Guaranteed Loop Bound Identification from Program Traces for WCET

Mark Bartlett, Iain Bate and Dimitar Kazakov
Department of Computer Science
University of York
Heslington, York, UK
Email: {mark.bartlett,iain.bate,dimitar.kazakov}@cs.york.ac.uk

***Abstract*—Static analysis can be used to determine safe estimates of Worst Case Execution Time. However, overestimation of the number of loop iterations, particularly in nested loops, can result in substantial pessimism in the overall estimate. This paper presents a method of determining exact parametric values of the number of loop iterations for a particular class of arbitrarily deeply nested loops. It is proven that values are guaranteed to be correct using information obtainable from a finite and quantifiable number of program traces. Using the results of this proof, a tool is constructed and its scalability assessed.**

I. Introduction

In many embedded systems, the real-time properties of the artefact are crucial. In particular, for Safety Critical systems, failure of tasks to meet their deadlines can be catastrophic. For this reason, it is imperative that estimates of the Worst Case Execution Times (WCET) of the tasks underlying the system are safe. At present, the only method for producing a guaranteed safe estimate of the Worst Case Execution Time of a piece of code is static analysis [1].

However, the safety of the estimate produced by static analysis comes at a price. Due to the halting problem [2], in general static analysis will produce pessimistic estimates of WCET.

Despite being one of the first identified challenges in the area of WCET [3], the issue of determining the number of iterations of a loop has remained largely unsolved by static analysis techniques [4]. Loose estimates can significantly increase the pessimism in the overall estimate of WCET. This effect is compounded when interactions between levels of nested loops are overlooked. Chapman [5] conducted a case study comparing the WCET calculated before and after adding manual flow annotations to non-rectangular loops. He observed improvements of as much as 64.5% in the estimates and around half that on average. Clearly this source of pessimism is far from negligible.

This paper presents a novel approach to solving the loop iteration problem. Rather than derive the number of loop iterations directly from the code, this is instead inferred from example program traces. This can then be used to annotate the program under analysis or be fed directly into a static analyser to produce tighter estimates of WCET. Furthermore, as the inferred number of iterations is a function of program variables, this can be utilised by parametric WCET techniques [6].

It is shown that this technique can be used to guarantee that the estimate of loop iterations is exact (and therefore safe) assuming that the code is restricted to a particular case of nested loops, those controlled by Presburger expressions [7]. The technique can deal with the case of non-rectangular loops of this form, something that until now has usually been manually annotated. In fact, it is mathematically proven that the number of iterations of loops of this type can be exactly established from a quantified number of observations of program behaviour, even in the absence of any information about the Presburger expressions controlling the loop. It is unnecessary that these observations are either exhaustive or include the worst case.

The remainder of this paper proceeds as follows. Firstly, the background to the work and related research are surveyed in Section II. Following this, in Section III, it is mathematically proven that the number of loop iterations can be precisely inferred from a specific number of observations of loop behaviour. Section IV then builds on this result to present a learner capable of correctly establishing the formula for the number of loop iterations from the theoretical minimum number of observations. Results concerning the scalability of the approach are presented in Section V, before concluding in Section VI.

II. Background and Related Work

A. Flow Analysis

There are currently two major approaches to estimating WCET [1]. Measurement-based approaches involve running code on the target hardware using a test suite and estimating WCET based on the observed run times. For any reasonably complex code, it will be impossible to exhaustively test code this way, hence in general this method produces optimistic WCET estimates which cannot be guaranteed as safe. In contrast, for systems in which the real-time constraints are hard, safe estimates of WCET must be obtained. Hence, in these situations, the alternative approach of static analysis is utilised.

Static analysis first conducts flow analysis, which builds a model of the code under examination. At this stage, the structure of the code is determined and information about program flow, including the number of loop iterations, will be estimated. This model is combined with a previously

constructed model of the hardware to produce a system of equations which can then be solved to give a safe estimate of WCET. Due to cautious assumptions and generalisations at each stage of this process, the estimate can be guaranteed to be safe. However, this will also inevitably introduce pessimism in the general case, as WCET estimation is undecidable [8].

