Towards New Methods for Developing Real-Time Systems: Automatically Deriving Loop Bounds Using Machine Learning

Dimitar Kazakov[1] and Iain Bate[2]

[1]Artificial Intelligence Group
[2]Real Time Systems Group
[1,2]Department of Computer Science, University of York,
York, YO10 5DD, United Kingdom
e-mail: {dimitar.kazakov,iain.bate}@cs.york.ac.uk

Abstract

Most development, verification and validation methods in software engineering require some form of model populated with appropriate information. Real-time systems are no exception. However a significant issue is that the information needed is not always available. Often this information is derived using manual methods, which is costly in terms of time and money. In this paper we show how techniques taken from other areas may provide more effective and efficient solutions. More specifically machine learning is applied to the problem of automatically deriving loop bounds. The paper shows how taking an approach based on machine learning allows a difficult problem to be addressed with relative ease.

1 Introduction

The need for new methods for developing real-time systems emerged when considering how Worst-Case Execution Time (WCET) analysis could be improved. The key issue was obtaining sufficient information about the control flow of the software being analysed. This is widely recognised as a difficult problem independent of the subsequent methods used [6, 30, 4]. The need for appropriate information in order to make decisions and perform analysis is common within any form of science. Unsurprisingly we are not the first to realise this:

"Measure what can be measured, and make measurable what cannot be measured." - Galileo Galilei circa 1610

"When you can measure what you are speaking about, and express it in numbers, you know something about it; but when you cannot measure it, when you cannot express it in numbers, your knowledge is of a meager and unsatisfactory kind. It may be the beginning of knowledge, but you have scarcely, in your thoughts, advanced to the stage of science." - Lord Kelvin circa 1890

Further consideration of embedded real-time systems show a number of other problems for which knowledge is needed but difficult to come by. These span a wide range of areas, such as traditional design and verification techniques, evolutionary design using search algorithms and bio-inspired approaches, and mechanisms to support on-line adaption for dynamic systems. Examples include the actual release patterns of non-periodic tasks, the branch prediction behaviour of processors, bottlenecks in multiprocessor systems, ad-hoc routing in sensor networks and timing overruns caused by extraordinary events (e.g. due to fault handling code being executed). There are many more. One characteristic of the examples is a deterministic causal chain originating from the dependencies within the systems which means these relationships can be learnt. The field of machine learning is a mature field and the conjecture is that it may already hold many of the answers needed.

Irrespective of domain or application a common component of methods for developing real-time systems, and in particular their schedules, is the need for the WCET of software [3, 1]. WCET research has always been founded on the principle that specific details about the hardware and software are available. If the details are not available then a great deal of pessimism may be introduced into the analysis in order to get a safe upper bound for the WCET. However determining the details can be computationally expensive especially if inappropriate techniques are employed. This often leads to a trade-off between pessimism and the cost of analysis (both to initially set up the analysis and then to run the analysis). Examples of the forms of analysis that present difficulties include for software determining loop bounds and mutually exclusive paths, and for hardware analysing cache and branch prediction behaviour. All of these examples are made more complex as their behaviour is influenced by specific dependencies in the data generated during execution. Often this leads to analysis that requires significant manual input from the developer.

To counter these problems, Lundquist [12] proposed a combination of automated analysis for the majority of the problem with the rest of the information being provided either manually or accepting the

pessimism through not providing it. The disadvantage of Lundquist's approach is that it either relies on a bespoke compiler tailored to the needs of his analysis or significant post-processing of the output from a conventional compiler. For many developments this is undesirable. The solution in this paper proposed here is to determine methods by which the majority of information needed can be learnt automatically based on measurements. The rest of the information is to be provided by the developer with assistance, to ease the problem, from the analysis and machine learning. Whilst the aim is to move towards a generally applicable approach for real-time systems, this paper will focus on the problem of loop bound derivation. An important issue is that the estimates obtained need to be safe without too much pessimism. Where possible independent verification or validation should be carried out. The solution is demonstrated by considering the problem of loop bound derivation. In this case validation of the results is possible based on the work of Chapman [2]. However demonstration of validating the results is outside the scope of this paper.

