Challenges in Relational Learning for Real-Time Systems Applications

Mark Bartlett[1,2], Iain Bate[2], and Dimitar Kazakov[1]

[1] Artificial Intelligence Group, Department of Computer Science, University of York, Heslington, York, UK.
[2] Real Time Systems Group, Department of Computer Science, University of York, Heslington, York, UK.
{mark.bartlett,iain.bate,dimitar.kazakov}@cs.york.ac.uk

Abstract. The problem of determining the Worse Case Execution Time (WCET) of a piece of code is a fundamental one in the Real Time Systems community. Existing methods either try to gain this information by analysis of the program code or by running extensive timing analyses. This paper presents a new approach to the problem based on using Machine Learning in the form of ILP to infer program properties based on sample executions of the code. Additionally, significant improvements in the range of functions learnable and the time taken for learning can be made by the application of more advanced ILP techniques.

Key words: Worst Case Execution Time (WCET), Inductive Logic Programming (ILP), Lazy Learning, Symmetry, Efficiency

1 Introduction

In the area of Real-Time Systems (RTS), the temporal behaviour of systems is of critical importance. In particular, a substantial amount of research effort goes into the issue of guaranteeing that code will be executed within a given time frame. While much scheduling theory exists on this matter, it is commonly assumed that the timing behaviour of the individual software components is known. One of the temporal properties of a process that schedulers require is that of the Worst Case Execution Time (WCET). As the name implies, this is the longest time that the process may require to run given the worst case input possible.

A fundamental problem exists however in determining the WCET of a program. The WCET of a program is in general undecidable, due to the well-known *halting problem* [1]. Therefore in the general case, any non-exhaustive attempt to determine this quantity will only ever be able to return an approximation. We propose that the use of Machine Learning may be a viable alternative to existing methods in the field, allowing for the accurate approximation of WCET in a competitive time.

One particular aspect of determining WCET that is examined here is the issue of deciding on the number of times that a loop is executed. Determining this

quantity with high accuracy has the ability to massively increase the accuracy for the WCET approximation as a whole. Again, this problem is in general undecidable, though there are well-defined classes subclasses of loop for which the problem is mathematically decidable. One of these classes, Presburger Loops, will be studied in this paper. For this class, our technique is able to produce an exact solution. Program flow analysis, and in particularly loop bound estimation, has been identified as an important source of overestimation, up to 30%, leading to a significant waste in resources.

There are several aspects of the WCET problem that makes Machine Learning, and ILP in particular, a good candidate for providing a solution. The data to learn from possesses many characteristics that are highly desirable in this realm: it is noiseless, discrete, deterministic and available in virtually limitless quantities [2]. These properties mean that many problems that occur in more complex domains should be avoided completely here.

Nevertheless, there are real challenges in using ILP for this task. These stem primarily from two sources. Firstly, basic ILP isn't particularly well-suited to the learning of numeric data and equation discovery. Secondly, the range of functions to be considered as hypotheses is vast. A significant part of this paper is devoted to showing how these limitations can be overcome in order to massively increase the speed with which hypotheses can be found and to allow the discovery of equations not possible with a naively coded learner.

The rest of this paper proceeds as follows. Firstly, the background to the WCET problem is presented in section 2 along with current approaches to its solution. The suitability of ILP and other related methods are then considered in section 3. Section 4 deals with the construction of a new ILP formulation to solve the a particular aspect of the problem. Firstly, a simple, previously published [2] implementation capable of learning the number of loop executions is presented. The problems that exist with this implementation are then highlighted. From this, a novel implmentation is then presented, through a series of refinements using more advanced ILP techniques, resulting in a learner capable of acquiring a larger class of equations and in a faster time. Results showing the improved ability of the more advanced learner to solve the problem are then given in section 5. The massive improvement in the behaviour of the more advanced learner is also shown. A related result is shown in section 6, in which machine learning was successfully applied to another aspect of WCET analysis, learning a branch predictor. Finally section 7 concludes and considers how ILP can be applied to other aspects of the WCET.

2 Worst-Case Execution Time Analysis

There are three current approaches to estimating WCET. Static analysis, measurement-based analysis and hybrid analysis. Static analysis examines the code and execution environment mathematically to build a model to reason about the program behaviour. In contrast measurement-based analysis comes to an estimate through running execution tests on the code and target platform

directly. Hybrid analysis combines the two previous approaches, building models based on the code and combining these with timing measurements found from actual executions. Each of these approaches has different strengths and weaknesses.