In order to conduct the flow analysis, one of two major methods are used: symbolic execution [9] or abstract interpretation [10]. Symbolic execution involves 'running' a program abstractly on symbolic variables. Such execution will involve the incremental construction of a constraint problem relating variable values to each other after the execution of each instruction. Combined with the timing information of the hardware, this yields a constraint program that can be solved to bound the WCET. In contrast, abstract interpretation builds model in which execution occurs on variables which take multiple values. This simulates the execution on all possible inputs simultaneously. In order that this is both decidable and safe, cautious calculations on the result of executing each instruction are taken, typically limiting variables to taking a contiguous range of values.

B. Loop Iteration Determination

One potential source of great pessimism in static analysis is in the determination of number of loop iterations that can occur. Overestimating such quantities is liable to add significantly to the estimated WCET. Despite being identified as an early source of problems for WCET analysis [3], this issue remains unsolved.

The earliest approaches to this issue relied on programmers supplying this information manually though annotations to the code [8]. However, such a scheme is dependent on programmers correctly annotating their code; annotations could be mistyped or could become inconsistent with the code as revisions are made.

One solution to this is to check that the supplied annotations are correct through automated theorem proving [3], [5]. While this overcomes those problems, other issues still remain. Program annotations require work to create and maintain by programmers, at a cost of time and hence money. Additionally, for externally developed libraries, code may not have been annotated. Finally, for modern optimising compilers, annotations in the source code may not reflect the program structure in the object code, the most obvious such transformation being loop unrolling. For these reasons, automated inference of loop iterations, and flow information generally, should be preferred [10].

The aiT tool[1] takes a small step towards the automation of derivation of loop iteration bounds. Rather than manually annotate the code, patterns are manually created which correspond to the outline of code produced by particular compilers for given ways of programming of loops. While examining code, the tool then searches for these patterns and can automatically produce the annotations based on those associated with the pattern. However, while this frees programmers from having to annotate their code, the patterns must be produced by hand and are specific to individual compilers at different optimisation levels. This clearly presents problems in extending the approach to a more general level. Furthermore, more complex loops cannot be handled [11].

Current work on automated loop iteration bounding has been based on data-flow analysis [11], [12], [13], [14]. At a conceptual level, this involves identifying the effects of executing the loop and searching, explicitly or implicitly, for loop counters through abstract interpretation. The way these counters change over each iteration can then be used to bound the number of executions.

C. Instrumentation of Code

Alongside the matter of choosing an appropriate test suite, the issue of monitoring execution is one of the most important challenges in measurement-based approaches to WCET estimation. The information gained from such monitoring can then be used to reconstruct the path taken through the code. This can be useful for ensuring coverage of paths [15].

Two methods exist to obtain this information: simulation or instrumentation. With simulation, a software model of the relevant hardware is constructed and the code under examination is then run on this. Data about paths taken and time to execute particular blocks is freely and easily available from such a model. However, constructing a model in this way is extremely time consuming for any modern processor (although existing models of a few processors are available in systems such as SimpleScalar[2] and M5[3]). Furthermore, it is infeasible to model complex state of the art processors, for which design details are kept commercially confidential.

In order to overcome these problems, code can instead be run on the target hardware with instrumentation providing outputs that can be observed. The least intrusive form of this is to add monitoring to the inputs and outputs of chips, such as the buses. However, for a modern processor with features such as out of order execution and on-chip caches, information obtained in this way may not be sufficient to deduce the path followed through the code [16]. A variant on this theme is to use debugging hardware built in to the chip by the manufacturer, but these systems may be expensive or unavailable for the target hardware.

A more intrusive method is to introduce monitoring software in to the code under examination [17]. This software causes externally observable affects when encountered, such as toggling values on the chip's output or writing values to memory. As this alters the actual code executed, it is possible that the observed execution times differ substantially from those that would be seen in its absence. One way to prevent this being a problem is to instrument the code as lightly as possible and then retain the instrumentation in the final version of the system.