The structure of the paper is as follows. Section 2 describes the previous work on WCET analysis and then motivates the need for new techniques building towards a set of objectives for this particular piece of work. Then, section 3 describes the problem to be addressed in more detail. Based on the problem definition, a framework for analysis is proposed in section 4. Next, in section 5 the framework is evaluated using some appropriate examples. Finally the paper is summarised and future work is proposed in section 6.

2 Background and Motivation

Despite being one of the first challenges identified in the area of WCET analysis [18, 19], the derivation of loop bounds remains largely unsolved [7]. However unlike other challenges within WCET, e.g. cache analysis, the problem has not become more difficult as technology has evolved. One reason why the automatic derivation loop bound challenge has potentially not been solved is that generally speaking it falls into the category of *undecidable* problems [21] defined by the *halting problem* [29].

Most work on WCET analysis assumes that loop bounds have been determined and associated with the software using an appropriate schema. The schema can either be in the form of a separate representation, in the case of Park's timing schema [18, 19], or embedded in the software, in the case of Chapman's annotations within SPARK Ada [2]. Both of these works however make the assumption the loop bounds have been derived and validated either manually or automatically. Where WCET analysis has been deployed in industry, it is manual derivation that has been used. This is extremely expensive and tedious especially as the information needs to be maintained during the lifetime of the project. This paper is concerned with the automatic derivation of the loop bounds.

Loop bounds are important to the WCET analysis problem as loop bounds directly influences a number of parts of the overall WCET analysis. This is independent of whether tree-based [10], path-based [21] or Integer Linear Programming [22, 25] forms of WCET analysis is used. Loop bounds influences both data and instruction cache analysis as the current cache state, and hence the likelihood of a cache miss, is directly affected by the number of times and how recent memory accesses have been made. Therefore loops within software are a significant issue. Similarly branch prediction behaviour is affected by control flow behaviour as the recent execution of branches changes the branch prediction state, and result, which is related to the number of times a loop is executed. Finally the pipeline behaviour is influenced as the loop bounds affects how many copies of the block(s) containing the loop body are concatenated when performing the analysis of the instructions.

Two principal approaches have been proposed for automatically deriving loop bounds. Firstly Lundquist [11] used symbolic execution. The work provides a great deal of promise, however there is the key limitation that it is reliant on having a cycle accurate processor simulator that can be heavily modified to support the proposed approach. The next significant attempt to solve the problem was by Gustafsson [6] using abstract interpretation. The early work was based on a subset of Real-Time Talk which was considered restrictive. Later the work was expanded to deal with the 'C' programming language [7] but this required specialised modification of a compiler and intermediate language representation. Whilst the proposed approach has many interesting features, it is considered too restrictive a solution. The other potential issue with this approach is it relies on the user providing information on the ranges of some variables if it is to be accurate, i.e. not too pessimistic.

In contrast machine learning based on measurement provides an interesting alternative as loop bounds have well-defined relationships. Also, the safeness of the results derived can still be validated using the proof mechanisms of Chapman [2] if the same assumptions are made. Demonstration of this is out of the scope of this paper.

Machine Learning (ML) aims at describing the properties of a set of observations from a given source, and/or making predictions about the nature of future observations from the same source. Both goals are achieved by changing the representation of available data as expressed in its original form (or *object language*) into another representation (using another formalism, known as *hypothesis language*). The new representation copies closely the information encoded

in the original data, but is usually more general, and allows one to make statements about yet unseen cases. ML can be seen as the search for a mapping from a set of inputs to an output; this mapping is often a function; when it is Boolean (i.e., a *predicate*), it could be seen as defining a *concept* as a subset C of its domain (or *universe*) U. In the context of loop bound determination, this means relationships between the software's variables and the loop bound can be determined.