2.1 Static Analysis

Static analysis is based on automated reasoning about the execution of a program based on its code and the execution hardware. As the determination of WCET is in general undecidable, static analysis will not in general yield the correct WCET of a piece of code. Instead, the incompleteness of the reasoning process ensures that the estimated WCET is always at least as long as the actual WCET (*safe*), with the overestimate being termed the *pessimism*. This safety is important in hard real-time systems where deadlines must be guaranteed to be met and overruns could literally cost lives. However, pessimism is undesirable as it can lead to underutilised hardware resources or unnecessary investment in faster equipment.

Typically static analysis is done in three phases; flow analysis, low level analysis and calculation.In the first phase, the analyser reasons about the possible paths that may be taken through the code. It does this by building a *control flow graph* (CFG) showing how control passes between basic blocks of code, and then reasoning about this flow. There are several common techniques that may be utilised such as abstract interpretation [3] and symbolic execution [4], the former simulating the effect of code on a value range rather than individual input values and the latter building up constraints that replicate the logical flow of the program. In either case, the CFG will be annotated with derived flow facts, such as the maximum number of loop executions and infeasible paths. Due to the undecidability of the problem, these facts are incomplete.

Following this, low level analysis determines the time that would be taken to execute particular paths on the target platform. This analysis must account for advanced hardware features such as processor pipelines and branch predictors. As modern processors have become more advanced, the potential is for the degree of pessimism to rise [5]. For modern processors, technical details are often difficult to obtain due to commercial confidentially. Therefore, current analysis is generally restricted to parts of the processor for which the low level behaviour is well known and for which analysis is feasible. It appears that this problem will only get worse as processors become even more complex, such as Multi-Processor Systems on a Chip (MPSoC).

Finally, the calculation phase combines these two earlier phases to reveal the WCET estimate of the code. There are three main methods used for this; structure-based [6], path-based [7] and implicit path enumeration (IPET) [8]. Each integrates the flow and timing information, but using different techniques.

2.2 Measurement-Based Analysis

The other existing approach to estimating WCET is measurement based analysis. Here, the code is analysed through repeatedly running it on different inputs in an attempt to manually find and exercise the worse case path.

There are three methods that may be used to do this. Firstly, it is possible to manual generate test data based on human reasoning about the inputs most likely to be time-consuming to process. This can be effective if the tester has good knowledge of the code and problem domain, but this is often not the case. Secondly, a coverage metric can be employed, which defines a set of necessary execution conditions that must be performed by the test suite [9]. For example, in order to achieve branch coverage, it is necessary that both the true and false branch of each conditional are exercised by tests. The coverage method of testing enables systematic examination of the software, but the source code must be available; this is not always possible when library functions are called. Finally, testing can be automated through the use of genetic algorithms [10]. The input used for a test is the chromosome and familiar cross-over and mutation techniques are used to explore the space of all inputs. Fitness can either be scored based directly on execution time, or based on factors within the code such as the number of times a loop executes or whether particular conditional branches were taken.

For any non-exhaustive test data set, however generated, there always remains the possibility that the WCET has not been observed. This means that measurement-based approaches cannot be considered safe. In practice, this is overcome by allocating a margin of error to the worst observed time (for example 10%), though this still does not guarantee the safety of the approach. For any hard real-time system in which deadlines must be met, measurement-based analysis is unsuitable. However, there are many soft real-time applications where an occasional missed deadline can be tolerated. For example, in telecommunication applications, failure to decode a single frame of video in time to display may be acceptable providing such failures are infrequent.

2.3 Hybrid Analysis

Hybrid analysis works like static analysis, but circumvents the problem of unhandleably complex hardware by using actual program executions for the low-level phase. The flow analysis and calculation phases are typically performed in a similar manner to that used in static analysis, with the low-level data coming directly from real executions of the basic blocks.

This approach removes the excessive pessimism associated with low-level analysis on highly complex or poorly understood hardware, but also results in safety no longer being guaranteed. While the testing guarantees the accuracy of timing behaviour for individual blocks, safety may be lost through interactions over longer ranges, such as through pipelines or alterations in the cache contents.

Nevertheless, hybrid analysis is becoming used in domains where occasional deadline misses can be tolerated. The combination of rigorous static code analysis

and measurement-based hardware analysis combines some of the strengths of both approaches.

3 Suitability of ILP and Related Methods

3.1 Suitability of ILP

Given the problems in the existing methods for WCET, there is potential for a alternative approach. A technique based on Machine Learning may fit the criteria for this approach. Using Machine Learning, execution traces of programs can be used to infer a mathematical model which is able to reproduce the observed data. This can then be used to provide information to be incorporated into a static analysis. For example, in the case of determining loop bounds, which is examined later, traces of the number of loop executions observed in practice can be used to construct a parametric model which enables the number of loop bounds to be predicted for any given input. This can then be incorporated into a static analysis at the flow analysis stage as a flow fact. Obtaining data through observation rather than through analysis of the code means the results may no longer be safe; this problem may be addressable through a suitable choice of which, and how many, execution traces to learn from.