[1]Website: `http://www.absint.com/ait/`

[2]Website: `http://www.simplescalar.com`

[3]Website: `http://www.m5sim.org`

for l_1 = $pe_{1a}(V)$ **to** $pe_{1b}(V)$
  for l_2 = $pe_{2a}(V \cup \{l_1\})$ **to** $pe_{2b}(V \cup \{l_1\})$
    $\vdots$
    for l_n = $pe_{na}(V \cup \{l_i | i \in \mathbb{Z}^+, i < n\})$ **to**
        $pe_{nb}(V \cup \{l_i | i \in \mathbb{Z}^+, i < n\})$
      Innermost Loop Body
    next
    $\vdots$
  next
next

Algorithm 1: Generalised nesting of Presburger loops

D. Presburger Expressions and Loops

Presburger expressions are a restricted class of arithmetic expressions over a set of variables [7].

Definition 1. *The set of Presburger expressions of a set of variables X, $PE(X)$, can be defined as follows*[4]

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

where $X = \{x_1, x_2, \ldots x_n\}$ and $\forall j \in [0, n], \alpha_j \in \mathbb{Q}$

In other words, a Presburger expression over a set of variables is a first degree polynomial 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, arithmetic using members of the class is 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 [8], [18]. This paper is concerned with such a use of Presburger expressions for controlling the number of loop iterations.

Algorithm 1 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 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, neither may a loop counter's value be changed except by the loop's control expression. We also restrict our treatment to the case where the lower bound is less than or equal to the upper bound.

III. Mathematical Derivation 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 III-A). The number of terms in this polynomial will then be demonstrated (Section III-B). 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 III-C). 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.

A. 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: From Faulhaber's Formula [19], it follows that

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

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 Algorithm 1, 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.

[4]Throughout this paper, the notation $pe_i(X)$ will be used to denote an arbitrary member of $PE(X)$.

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 $\forall n \in \mathbb{Z}^+$. ■

B. 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 $\binom{n+|V|}{|V|}$. ■

C. 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. This can be done in any of the ways that are currently used to record values and flow in dynamic program analysis, such as instrumentation of the code or hardware probing. The various methods and the trade-offs that exist between altering the behaviour of code and simplicity of collection are well known [17].

Substituting this information 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 a maximum of $\binom{n+|V|}{|V|}$ 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.[5]

IV. Inferring Loop Iterations from Program Traces

We have implemented a tool to infer loop iterations from program trace data using the method described in the previous section.

This tool is based an Inductive Logic Programming (ILP) [20], 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

[5]Subject to the restriction that the equations to be solved by the Gaussian Elimination algorithm corresponding to two example executions are not collinear.

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

Algorithm 2: The loop iteration formula inferring algorithm

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 when 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 [21] using Progol syntax. The present version of the tool is an evolution of that which was described in detail in [22] (and first presented in an early form in [23]). The current version of the tool 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 theory 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 are referred to [22]. An outline of the algorithm behind the tool is shown in Algorithm 2.

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`.[6] 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.

[6]These values are found through the addition of counters and print statements to the high-level code. Such instrumentation may affect the flow of the low-level code generated, but is sufficient here to accurately illustrate the principles.

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.

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 III) 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 in to 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 V-B.

V. Scalability of the Approach

A. Number of Traces Needed

In Section III it was proven that, for the class of loops considered, the exact number of loop iterations could be precisely determined from observation of $\binom{n+|V|}{|V|}$ examples of the loop's behaviour. This relationship is displayed graphically in Fig. 1. The numbers of observations increases near exponentially in both the number of variables to be considered and the depth of loop nesting.