No ML algorithm can make predictions unless it employs a *bias*. In the case of concept learning, this means that not all possible subsets of the universe U are expressible in the hypothesis language (see [14] for details of the argument). In general, the bias will restrict the range of possible functions (models, hypotheses) that can be used as hypothesis language. For instance, the set of data points $\{(0,0),(\pi,0),(2\pi,0)\}$ can variously be modelled by the functions $y=0, y=\cos x$ *or* $y=x(x-\pi)(x-2\pi)$, depending on the bias, which may restrict the hypothesis to a linear, trigonometric or polynomial function.

Such a bias is also called *language bias* to distinguish it from the *preference bias*, allowing one to choose between alternative models with equal coverage of the available data. Here some simple, but general principles (heuristics) are often employed. For instance, Occam's razor [14] favours the simplest hypothesis language, while the Minimal Description Length (MDL) bias [23] suggests a trade-off between the complexity of the hypothesis language and that of the resulting representation of the data.

The area of ML focussing on the search for quantitative laws, expressed as equations, is known as equation discovery. When an initial draft of the equation is provided, the process is known as equation revision [27]. In this case, initial input is required from experts, but the changes carried out by the learner can be non-trivial, and result in substantial improvements [28].

Inductive Logic Programming (ILP) is a form of ML where both data and hypotheses are expressed in first order logic [17]. The great advantage of ILP is in its ability to vary its language bias through the use of *background knowledge*, the only set of predicates that can be used to describe the target hypothesis. For instance, the predicates `parent/2` and `male/1` can be used to learn the definition of `father/2`. While such categorial hypotheses are more common, it is possible to define operators and standard functions as background knowledge, and search for the equation that best models the data. This potential is exploited for instance by the system LAGRAMGE [26]. Another approach where a similarly flexible hypothesis space is combined with evolutionary search is Cartesian Genetic Programming [13].

The area of real-time systems analysis, and in particular WCET analysis, is in many ways an excellent ground for the application of ML and ILP.

1. The data may however be complex, yet it contains no noise, a major issue in almost any other ML application area.
2. The control variable range is discrete, and often can be sensibly restricted to an interval. One can make a simplifying assumption and treat such data as measured on a nominal scale. As a result, even the simplest ILP algorithms can be applied without any adaptations.
3. Implementing equation discovery with ILP means one has to deal with "positive only" learning: the data consists of examples (instantiations) of the equation, but no counter-examples are provided. ILP, where both are usually needed, overcomes the issue by generating random examples, and assuming that the substantial majority of those are counter-examples. Choosing a sufficiently large range for the control/input variables will guarantee this assumption, while their discrete nature makes it easier to choose a sensible sampling strategy (already available in off-the-shelf ILP tools, such as Progol [15]).
4. As long as the analysed software is not part of a control loop involving external hardware (including the "physical system"), the cost of obtaining data is very low, which means one can test any hypothesis at will, e.g., to meet the requirements of a statistical test. Since the input variable domains can be very large, one can employ *active learning* to let the learner select the examples, for instance, to minimise uncertainty in the model [5]. The whole process of learning and testing can be automated in a *closed loop machine learning* style [8].

3 Problem Definition

Two types of loops are to be considered here - flat loops and nested loops. A typical flat loop is illustrated in Figure 1 and a nested loop in Figure 2. Here is assumed the semantics of an individual loop means i in Figure 1 takes the values $m_1, m_1+1, m_1+2, \cdot\cdot, m_2$. In the general case the loop count for the body of the flat loop is given in equation (1) and the nested loop in equation (2). A generalised form of equation (2) is presented in equation (3).

$$lc_{flat} = m_2 - m_1 + 1 \quad (1)$$

$$lc_{2\_lev\_nest} \leq (m_2 - m_1 + 1) \cdot (n_2 - n_1 + 1) \quad (2)$$

$$lc_{gen\_nest} \leq \prod_{\forall i \in loops} (m_{i,2} - m_{i,1} + 1) \quad (3)$$

Clearly an arbitrary level of nesting can be supported. With respect to the loop bounds (i.e. m_1, m_2, n_1, n_2)

```
for i = m1 to m2 loop
  loop body
end loop
```

Figure 1. Flat loop

there are two types - invariant and variant. The variant case is the most difficult as it is data dependent and in the general case can be expressed with arbitrary complexity. Also the scope of the variables that make it data dependent can be the whole program and their range the limits of the data type that represents them. These reasons are key to why the problem is classed as *undecidable* in the general sense [2].