The approach is somewhat akin to hybrid analysis, though rather than incorporate observed data directly into the analysis, it is used to build a model which can then be included. Additionally, whereas hybrid analysis uses only observed behaviour for the low-level stage, our approach allows models based on observed data to be included at the flow analysis stage as well. Finally, this method allows existing static analysis to be retained and complements it with additional information; in hybrid analysis, reasoning about the low-level behaviour is discarded and replaced in its entirety by execution data.

There are several aspects of the problem which make ILP an ideal Machine Learning technique for the task at hand. The data from which the model will be learned is noiseless; this removes one of the problems that frequently makes ILP difficult to apply to a domain. The variables likely to be encountered are discrete, and often either intervals or categories. This immediately suggests the use of a first-order logical representation for the data and theories such as is seen in ILP. The examples to be learned from will only be positive; one will be unlikely to observe, for example, how many times a loop is *not* executed. Off the shelf ILP tools such as Progol [11] and Aleph [12] have the capability to learn from positive only data built in.

3.2 Dynamic Invariant Detection

Learning the number of loop executions as a function of program variables is highly analogous to the process of dynamic invariant detection. Invariants are relationships between program variables that must always hold true at a certain point in a program. Detecting invariants statically is closely related to the

static analysis techniques already described: invariants are found by reasoning mathematically about the program. In contrast, the dynamic technique infers invariants based on observing which properties are always true over a set of program traces.

The most widely used dynamic invariant detector is Daikon [13], which works in four stages. Firstly, the program to be examined is instrumented to record all program variables at each procedure entry and exit point. The instrumented program is then run on a user specified suite of test examples. Next, the resulting traces are processed to establish invariants for particular instrumentation points, and finally these candidate invariants are filtered to remove implied invariants and those likely to be due to chance observations.

Daikon detects a wide range of relationships that can hold between variables (including unary, binary and trinary relationships) and for many different datatypes. However, Daikon in an unmodified form cannot be used to learn program flow information. Daikon learns constraints that apply at procedure entry and exit; for learning of program flow, knowledge is required of how many times the body of a loop is entered and which branch is taken at conditional statements.

While conceivably these problems could be overcome by adding additional variables to the program, such as counters to loops[3], the learning bias is hard-coded into Daikon and difficult to modify. For ILP systems, this can be easily modified through altering the background knowledge. Furthermore, it is unclear how well Daikon can handle complex formulae with multiple variables and several constants which must be abducted.

3.3 LAGRAMGE

Lagramge [14] is a non-linear numerical regression (aka equation discovery) tool closely related to the ILP family, in which the user provides a grammar to describe the range of possible equations that should be considered in the search for the best fit for the data with respect to a given optimality criterion, typically the least squares. Here the grammar provides a description of the way how the independent variables could be linked through the use of certain operators to build an equation that best models the data in a way similar to the use of background predicates in mainstream ILP. The tool offers a choice between ordinary and differential equations, and later versions allow for the simultaneous learning of several equations. There are two aspects that make Lagramge unsuitable for the task at hand: the variable range cannot be restricted to integers, and, even more importantly, the algorithm focuses on minimising a function of the error on the average, whereas here one is interested in outliers and extreme behaviour.

[3] Though doing so would modify the code which may affect the compiled code generated, especially in the presence of compiler optimisations.

4 Loop Bound Learning

4.1 Problem Domain

In order to understand why achieving tight loops bounds for WCET analysis is important, it is worth considering the example of the typical nested loop used in many sort routines.

```
for i = 0 to n
    for j = 0 to n − i − 1
        if a[j] < a[j+1] then
            ⋮
        end
    next
next
```

For many existing WCET tools, the maximum number of executions of the outer loop will be found to be n, and the maximum number of executions of the inner loop will be $n-1$. Traditional WCET techniques fail to determine the existence of a relationship between the number of loop executions and a counter, simply assuming that the worst possible value may occur in all interations. This results in an estimate of $n^2 - n$ executions of the loop body. In reality, the inner loop only executes $n-1$ times when i is 0 and fewer in all other circumstances.

The actual total number of executions of the inner loop body will be $\frac{n^2-n}{2}$, exactly half that obtained by a basic analysis of the loops. Consequentially, we would expect the estimated execution time to be immensely pessimistic and twice the actual execution time. This pessimism arises in the cases where the number of executions of the inner loop is variable and depends on the value of the outer loop counter. A relational formula linking the loop executions to variables in the program can accurately record this situation.

No existing WCET technique is able to automatically detect such dependencies in program flow. As will be shown in later sections, through the use of machine learning for equation discovery, it is possible to establish relationships such as these.