While the magnitude of these values becomes very large, it should be noted that real code seldom contains nested loops

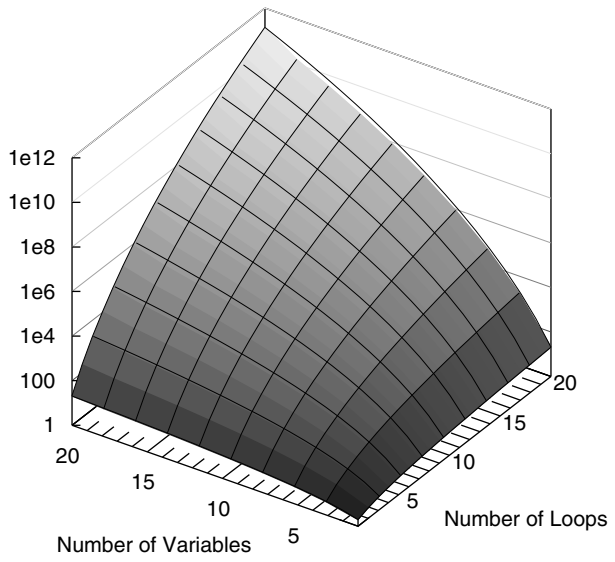

Fig. 1. Sufficient observations to infer the number of loop iterations for a given number of variables and depth of loop.

of such depth [5]. For practical purposes, a depth of around 8 loops is likely to form the limit necessary to consider and loops beyond a depth of around 4 are unlikely to be frequently encountered. At these depths, less than 10,000 observations are sufficient for a situation where 7 variables are to be considered (at a depth of 8 loops) or 19 variables (at a depth of 4). Furthermore, the number of variables to consider can also be generally reduced through techniques such as program slicing [24].

These two factors taken in conjunction mean that while the number of traces needed grows rapidly with increased depth or number of variables, the technique remains practicable in conditions likely to be observed in real code.

B. Time For Inference

In order assess the effectiveness of the tool described in Section IV, tests have been conducted using a selection of data corresponding to varying numbers of variables and levels of loop nesting.

This data is obtained from an automated test harness, which takes as input a number of variables and a depth of nesting. From this, a random program with the given characteristics is generated and executed multiple times with different input values for the variables. Output consists of a file of `target(A,B)` predicates (as described in Section IV) corresponding to the execution of the innermost loop body. Sufficient predicates are generated to allow the iteration count to be uniquely determined as described in Section III.

This test harness is used in preference to data obtained from real programs. We lack access to the type of industrial code for which WCET is usually performed. Even if such code was available, it would require analysis to locate suitable loops and would require instrumenting to obtain traces containing variable values and the iteration counts. While techniques to do this are well established, the amount of implementation work required would be immense and would draw attention to the specific approaches adopted rather than the general applicability of the method.

Likewise, it is preferable to use the test harness rather than established benchmarks, such as the Mälardalen benchmark suite[7]. Only 15 of the Mälardalen benchmarks contain nested loops. Of these, several contain only loops in which inner loop bounds are not a function of outer loop counters; these loops are already adequately handled by existing techniques. In short, there are insufficient examples in such suites to sufficiently demonstrate the benefits of this approach.

Most importantly, by generating our own code, we are able to manipulate the number of loops and variables as we wish. This enables a systematic testing of the tools performance as the number of loops or variables is altered. If testing were tied to real-world or benchmark code, then such examination would be piecemeal, according to the particular loops that the examined code contained.

The tool has been tested on a range of loops varying in the number of exogenous variables to consider and in the depth of the innermost loop body. Such testing has been conducted on a loop nest for each possible combination of depth and variable count from 1 to 4 for each. Such ranges should cover the vast majority of loops encountered in real code. In theory, we expect our analysis technique to continue to work for greater numbers of loops and variables; this range was chosen for exploration due solely to the convenience of generating tests for such loop nests. 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 function of all exogenous variables and all outer loop counters.

Times taken by the tool to learn the formulae for the number of iterations are shown in Fig. 2. All times have been collected from a PC with an Intel T2300 1.66GHz processor with 1GB of RAM running Microsoft Windows Vista.

The graph shows two clear trends. Firstly, the time taken appears independent of the depth of the loop, at least up to the limit explored here. Secondly, the time taken for learning increases exponentially as the number of variables to be considered increases.

