "Counting solutions to linear and nonlinear constraints through Ehrhart polynomials: Applications to analyze and transform scientific programs," in *International Conference on Supercomputing*, 1996, pp. 278–285.
[29] D. E. Knuth, "Johann Faulhaber and sums of powers," *Mathematics of Computation*, vol. 61, no. 203, pp. 277–294, 1993.
[30] S. Muggleton, "Learning from positive data," *Inductive Logic Programming: 6th International Workshop, ILP-96, Stockholm, Sweden, August 26-28, 1996, Selected Papers*, 1997.
[31] A. Srinivasan, "The Aleph manual," *Computing Laboratory, Oxford University*, 2000.
[32] M. Bartlett, I. Bate, and D. Kazakov, "Challenges in relational learning for real-time systems applications," in *Proceedings of the 18th International Conference on Inductive Logic Programming*, ser. Lecture Notes in Computer Science, vol. 5194. Springer, 2008, pp. 42–58.
[33] D. Kazakov and I. Bate, "Towards new methods for developing real-time systems: Automatically deriving loop bounds using machine learning," in *Proceedings of the 11th IEEE International Conference on Emerging Technologies and Factory Automation (ETFA)*, 2006.
[34] A. Betts and G. Bernat, "Tree-based WCET analysis on instrumentation point graphs," in *ISORC '06: Proceedings of the Ninth IEEE International Symposium on Object and Component-Oriented Real-Time Distributed Computing*. Washington, DC, USA: IEEE Computer Society, 2006, pp. 558–565.



Mark Bartlett received his PhD. from the Department of Computer Science at the University of York in 2006. He is currently a postdoctoral research associate jointly in the Real-Time Systems and Artificial Intelligence Groups at the University of York. His recent research focuses on the application of Machine Learning techniques to Real-Time Systems problems.

Iain Bate is a lecturer in Real-Time Systems. His research interests include scheduling and timing analysis, design and analysis of safety-critical systems, and engineering of complex systems of systems including sensornets. He is the Editor-in-Chief of the Journal of Systems Architecture and a frequent member of programme committees for distinguished international conferences.

Dimitar Kazakov received his first degree in Technical Cybernetics from the Czech Technical University of Prague in 1993, followed by a PhD in Biocybernetics and Artificial Intelligence from the same university in 2000. He is a Senior Lecturer at the Department of Computer Science at the University of York, UK, where he has been since 1998. His research focusses on the applications of machine learning and multi-agent systems in areas as varied as computational linguistics, language evolution, real time systems and systems of systems. He has been a committee member of the UK Society for the Study of Artificial Intelligence and Simulation of Behaviour (SSAISB) since 2004.