Loop bounds can be data dependent and in the general case can be expressed with arbitrary complexity. These reasons are key to why the problem is classed as undecidable in the general sense [15]. However it is possible to describe restricted classes of loops for which the loop bound is a mathematically decidable problem. Presburger loops are one such class.

Presburger expressions [16] can be written in the form

$$k + \sum_p a_p V_p \qquad k, a_p \in \mathbb{R}$$

where V_p are variables. The number of executions of a loop can be decided if the loop conditionals are Presburger expressions in which the variables are either

```
% tp(A,B) where B = A * (A - 1) / 2
  tp(1,0).   tp(2,1).   tp(3,3).   tp(4,6).
  tp(5,10).  tp(6,15).  tp(7,21).  tp(8,28).
  ...
```

Fig. 1. Sort routine data set

loop counters from an outer containing loop or program variables which do not have their value changed.

for $i = (\alpha_0 + \sum_p \alpha_p V_p)$ **to** $(\alpha'_0 + \sum_p \alpha'_p V_p)$
  for $j = (\beta_0 + \beta_i i + \sum_p \beta_p V_p)$ **to** $(\beta'_0 + \beta'_i i + \sum_p \beta'_p V_p)$
    for $k = (\gamma_0 + \gamma_i i + \gamma_j j + \sum_p \gamma_p V_p)$ **to** $(\gamma'_0 + \gamma'_i i + \gamma'_j j + \sum_p \gamma'_p V_p)$
      $\vdots$
    next
  next
next

In the remainder of this paper, we shall restrict the class of loops considered to Presburger loops. There are several reasons for this. Firstly, many nested loops in actual program code correspond to this class, making the results relevant for real world situations. Secondly, as the result is decidable for this class, it is possible to generate data and check the accuracy of learned results without any complications. Finally, for a restricted class such as this, it is possible to make improvements to the efficiency of the learning algorithm based on knowlegde of the target concepts.

4.2 A Naive Loop Bound Learner

The potential of using ILP to learn a WCET bound was first shown by Kazakov and Bate [2] on the case of learning nested loop bounds. It is necessary to give an overview of this work here, as subsequent sections go on to discuss the problems with this implementation and develop an improved learner for the same task.

The use of ILP can be demonstrated with a sort routine which illustrates the case of nested loops where the inner loop bounds are functionally dependent on the outer loop counter. The known equation for this routine was then used to generate pairs of numbers representing the upper bound n and the corresponding number of times of running the inner loop body (see Fig. 1). The variable range was set to $n \in \{1, \ldots, 30\}$.

Two background predicates, `sum/3` and `product/3`, were used which calculated the sum and product of two arguments respectively. Using this formulation and data set, Progol4.4 finds a one-rule model.

```
tp(A,B) :- product(C,A,A),
           sum(D,B,A),
           sum(C,B,D).
```

This translates to a system of three equations:

$$C = A \times A$$
$$D = A + B$$
$$C = B + D$$

Note that the division operator is not part of the background knowledge, nor is Progol allowed to use constants in its hypotheses. Nevertheless, the result is correct, albeit expressed in a somewhat unusual way. Indeed, the above equations can be reduced to:

$$B = \frac{A \times (A - 1)}{2}$$

This is, of course, the correct formula.

Using a similar formulation, it was also shown that simple loops and nested invariant loops can be learned [2]. These early experiments are mostly valuable as a proof of concept, but they already show the potential for empirically deriving accurate upper bound estimates with acceptable amounts of processing. Previously, automatic processing in the WCET domain had not been able to acquire parametric formulae for loop nesting where the inner loop counter depends on the outer.

4.3 Limitations of the Naive Loop Bound Learner

While the learner based on the two background predicates `sum/3` and `product/3` was able to automatically acquire a range of simple expressions for loop bounds, it is unable to acquire the full range of Presburger loops.

In fact, it is not necessary to even look at nested loops to find the first example for which the naive learner cannot find the correct formula. A loop as simple as

for $i = 0$ **to** $1000V_1 + 1000$
$\vdots$
next

is virtually unlearnable by the naive implementation. In order to find this formula, the learner would need to abduct 2 constants from the data. As there is no a priori reason to suppose that 1000 is a particularly interesting number, the learner must try all possible constant combinations in order to locate this particular example. Either the presented learner would fail to learn this formula at all (due to resource limitations), or would take an extraordinary time to find it. Therefore it is necessary to implement a more effective learner capable of learning the behaviour of Presburger loops in the general case.