The fact that the time required does not differ significantly as the loop depth is increased may be in part an artefact of the tool. The tool searches for polynomials up to a given degree and then selects the simplest consistent one found. Should the tool instead terminate its search as soon as it finds a suitable polynomial, then one would expect time to increase as depth increases. Additionally, if the tool could be supplied with the actual level of loop nesting as an input, then inference time

[7]Website: `http://www.mrtc.mdh.se/projects/wcet/`

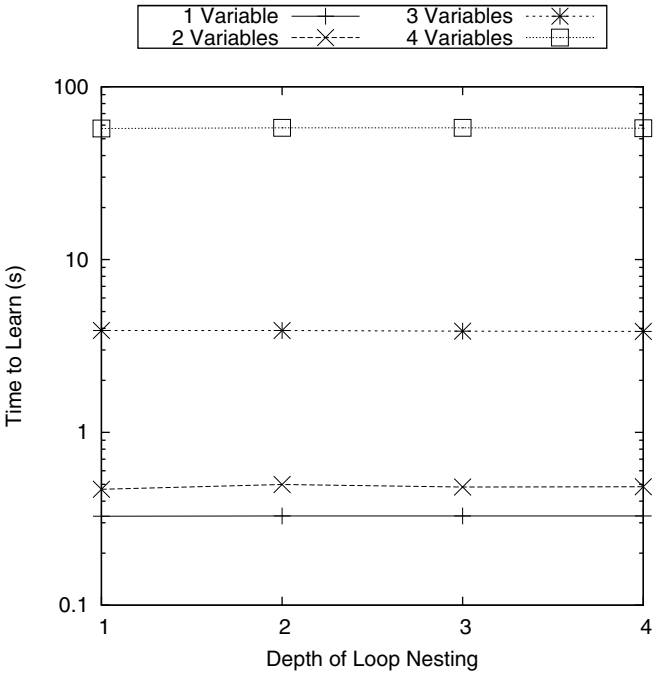

Fig. 2. Time taken to infer loop iteration formula for given number of variables and depth of loop.

could be cut significantly by only searching for a polynomial of the relevant degree.

The significant increases in time with increasing numbers of variables on the other hand reflects an underlying increase in the complexity of the task as more variables must be considered. Additional variables mean increases in the size of the system of equations that must be solved, and also in the number in terms in each of these equations.

The increases in time seen as the number of variables increases in Fig. 2 mirrors the increases in the number of examples needed in Fig. 1. This fits seems intuitive; the number of examples needed and the time taken for learning both reflect the complexity of determining the underlying function. In contrast, the increases in the number of examples needed as the number of loops increases does not have a corresponding increase in the time taken. As previously stated, this is due to the tool, which performs a similar amount of work regardless of the actual number of loops.

As the test loops were created to ensure that the formulae for the number of iterations were maximally complex, the reported values form an upper bound on the time needed to learn general loop formulae for the given situations. Nevertheless, the times observed are more than acceptable for practical applications of the technique. For 4 variables under consideration, the observed learning time was around 60 seconds. For 3 variables, this falls to under 5 seconds. Both of these values are practical for a field in which analysis times are often measured in hours. In a practical setting, time to identify the loops and variables under consideration and to generate program traces would need to be taken into account too[8]. However this is unlikely to be particularly time-consuming, and in the case of identifying the loops to examine and the variables to consider is likely to already be part of the static analysis, being necessary for control flow graph construction and program slicing respectively.

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 III 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. The generated code is simply a shell of loops in to 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.

VI. Conclusions and Future Work

This paper has presented a novel method for the automated derivation of the number of iterations of a loop. This has relied on the observation that the number of executions of a potentially nested, Presburger controlled loop can written as a polynomial of a given order. Specifically, it has been proven that for n well structured nested loops whose bounds can be expressed as Presburger expressions, the number of loop iterations is a degree n polynomial. Furthermore, from $\binom{n+V}{V}$ program traces, it is possible to derive the exact formula for the number of loop iterations as a parametric formula of V exogenous variables. For any realistically complex loop nest, obtaining this quantity of program traces should be feasible.