To make the problem manageable the typical restrictions of real-time systems proposed by Chapman [2] are followed. The following restrictions were chosen by Chapman as they fall within the subset defined by the Presburger arithmetic subset which helps make the problem decidable. It is considered the restrictions are representative of typical coding standards for real-time systems. Other work [9, 21] have adopted more stringent restrictions on the loop bounds to be supported in order for the problem to be *decidable*.

1. Integer constants
2. Variables that are constant integers
3. +, - operators
4. Multiplication by an integer constant
5. Variables that are a loop index for an enclosing **for** loop

The last of these restrictions could lead to pessimism if the maximum value given by equation (2) is used, i.e. $(m_2 - m_1) \cdot (m_2 - m_1 - 1)$. Consider the typical nested loop used in sort routines shown in Figure 3. In this case the loop bound of the nested body is given in equation (4). The result, lc_{sort}, is considerably less (actually half the value) than the general case for a nested loop, $lc_{2\_lev\_nest}$, which means an overestimate, and hence pessimism, can be avoided without compromising the safeness of the result.

$$lc_{sort} = \frac{(m_2 - m_1) \cdot (m_2 - m_1 - 1)}{2} \qquad (4)$$

```
for i = m1 to m2 loop
  for j = n1 to n2 loop
    loop body
  end loop
end loop
```

Figure 2. Nested loop

```
for i = m1 to m2 loop
  for j = m1 to m2 − 1 − (i − m1) loop
    if a[j] < a[j+1] then
      ...
  end loop
end loop
```

Figure 3. Sort routine

4 Analysis Framework

There are three key parts to the analysis framework. Firstly, the software under test should be examined in order to provide information on which variables are within scope and monitored to indicate how many times each block of the software is exercised. Approaches for doing this are well understood so the mechanisms are not considered further here. It is considered these approaches can be carried out as a separate testing activity or to save effort as part of testing already carried out, e.g. structural or functional testing.

Secondly, there are test cases used to exercise the software. These should be sufficient to identify:

1. whether a flat loop or the outer loop has constant loop bounds or they are dependent on variable(s) that are constant within the scope of the loop
2. where the loop bound is constant the value needs to be determined
3. where the loop bound is dependent on constant variables which variables are featured and how they are related with respect to the allowed arithmetic operators

The method used is to randomly alter each variable within the scope of the loop until N consecutive iterations have been performed without anything new being learnt about the loop bounds. The value of N will be considered later. This will be sufficient to identify whether the variable is related to the loop bound and for the derived nature of the relationship to be established.

Finally, the information from testing is used to learn the characteristics of the software in order to determine the actual loop bounds or where this is not possible the relationship that defines it. Where inner loop(s) of a nested loop potentially has a variable loop bound (i.e. it is dependent on the number of times an outer loop iterates) then only the case resulting in the worst case execution time is represented in the training data. That is, the case in which the maximum number of iterations of the inner loop occurs given the number of times the outer loops executes.