4.4 An Improved Loop Bound Learner

While the naive approach is capable of learning a range of functions, including those for variously nested loops, it suffers from limitations. These are of two

types. Firstly, the range of functions learnable is actually quite limited in practice. Secondly, the time to learn a new function increases quite considerably as the function becomes more complex. The underlying reason for both of these is that the search space of expressible hypotheses is a very large superset of the potentially occurring functions. This results in some functions being very time-consuming for the learner to reach, and potentially beyond the resource limits of the computer. The remainder of the section develops an improved learner by outlining the exact sources of these limitations and describing ILP techniques that can be used to overcome them. Aleph [12] was used in preference to Progol4.4 (which was used for the naive learner) as it included all the features necessary for the new implementation.

Removing Impossible Hypotheses Given that the type of functions that should be learnable have been explicitly stated in the previous section, the first modification in building a better learner should be to restrict the expressible hypotheses to as close to this set as possible; there is no point in providing background knowledge which enables a hypothesis search space much larger than the known class of hypotheses.

It can be shown that the total number of executions, E, of a nest of Presburger loops of depth D is equal to a function of the form

$$E = \sum_{\beta_1=0}^{D} \sum_{\beta_2=0}^{D} \cdots \sum_{\beta_n=0}^{D} \left[\alpha_{\beta_1\beta_2\ldots\beta_n} \times \prod_{i=1}^{n} V_i^{\beta_i} \right]$$

where

$\alpha_{\beta_1\ldots\beta_n} \in \mathbb{N}$

All coefficients $\alpha_{\beta_1\ldots\beta_D} = 0$ if $(\sum_{x=1}^{n} \beta_x) > n$

$V_1 \ldots V_n$ are the variables

In other words, the function is sum of terms, where there is one constant term and the other terms are coefficients multiplied by various combinations of the variables. This can be a little difficult to comprehend in its generalized form, so consider the particular case for 3 levels of nesting and two variables.

$$\begin{aligned} E &= \sum_{\beta_1=0}^{3} \sum_{\beta_2=0}^{3} \left[\alpha_{\beta_1\beta_2} V_1^{\beta_1} V_2^{\beta_2} \right] \\ &= \alpha_{00} + \alpha_{10}V_1 + \alpha_{20}V_1^2 + \alpha_{30}V_1^3 + \alpha_{01}V_2 + \alpha_{11}V_1V_2 + \\ &\quad \alpha_{21}V_1^2V_2 + \alpha_{02}V_2^2 + \alpha_{12}V_1V_2^2 + \alpha_{03}V_2^3 \end{aligned}$$

In light of this, the background knowledge for the improved learner is altered by removing the `sum/3` and `product/3` predicates and replacing them with more specialised predicates directed towards learning functions of this form. Two new families of predicates are introduced in their place. Firstly, there is a new set of new term generating predicates `make_term/X` which multiply $X - 1$ variables

```
make_term(Variable1,Variable2,Term) :-
   Term is Variable1 * Variable2.

make_term(Variable1,Variable2,Variable3,Term) :-
   Term is Variable1 * Variable2 * Variable3.

weighted_sum(Constant,Coefficient1,Term1,Output) :-
   Output is Constant + Coefficient1 * Term1.

weighted_sum(Constant,Coefficient1,Coefficient2,Term1,Term2,Output) :-
   Output is Constant + Coefficient1 * Term1 + Coefficient 2 * Term2.
```

Fig. 2. More Specialised Background Predicates

together to give a term. These terms are then taken as input to the second new set of predicates `weighted_sum/X` which produces the weighted sum of $\frac{X}{2} - 1$ terms. Examples of some `make_term/X` and `weighted_sum/X` predicates are given in fig. 2. Obviously, the more predicates of each of these types that are added, the greater the class of hypotheses that are representable in the language, but also the larger that the search space of hypotheses will be. For practical reasons, the predicates are limited here to `make_term/1`, `make_term/2`, `make_term/3` and `make_term/4`, and `weighted_sum/4`, `weighted_sum/6` and `weighted_sum/8`. These prove adequate to learn the functions necessary in this paper, and in principle they can be extended trivially to allow more functions, with greater levels of nesting or more variables, to be learned.

Lazy Learning Having altered the background knowledge to focus the search space on the valid function space, the issue of the unlearnability of some functions by the naive learner is now addressable.

The primary limitation on the functions that are learnable arises from the need to deal with numerical constants in the discovered equation. In the examples previously considered, these constants were either absent or small positive integers. Learning the relationship between a set of input variables and the number of loop executions requires the discovery of an equation expressing the loop executions as a function of the input variables. This presents problems for traditional ILP which is very poor at generating the numbers needed for such formulae. The problem occurs due to the construction of a bottom clause to guide the search.

The task of learning the equation requires the construction of a formula that is true for all the data. When the bottom clause is created, it will contain all possible formulae that are consistent with that datum. However with no limits on the possible formula, this will consist of an infinite number of formulae, the majority of which are inconsistent with any other data. Even in an extremely simplified situation in which there is only a single variable, V, and the formulae for the number of loop executions, E, are limited to the form $E = k + aV$, an

infinite number of formulae should be placed in the bottom clause, each with a unique (k, a) pair; again the vast majority would still be found to be inconsistent with all other data points.