A tool using the technique to calculate loop iterations was then presented. This tool is constructed in a machine learning environment and uses inductive logic programming to acquire the appropriate formulae in a Prolog representation. This tool was tested on a suite of synthetically generated examples. Practical experience bore out the theoretically derived result, with all formulae being identifiable from the prescribed number of example traces.

The time taken by the tool to learn the formulae was unaffected by the depth of loop nesting involved. In contrast, as

[8]The figures reported here refer solely to the time needed to read in the data, generate potential formulae and then select a consistent one from them.

the number of potential salient variables to consider increased, the time to learn increased at an exponential rate. Observed times to learn formulae were of the order of 4 seconds for 3 variables and a minute for 4 variables; these times are viable for practical applications. However, the exponential increase in time with additional variables highlights the need for program slicing [24] as a technique to focus on a low number of variables for consideration.

As the formulae learned for the number of loop iterations are safe under the stated assumptions (and are in fact exact), they can be incorporated in to a static analysis and preserve the safety of that approach. In the future, we aim to perform such integration with an existing static analyser, acquiring the segments of the program to trace from the control flow graph and then writing back the learned formulae as flow facts. It may also be possible to safely include formulae derived from this method, even if the necessary assumptions cannot be guaranteed. One early suggestion involving manual loop iteration annotations was that they could be checked by an automated theorem prover [3]; a similar method could be used here, with a candidate formula being learned and then checked to see whether it is valid.

Looking beyond the confines of the loop iteration task studied in the current paper, this technique has the potential to be used in many other aspects of WCET analysis. The technique underlying the method is to infer a model of the behaviour of a certain aspect of the code or hardware based on execution behaviour. This contrasts with static analysis, which builds models based on structural analysis of the code or hardware, and also with measurement-based approaches, which use the execution behaviour directly without creating models. Already, in addition to its use here, the technique of model inference from program traces has been demonstrated to be of use in deducing aspects of the hardware from observations of behaviour [25]. Doubtlessly, there are many other part of the WCET estimation task where this could also be useful.

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] A. M. Turing, "On computable numbers, with an application to the entscheidungsproblem," *Proceedings of the London Mathematical Society*, vol. 42, no. 2, pp. 230–265, 1936.

[3] 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.

[4] J. Gustafsson, A. Ermedahl, and B. Lisper, "Towards a flow analysis for embedded system C programs," in *WORDS '05: Proceedings of the 10th IEEE International Workshop on Object-Oriented Real-Time Dependable Systems*. Washington, DC, USA: IEEE Computer Society, 2005, pp. 287–300.

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

[6] B. Lisper, "Fully automatic, parametric worst-case execution time analysis," in *Workshop on Worst-Case Execution Time (WCET) Analysis*, 2003, pp. 77–90.

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

[8] 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.

[9] P. Altenbernd, "On the false path problem in hard real-time programs," in *Proceedings of the Eighth Euromicro Workshop on Real-Time Systems*, 1996, pp. 102–107.

[10] 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.

[11] 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.

[12] 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.

[13] 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.

[14] 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.

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

[16] S. M. Petters, "Comparison of trace generation methods for measurement based WCET analysis," in *3nd International Workshop on Worst Case Execution Time Analysis*, 2003, pp. 61–64.

[17] N. Wilde and D. Knudson, "Understanding embedded software through instrumentation: Preliminary reults from a survey of techniques," Technical Report, Department of Computer Science, University of Florida, 1999.

[18] 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.

[19] D. E. Knuth, "Johann Faulhaber and sums of powers," *Mathematics of Computation*, vol. 61, no. 203, pp. 277–294, 1993.

[20] S. Muggleton, "Learning from positive data," *Inductive Logic Programming: 6th International Workshop, ILP-96, Stockholm, Sweden, August 26-28, 1996, Selected Papers*, 1997.

[21] A. Srinivasan, "The Aleph manual," *Computing Laboratory, Oxford University*, 2000.

[22] 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.

[23] 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.

[24] 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.

[25] 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.