We look at the execution time analysis of nested loops to demonstrate the potential of an empirical,

machine learning approach. The nested loop example is chosen as it is more complex, interesting and challenging than the flat loop and effectively the flat loop case has to be solved as part of considering the nested loop. We assume the data available contains the number of times each of the nested loops has been (re-)entered, as well as the overall number of times the body of the innermost loop has been run. This information can easily be obtained by instrumenting the code during testing. It is assumed the instrumented code does not alter the control flow, and hence loop bounds, of the system being tested. The target hypothesis then expresses the latter parameter as a function of the other observations. The ILP learner is provided with background knowledge in the form of algebraic operators. While it is possible to define a hypothesis language containing inequalities, which could be used to express upper (worst-case) bounds, here we look for the *exact* formula. Once the data has been made available, the range of all variables is capped. As we mentioned earlier, this allows the ILP learner to carry out positive-only learning, i.e., sample the range and generate random data entries, which are substituted for counter-examples. The greater that range, the higher the likelihood for these random entries to be negative examples. For instance, for a function of type $(D \times D \times \cdots \times D) \rightarrow D$, where D is a set of size 10, a random n-tuple has a 90% chance of not being a valid instantiation of that function. The final step, which involves a certain amount of craftsmanship, is to define how the search through the hypothesis space is guided, and pruned. Both are done to improve the search efficiency, and to cap its maximum complexity. This step is not unique to the application at hand, it is commonly done whenever ILP is used [16], and the choices made in our case are given by the formal properties of the data, rather than based on specific domain knowledge. Still, a good understanding of the way an ILP learner operates is required, and the need to master this element of the ILP setup is one of the main obstacles to its wider use.

5 Evaluation

Here we illustrate the idea with a number of test cases. Initially, we look at the case of two nested loops, which are executed A and B times respectively (cf. Figure 2). The inner loop body will obviously be run $A * B$ times. The data set is generated by enumerating all combinations of the two independent variables for values between 1 and 5, combining these with their product, and storing the resulting instances as clauses of the target predicate `tp/3`. Only 9 out of all 25 clauses are included in the data set. In line with the restrictions on real-time systems, discussed at the end of Section 3, two operators, multiplication and addition are supplied as background knowledge, i.e., these operators are the only ones that can be used in hypotheses to link the three variables (see Table 1). The ILP learner is also informed that the range of all three attributes is limited to $\{1, \ldots, 25\}$. The ILP learner Progol4.4 is then run on the data. The output is given in Table 2. The only clause found matches the data perfectly: $C = A * B$, i.e., the third attribute is the product of the other two.

The second example extends the problem to three nested loops with independent ranges (see Figure 4). The hypothesis language is based on the same two operators, "$*$", and "$+$". The target predicate now contains 4 arguments, the 3 independent variables A, B and C, and their product D. The range of A, B and C was $\{1, \ldots, 5\}$; just over half (63) of all 125 combinations were used as training examples. Progol considered $\{1, \ldots, 125\}$ as the range for all four variables. This time, the learning took 3.56s (all trials are performed on a 750MHz Pentium 3 PC), and the final hypothesis contained four cases shown in equations (5–8).

$$D = B * C \tag{5}$$

$$D = A * B * A \textit{ when } C = A \tag{6}$$

Table 1. Data set 1

```
% Background knowledge

sum(A,B,C) :-
  A is B + C.

product(A,B,C):-
  A is B * C.

% Target predicate:
% tp(A,B,Product)

tp(1,1,1).
tp(1,2,2).
tp(1,5,5).
tp(2,3,6).
tp(3,1,3).
tp(3,4,12).
tp(4,2,8).
tp(4,5,20).
tp(5,3,15).
```

Table 2. Progol output for data set 1

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

[Total number of clauses = 1]

