New Directions in Worst-Case Execution Time Analysis

Iain Bate and Dimitar Kazakov

***Abstract*— Most software engineering methods require some form of model populated with appropriate information. Real-time systems are no exception. A significant issue is that the information needed is not always freely available and derived it using manual methods is costly in terms of time and money. Previous work showed how machine learning information derived during software testing can be used to derive loop bounds as part of the Worst-Case Execution Time analysis problem. In this paper we build on this work by investigating the issue of branch prediction.**

I. Introduction

IN recent years there has been a move towards measurement-based approaches for timing analysis and more specifically for Worst-Case Execution Time (WCET) analysis problem [5]. The reasons for this move are that the hardware available for embedded real-time systems is becoming increasingly complex to model and consequently analysis is proving difficult.

Early work on measurement-based techniques used search-based test case generation approaches. Genetic algorithms (GAs) were used to generate test data that made the software execute its longest path [13]. However the work also showed that a purely black box approach to the problem did not scale and could not deal with the complexity of modern processors. Other work looked to adopt probabilistic methods [2]. However this work also acknowledged the limitations of black box approaches and the need for hybrid approaches combining static and measurement-based analysis, i.e. there are significant benefits of having models of the system to help drive the measurement-based analysis. For example, Wegener recognises his work would benefit from white box knowledge and multiple objectives such that the genetic algorithm maximised objectives other than WCET, e.g. the number of times round a particular loop [21]. We are currently working on a project that uses a two pronged approach where techniques from the artificial community, primarily machine learning, are used to establish models that then support more targeted test case generation.

Typically the WCET problem can be split into three parts - program flow analysis, low level analysis and the integration / calculation phase. Program flow analysis concentrates on understanding the structural characteristics of the software, e.g. loop bounds. The low-level analysis considers how basic blocks execute on the target processor. A basic block has a single entry and exit point, and no jump instructions are contained within it. Finally the integration / calculation phase takes the results from the other two phases to determine the overall WCET. In [8] it is shown how machine learning could be used to model the program flow characteristics of software. Specifically loop bounds are learned.

Branch prediction analysis, which is part of the low-level analysis, represents an extremely challenging problem in that it requires two models to be obtained and then integrated. The first model is how the branch predictor itself works. It is noted a full reverse engineering is not needed. Instead only the relevant information is obtained, i.e. the basic decision logic for deciding whether the branch is taken or not. Then, a second model is needed which contains knowledge of how the branch predictor affects the execution of the software. The two models are used with information from the program flow analysis, e.g. loop characteristics, to determine the maximum number of branch predictions that can occur as part of the integration phase of the WCET analysis. To demonstrate the potential of the approach relatively simple predictors are analysed. These are chosen as they have reasonable levels of complexity but not that complex that the principles cannot be demonstrated in the space available. Also in our experience, and that of others [6], more complex branch predictors have a negative effect on the WCET and its analysis.

The structure of the paper is as follows. Section II presents background material on branch prediction analysis and our previous work on learning program flow characteristics. Then, section III surveys the previous work on branch prediction analysis and machine learning. Section IV explores how branch prediction analysis can be derived using machine learning and considers potential solutions. An evaluation is given in section V. Finally, section VI presents a summary.

II. Background

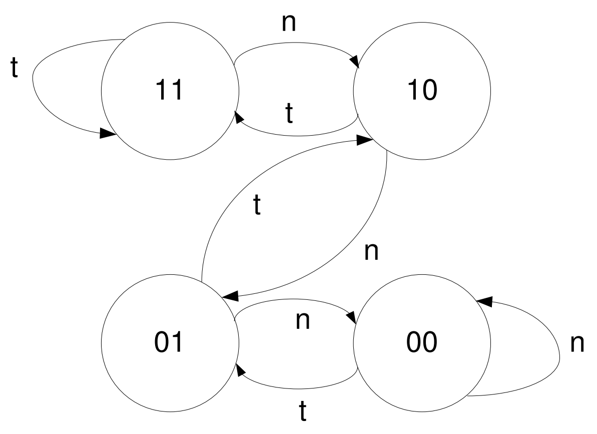
The purpose of this section is to present our previous work on branch prediction analysis and loop bound determination in order to set the context for the work within this paper.

A. Branch Prediction Analysis

The purpose of this section is to present our previous work on branch prediction analysis. This knowledge is later used as part of the integration phase of the WCET analysis. Modern 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 a *bimodal branch predictor* [18] that stores branch history in a 2^n-entry branch history table (BHT), which is indexed by the n lower bits of the address of the branch instruction.