One approach to counter this problem is to limit the numbers that can be considered to some given set, but this limits the hypothesis space and may still lead to a large bottom clause and hence an over-large search space. For these reasons, an alternative technique is adopted which retains the size of the hypothesis space, but greatly limits the search space within it.

Lazy learning in ILP was first proposed by Srinivasan and Camacho [17]. Using this technique, clauses featuring constants can be added to the bottom clause as usual, but the constants themselves are only determined later during the subsequent search though the hypothesis space. This enables all the data to be used to determine the constants instead of just the single datum used to generate the bottom clause. Returning to the example given earlier, the bottom clause could have a single $E = k + aV$ clause added to it, with the actual values of k and a being calculated at search time from all the data. Clearly, this reduces the size of the bottom clause and consequently the search space, but crucially not the hypotheses that can be returned. This also decreases the time spent searching for the correct solution.[4]

In essence, lazy learning is used to transform the process of learning from a search of the *function* space to a search of the *functional form* space.

Symmetry Removal One final improvement to the learner that can be made to reduce the size of the search space is the removal of symmetry. For example, if

```
target(A,B,C) :- make_term(A,B,D), weighted_sum(0,1,D,C).
```

fits the observed data exactly,[5] then so will

```
target(A,B,C) :- make_term(B,A,D), weighted_sum(0,1,D,C).
```

Ideally, this symmetry should be removed to reduce the space that must be searched. Symmetry also creates a problem in the construction of terms of the form $A^n B^m$; using `make_term/4`, there are 3 different ways to create A^2B based on permuting the order of As and Bs. The more arguments permitted to `make_term` and the larger the number of variables present, the greater this problem becomes.

The approach adopted to remove this symmetry is to prevent two clauses with the same meaning both being added to the bottom clause. This forces the learner to only consider one of the many cases when searching for a solution.

[4] Assuming that a more efficient method exists for finding the constants involved than a brute-force search through their possible values. In the case of fitting the functional forms mentioned in this paper, Gaussian Elimination [18] can used as this more efficient algorithm.

[5] i.e. C = AB

```
% Symmetry suppressing version for use in creating the bottom clause
make_term(v(X),v(Y),v(Z),t(A)):-
  setting(stage,saturation),
  X =< Y, Y =< Z,
  A is X * Y * Z.

% Symmetry allowing version for use in searching
make_term(v(X),v(Y),v(Z),t(A)):-
  \+ setting(stage,saturation),
  A is X * Y * Z.
```

Fig. 3. Symmetry suppressing and allowing clauses for `make_term/4`

Specifically, it is required that variable arguments to a `make_term/x` predicate are sorted in a non-decreasing order[6].

Using this approach, only `make_term(A,B,C)` can be added to the bottom clause if $A > B$ and only `make_term(B,A,C)` if the reverse is true. This immediately removes both sources of symmetry identified above. However, while this property may apply to A and B in the example used to create the bottom clause, it may not hold for other data in the set. Therefore, this requires splitting each background predicate into two: one is used when the bottom clause is constructed and the other used during the search for testing the coverage of a hypothesis. The first clause includes the ordering condition to suppress all but one of the symmetric cases. In contrast, the second allows all the symmetric cases to succeed. These pair of clauses are shown for `make_term/3` in fig. 3. Note that `setting(stage,saturation)` is an internal Aleph predicate that succeeds only when the bottom clause is under construction.

5 Results

Having identified weaknesses in a basic implementation of the loop bound learner and having suggested how they may be addressed, it is necessary to assess the extent to which the modified version overcomes the problems. In order to do this, implementations of both versions were coded for use in Aleph (the simple version being ported from an original implementation in Progol4.4 for fair comparison).

Three problems were chosen which serve to highlight the difference between the learners well. Firstly, a relatively simple example of three loops nested inside each other for which there was no interaction between the loop counters. Secondly, a nested loop structure featuring multiple constants without interaction between loop counters, and finally, an example of the sorting routine nested loops which has been mentioned extensively in this paper.

While benchmark suites do exist for WCET analysis, artificially generated problems are used in preference here. There are multiple reasons for this decision.

[6] Similar constraints apply to the term arguments of `weighted_sum/X`. However, for clarity, all the discussion will focus on `make_term/X`.