[Time taken 0.01s]
```

```
for i = 1 to A loop
 for j = 1 to B loop
  for k = 1 to C loop
   loop body
  end loop
 end loop
end loop
```

Figure 4. Example 2

$$D = A * A * C \text{ when } B = A \quad (7)$$
$$D = A * B * C \quad (8)$$

This result requires some explanation. The hypothesis is made of the four rules, with no particular order of application. However, they have been learnt one at a time, choosing the "best" rule for the data at hand, and discarding the examples it covers, then looking for the next best, etc. Progol has a built-in preference bias in favour of shorter rules, using as few operators and variables as possible. A look at the actual Progol output (in Table 3) makes clear the effect of this bias. Equation (5) covers all cases where $A = 1$ (13 in total). Equations (6–7) have also been produced before equation (8), which subsumes them, as it can be easily seen. Subsumption of clauses under logical implication is undecidable [24]. However, the very specific case here could probably be automated by spelling out the equivalent algebraic expression for each rule and using the relatively simple θ-subsumption procedure between terms [20]. For instance, representing equations (6) and (8) as Prolog terms, and using the Sicstus Prolog predicate `term_subsumer/3` from library `terms`, one obtains the expected result. Another possible approach is to compare the functions represented by each rule and keep the one growing fastest. This approach would eliminate rule (5) when compared with rule (8).

It has to be noted that rule (5) is overly general, and incorrect for a number of training examples, as it completely ignores argument A. There are several factors that may have led to this result, among them: (1) Progol's "the shorter, the better" preference bias, (2) the fact that the induction step is data driven, and the first training example is consistent with the rule, and, finally, (3) in order to speed up the hypothesis search, Progol was told the rule would be used in `tp(+,+,+,+)` mode, that is, with all arguments instantiated. The latter means the rule would not be inconsistent, as whenever the D in the data is not equal to the product of B and C, the rule would not fire. A different choice of the learner setting may have led to a different result, but it was deemed important to point out that such inconsistent rules may be produced.

Table 3. Progol output for data set 2

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

```
for i = 1 to A loop
 for j = 1 to B loop
  loop body
 end loop
end loop
for k = 1 to C loop
 loop body
end loop
```

Figure 5. Example 3

The third case models two nested loops followed by another (see Figure 5). If the range of the three counters is A, B and C, the overall execution time is $A * B + C$. In this case, we provide the ILP learner with all 125 examples for A, B, and C taking values between 1 and 5. Two rules are obtained as a result (in less than 5s). Again, the first is a specific case of the second:

$$D = A + B \text{ when } C = A \quad (9)$$
$$D = A * B + C \quad (10)$$

The first rule is true for all examples where $A = 1$. As the learning process is data driven, and the first example in the data set does correspond to $A = 1$, the rule is easily learned. Reordering the examples can affect the final hypothesis, but is by no means a general solution, as the learner may just pick up a different set of specific cases. One can even imagine a situation where a very unlucky training data sample is entirely covered by specific cases, and learning fails to reveal the underlying general pattern.

The fourth case studied is one where the number of times a piece of code is executed depends on two observable variables, and a constant, which is not supplied, but hard-coded and not directly observable. The case can be illustrated by the pseudocode in Figure 5 where C is a fixed constant. The training data consists of triples where the first two arguments, A and B, are independent variables, and the third, C, is the execution time being modelled. The data contains 25 examples – as before, A and B take values between 1 and 5. C is then computed as $A * B + 5$. The range of all variables is capped at 30, the maximum value for C. This time, the only background predicate used is one defining the class of functions $x * y + const$. The correct rule is the only one produced by the learner, in less than 0.1s. Extending the hypothesis language with separate multiplication and

addition operators, as used in the previous examples, slows down the search to about 0.5s, but does not change the result.

Finally, we have looked at the case of nested loops where the inner loop bounds are functionally dependent on the outer loop counter. The sort routine in Figure 3 is used as an example. To generate the data, we assume (with no loss of generality) that the parameter m_1 in equation (4) is equal to 1. We then used the equation to generate pairs of numbers representing the upper bound m_2 and the corresponding number of times of running the inner loop body (see Table 4). The variable range was set to $m_2 \in \{1, \ldots, 30\}$. The simulated data used in the sort routine example represents the maximum number of steps needed to sort a list of certain length. This is equivalent to preprocessing the data, so that of all permutations of input list elements, only the one resulting in the WCET is represented in the training data.

Table 4. Sort routine data set

```
% 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).
  ...
```

Using `sum/3` and `product/3` as background knowledge, Progol4.4 found a one-rule model in less than four seconds (Table 5).

Table 5. Progol output for the sort routine

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

This translates to a system of three equations shown below.

$$C = A * A \tag{11}$$

$$D = A + B \tag{12}$$

$$C = B + D \tag{13}$$

Note that in this case the division operator was not part of the background knowledge, nor was 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 the following equation which is, of course, identical to the formula in equation (4).

$$B = \frac{A * (A - 1)}{2} \tag{14}$$

6 Summary and Future Work

These early experiments are meant as a proof of concept, but they already show the potential for empirically deriving accurate upper bound estimates with acceptable amounts of processing for loop bounds expressed in a way that have so far been unknown within the RTS community. A number of examples were considered. By no means are these a comprehensive assessment, but we believe they are relatively general in their nature and provide a representative sample of the types of loops that will be found in practice. The studied cases were chosen to illustrate a range of issues: ILP can learn from a small number of examples; these examples do not need to cover the whole range of the variable domains, but such an exhaustive data set is easy to generate and can be used. In addition, the last presented case shows that the dependency of an inner loop bound on the outer loop counter can be successfully detected and hence potential pessimism can be avoided. The results also demonstrate some of the potential difficulties that the emerging methodology has to address, such as the need to post-process the rules learnt to introduce a partial order (lattice) of generality among them, and from a set of rules that do not subsume each other, select the most pessimistic one. There are another two issues that are inter-related: the way in which the training data sets are generated, and the choice of a testing framework, which assigns a level of confidence to the ILP hypothesis. As the data does not contain noise, and can be extended at will, it is likely that an accurate hypothesis could be learnt from a relatively small data set, and then tested extensively on a much larger number of cases. These cases could be derived during the usual functional testing performed of systems. Very unusually for a machine learning application, the final hypothesis should always test 100% accurate—or it will have to be reconsidered when a counter-example is found.

Among the advantages of ILP is the fact that it can include any type of function in its hypothesis space through an adequate choice of background knowledge. On the other hand, the currently recommended restrictions on loop bounds mean that the hypothesis space can be restricted to a set of linear functions, in the way we dealt with example 4 here. This discrepancy can be resolved in either direction, by restricting the ILP learner to a much more specialised application, or by relaxing the constraints imposed by Chapman and others. Our future work will look at the wider exploitation of machine learning within

the design and analysis of real-time systems.

References

[1] A. Burns and P. Puschner, editors. *Special Issue: Worst-Case Execution Time Analysis*, volume 18(2–3) of *Real-Time Systems Journal*. Kluwer Academic Publishers, May 2000.

[2] R. Chapman. *Static Timing Analysis and Program Proof*. PhD thesis, Department of Computer Science, University of York, March 1995.

[3] R. Wilhelm (Editor). Special issue on timing analysis and validation for real-time systems. *Real-Time Systems Journal*, 17(2/3), Nov 1999.

[4] C. Ferdinand, F. Martin, and R. Wilhelm. Applying compiler techniques to cache behavior prediction, June 1997.

[5] Y. Freund, H.S. Seung, E. Shamir, and N. Tishby. Selective sampling using the query by committee algorithm. *Machine Learning*, 28(2–3):133–168, 1997.