ABC	Bottom Clause	Learning Time (s)
Naive Learner	56	2.48
Reformulated Lazy Learner	1139	8.41
+ Symmetry Removal	118	0.70

100ABC+1	Bottom Clause	Learning Time (s)
Naive Learner	88	N A
Reformulated Lazy Learner	1139	8.49
+ Symmetry Removal	118	0.73

A(A-1)/2	Bottom Clause	Learning Time (s)
Naive Learner	99	1.26
Reformulated Lazy Learner	19	0.08
+ Symmetry Removal	11	0.03

Fig. 4. Number of literals in bottom clause and search time for different learners on various problems.

The benchmark suites typically used are made of very simple functions and are not representative of actual real-time code, which is itself unavailable due to commercial confidentiality. Those benchmarks that do exist feature few functions with interaction effects between counters of nested loops, and those which do are no more complex than sort algorithms. Finally, as no existing techniques are able to automatically infer these loop bound relationships for interacting loops, there is no data to compare our performance to.

For each loop's structure, two pieces of information were recorded; the number of literals in the bottom clause and the time taken to find the solution. Tests were conducted with the original naive learner, with the reformulated lazy learner, and with the reformulated lazy learner with the symmetry removal turned on. Each learner received the same input data file, and was run in identical conditions. Bottom clause sizes were obtained from one particular datum in each data set; the datum being used for this purpose was the same for all learners.

Results of these experiments are shown in Fig 4. For all experiments, the correct formula was always discovered by the learner.

For the simple nested loop, the results surprising show that the reformulated learner has a larger bottom clause and search time. This counter-intuitive result is actually due to the naive learner excluding some solutions that should be considered potentially true from the solution space. This can be seen in the second example, where $100ABC+1$, which should rightly be in the search space, is not found at all by the naive learner.

For the sorting style loop, it can be seen that reformulation actually reduces the search space. This is because the number of functional forms that could be created for this space, is actually smaller than the number of functions that the naive leaner could return for a target with only a single parameter. The creation

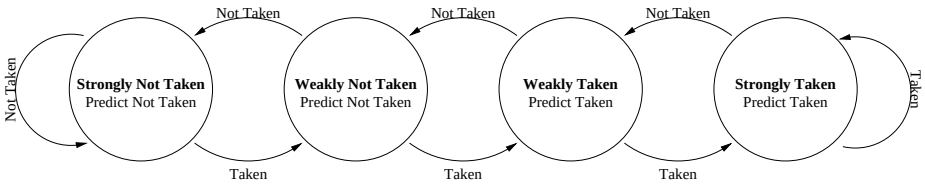

Fig. 5. Two bit prediction scheme

of learner operating on functional forms is clearly superior here, even in the absence of symmetry removal.

The effects of symmetry removal are evident throughout. Removing these unproductive clauses has a massive effect both directly on the bottom clause size and as a result on search time. While the reformulation and lazy learning expands the search space to not exclude those candidate hypotheses wrongly excluded by the naive learner, the symmetry reduction shrinks it down again. Crucially however, this is done while allowing a solution to always be found if it would be found in the absence of the symmetry removal. The large reduction in clause size and learning time illustrates the power and importance of this technique.

6 Related Work - ILP for Branch Prediction

The applicability of ILP in RTS is not limited to the loop bound learning task, and has also previously been tested on the problem of branch prediction analysis [19].

Modern pipelined microprocessors combine the approach of out-of-order execution with branch prediction and speculative execution to try to alleviate the problem of disrupting the instruction flow into the pipeline due to branches. A simple, but commonly used, dynamic branch prediction technique is an n-bit branch predictor [20] that uses the behaviour of the branch at its previous executions to predict its behaviour on the next occurrence. An n-bit predictor can be visualised as a finite state automaton (FSA) containing 2^n states. Each state predicts whether the branch will be taken or not at the next observation, and has deterministic transitions to other states based on the actual observed behaviour. An example of a two bit predictor is shown in Fig. 5. Common alternative branch prediction schemes include zero bit and one bit prediction.

While older processors have their branch prediction strategy well documented, for modern processors these details are normally commercially confidential. Without this information, WCET analysis must make conservative assumptions and produces unnecessarily pessimistic esimates. Bate and Kazakov [19] applied ILP learning to determine the type of branch predictor used by a processor. Test cases were produced for three configurations of hardware – zero bit, one bit and two bit predictors. The results of these executions were then fed into the learning engine. For each of the test cases the type of branch predictor was correctly

learned. Processing was of the order of tens of seconds, an acceptable time for this type of task.

While this work demonstrates definite progress on an aspect of WCET analysis, it suffers from one major drawback. In the published work, the branch predictor is simply chosen from one of several common types. In contrast, at the forefront of chip design, branch predictors are becoming available based on novel prediction schemes. It is details of precisely these chips that are most likely to be commercially confidential. One possible approach is to assume the general task of identifying a finite state automaton with an unknown number of states, choice of initial state and transitions. While the background knowledge may not be difficult to describe, the task at this level appears very hard in the general case. A possible way ahead is to study the known existing variations in the realm of branch predictor design and encode building blocks from which the target automaton is likely to be built.

7 Conclusion

This paper has presented an application of ILP for solving a particular problem in the area of Real-Time Systems. Specifically, the issue of determining the Worst Case Execution Time of a piece of code has been considered. A particular aspect of this problem was tackled; the question of determining the number of executions of a loop body.

Having demonstrated the potential of the technique, a much improved learner was implemented. Using more advanced ILP techniques, it was possible to vastly expand the range of loops for which the number of executions could be learned. Furthermore, the implemented techniques reduced the time needed to learn the loop count substantially.

The results showed that it was possible to accurately achieve bounds for nested loops in which the outer loop counter effected the range of the inner counter. This goes beyond what can be achieved by other existing WCET techniques.

Building on the work presented here, it should be possible to apply ILP to other areas of WCET analysis. In addition to the use of ILP for determining facts about program flow, related work was also shown in which ILP could be used to determine features of the hardware used to execute the code. There are many open issues in this area of WCET analysis, including the simulation of caches and out-of-order pipelines.

References

1. Turing, A.: On Computable Numbers, with an Application to the Entscheidungsproblem (7936). The Essential Turing: Seminal Writings in Computing, Logic, Philosophy, Artificial Intelligence, and Artificial Life, Plus the Secrets of Enigma (2004)

2. Kazakov, D., Bate, I.: 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)
3. Thesing, S., Souyris, J., Heckmann, R., Randimbivololona, F., Langenbach, M., Wilhelm, R., Ferdinand, C.: An abstract interpretation-based timing validation of hard real-time avionics software. In: Proceedings of the International Conference on Dependable Systems and Networks. (2003) 625–632
4. Coen-Porisini, A., Denaro, G., Ghezzi, C., Pezzè, M.: Using symbolic execution for verifying Safety-Critical systems. In: Proceedings of the 8th European software engineering conference held jointly with 9th ACM SIGSOFT international symposium on Foundations of software engineering. (2001) 142–151
5. Engblom, J.: Analysis of the execution time unpredictability caused by dynamic branch prediction. In: Proceedings of the 9th IEEE Real-Time and Embedded Technology and Applications Symposium. (2003) 152–159
6. A. Colin, A., Puaut, I.: Worst case execution time analysis for a processor with branch prediction. The Journal of Real-Time Systems **18**(2-3) (2000) 249–274
7. Healy, C., Arnold, R., Müller, F., Whalley, D., Harmon, M.: Bounding pipeline and instruction cache performance. IEEE Transactions on Computers **48**(1) (1999)
8. Li, Y.T.S., Malik, S.: Performance analysis of embedded software using implicit path enumeration. In: Proceedings of the 32:nd Design Automation Conference. (1995) 456–461
9. McMinn, P.: Search-based software test data generation: a survey. Software Testing, Verification & Reliability **14**(2) (2004) 105–156
10. Wegener, J., Sthamer, H., Jones, B., Eyres, D.: Testing real-time systems using genetic algorithms. Software Quality Journal **6**(2) (1997) 127–135
11. Muggleton, S.: Learning from Positive Data. Inductive Logic Programming: 6th International Workshop, ILP-96, Stockholm, Sweden, August 26-28, 1996, Selected Papers (1997)
12. Srinivasan, A.: The Aleph Manual. Computing Laboratory, Oxford University (2000)
13. Ernst, M.: Dynamically Discovering Likely Program Invariants. PhD thesis, University of Washington (2000)
14. Todorovski, L., Dzeroski, S.: Declarative bias in equation discovery. Proceedings of the Fourteenth International Conference on Machine Learning (1997) 376–384
15. Chapman, R.: Static Timing Analysis and Program Proof. PhD thesis, Department of Computer Science, University of York (1995)
16. Presburger, M.: Über die Vollständigkeit eines gewissen Systems der Arithmetik ganzer Zahlen, in welchem die Addition als einzige Operation hervortritt. s. n (1931)
17. Srinivasan, A., Camacho, R.: Numerical reasoning with an ILP system capable of lazy evaluation and customised search. The Journal of Logic Programming **40**(2-3) (1999) 185–213
18. Atkinson, K.: An introduction to numerical analysis. John Wiley (1989)
19. Bate, I., Kazakov, D.: New directions in worst-case execution time analysis. In: IEEE Congress on Evolutionary Computation (IEEE CEC 2008) within 2008 IEEE World Congress on Computational Intelligence (WCCI 2008). (2008)
20. Smith, J.: A study of branch prediction strategies. In: Proceedings of the 8th International Symposium on Computer Architecture. (1981) 135–148