[6] J. Gustafsson. Analyzing execution-time of object-oriented programs using abstract interpretation. Technical Report DoCS 00/115, Department of Computer Science, Uppsala University, Sweden, May 2000.

[7] J. Gustafsson, A. Ermedahl, and B. Lisper. Towards a flow analysis for embedded system c programs. In *Proceedings of the 10th IEEE International Workshop on Object-oriented Real-time Dependable Systems (WORDS 2005)*, 2005.

[8] R. King, K. Whelan, F. Jones, P. Reiser, C. Bryant, S. Muggleton, D. Kell, and S. Olivier. Functional genomic hypothesis generation and experimentation by a robot scientist. *Nature*, 427(6971):247–252, 2004.

[9] E. Kligerman and A.D. Stoyenko. Real-time Euclid: a language for reliable real-time systems. *IEEE Transactions on Software Engineering*, vol.SE-12, no.9 A06:941–9, Sept. 1986. IEEE Trans. Softw. Eng. (USA).

[10] S. Lim, Y. Bae, G. Jang, B. Rhee, S. Min, C. Park, H. Shin, K. Park, and C. Kim. An accurate worst case timing analysis for risc processors. *IEEE Transactions on Software Engineering*, 21(7):593–604, July 1995.

[11] T. Lundqvist and P. Stenstrom. An integrated path and timing analysis method based on cycle-level symbolic execution. *Real-Time Systems*, 17(2/3):183–207, November 1999.

[12] T. Lundqvist and P. Stenstrom. A method to improve the estimated worst-case performance of data caching. In *Proceedings of the 6th International Conference on Real-Time Computing Systems and Applications (RTCSA'99)*, pages 255–262, December 1999.

[13] J. Miller and P. Thomson. Cartesian genetic programming. In R. Poli, W. Banzhaf, W. Langdon, J. Miller, P. Nordin, and T. Fogarty, editors, *Genetic Programming, Proceedings of EuroGP'2000*, volume 1802, pages 121–132, Edinburgh, 2000. Springer-Verlag.

[14] T. Mitchell. *Machine Learning*. McGraw-Hill, 1997.

[15] S. Muggleton. Inverse entailment and Progol. *New Generation Computing*, 13:245–286, 1995.

[16] S. Muggleton and J. Firth. *Relational Data Mining*, chapter CProgol4.4: a tutorial introduction, pages 160–188. Springer-Verlag, 2001.

[17] S. Muggleton and L. De Raedt. Inductive logic programming: Theory and methods. *Journal of Logic Programming*, 19/20:629–679, 1994.

[18] C. Park. Predicting program execution times by analyzing static and dynamic program paths. *Real-Time Systems*, 5(1):31–62, 1993.

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

[20] G. Plotkin. A note of inductive generalization. In B. Meltzer and D. Mitchie, editors, *Machine Intelligence 5*, pages 153–163. Edinburgh University Press, 1970.

[21] P. Puschner and C. Koza. Calculating the maximum time of real-time programs. *Real-Time Systems*, 1(2):159–176, 1989.

[22] P. Puschner and A. Schedl. Calculating the maximum execution times with linear programming techniques. Technical report, Institut fur Technische Informatik, Technische Univeristat Wien, 1995.

[23] J. Rissanen. Modeling by shortest data description. *Automatica*, 14:465–471, 1978.

[24] M. Schmidt-Schauss. Implication between clauses is undecidable. *Theoretical Computer Science*, 59:287–296, 1988.

[25] H. Theiling and C. Ferdinand. Combining abstract interpretation and ILP for microarchitecture modelling and program path. In *Proceedings of the IEEE Real-Time Systems Symposium '98*, pages 144–153, Dec 1998.

[26] L. Todorovski and S. Džeroski. Declarative bias in equation discovery. In *Proc. 14th International Conference on Machine Learning*, pages 376–384. Morgan Kaufmann, 1997.

[27] L. Todorovski and S. Džeroski. Theory revision in equation discovery. *Lecture Notes in Computer Science*, 2226:389+, 2001.

[28] L. Todorovski, S. Džeroski, P. Langley, and C. Potter. Using equation discovery to revise an Earth ecosystem model of carbon net production. *Ecological Modelling*, 170:141–154, 2003.

[29] A. Turing. On computable numbers, with an application to the Entscheidungsproblem. *Proceedings of the London Mathematical Society*, Series 2(42):230–265, 1936.

[30] R. White, F. Mueller, C. Healy, D. Whalley, and M. Harmon. Timing analysis for data caches and set-associative caches. In *Proceedings of the IEEE Real-Time Technology and Applications Symposium*, pages 192–202, 1997.