A sequence of executed branches can be interpreted as a string over the alphabet $\Sigma = \{n, t\}$ and the two-bit saturating

Iain Bate and Dimitar Kazakov are with the Department of Computer Science, University of York, York, YO10 5DD, United Kingdom (email: {iain.bate,dimitar.kazakov}@cs.york.ac.uk).



Bits	State	Prediction	Key
`00`	strongly not-taken	n	SN
`01`	weakly not-taken	n	WN
`10`	weakly taken	t	WT
`11`	strongly taken	t	ST

Fig. 1. Two-bit prediction scheme

Pattern	Class	Initial Predictor State			
		SN	WN	WT	ST
t^i	T	2	1	0	0
n^i	N	0	0	1	2
$(tn)^i$	A	i	**2i**	i	i
$(nt)^i$	A	i	i	**2i**	i
$(n^{j-1}t)^i$, $j = 3$	N	i	i	$1 + i$	$3 + i$
$(n^{j-1}t)^i$, $j > 3$	N	i	i	$1 + i$	$2 + i$
$(t^{j-1}n)^i$, $j = 3$	T	$3 + i$	$1 + i$	i	i
$(t^{j-1}n)^i$, $j > 3$	T	$2 + i$	$1 + i$	i	i

TABLE I
NUMBER OF BRANCH MIS-PREDICTIONS

counter of a bimodal branch predictor can be represented by a *deterministic finite automaton* (DFA),whose states and transitions are depicted in Figure 1. n indicates a branch is *not-taken* and t that the branch is *taken*. For example, the pattern $(t^{i-1}n)^j$ represents the behaviour of a branch associated with a loop construct that iterates i times and is repeated itself j times. A feature of a loop construct is that the conditional test of the loop index resolves to `true`, $i - 1$ times, and `false` once (on loop exit). The scheme provides some degree of hysteresis and thus is less affected by occasional changes in branch direction. The state of the counter is updated after the branch outcome has been resolved.

Table I, summarised from [1], details the maximum number of branch mis-predictions that can be expected for various branch patterns being repeated i times, depending on the initial state of the predictor. The second column in this table defines for each branch pattern the classification of the associated branch instruction into *taken-biased* (T), *not-taken-biased* (N), and *alternating* (A). The worst-case number of mis-predictions, i.e. each instance of a branch is mis-predicted, is where the branch alternates between its *taken* and *not-taken* directions. In the table, the two cases corresponding to this worst-case predictor scenario are highlighted in bold. The last four rows represent the branch behaviour of a loop typically generated by a compiler for $j = 3$ and $j > 3$ iterations, respectively. There are many other alternatives to branch prediction including zero bit (i.e. effectively no prediction), one bit prediction and the general N-bit predictors.

B. Using Machine Learning for Determining Loops

In [8] a framework was presented based on typical restrictions of real-time systems proposed by Chapman [3] to make the problem of analysing the bounds for both flat and nested loops manageable. 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. These are: integer constants; variables that are constant integers; +, - operators; multiplication by an integer constant; and variables that are a loop index for an enclosing **for** loop. It is considered the restrictions are representative of typical coding standards for real-time systems. Other work [10], [16] adopted more stringent restrictions on the loop bounds to be supported in order for the problem to be *decidable*.

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. This 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.

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. An Inductive Logic Programming learner is provided with background knowledge in the form of algebraic operators which in this case relates to the operations allowed by the Presburger subset. Further details of the method and evaluations can be found in [8].

III. Related Work

This section presents related work for two areas: branch prediction analysis and machine learning.

A. Branch Prediction Analysis

Several static analysis techniques for obtaining WCET estimates of real-time programs have emerged over the last two decades. The first work on execution time analysis for microprocessors using dynamic branch prediction techniques to model the branch target buffer (BTB) of a Intel Pentium microprocessor is presented by Colin and Puaut [4]. Disadvantages of their approach include it is based on source-code, rather than object-code, which ignores additional branches introduced by the compiler, and it does not integrate the results with other parts of WCET analysis. Mitra et al. [12] present a framework for modelling the effects of global-history branch prediction schemes on WCET analysis. This only considers single bit predictors, however it does deal with global history predictions. All the techniques require a basic knowledge of how the branch predictor works.

B. Machine Learning

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 statements to be made 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 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 not all possible subsets of the universe U are expressible in the hypothesis language (see [11] for details of the argument). In general, the bias will restrict the range of possible functions (models, hypotheses) that can be described by the hypothesis language. For instance, the set of data points $\{(0,0),(\pi,0),(2\pi,0)\}$ can 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 a choice 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 [11] favours the simplest hypothesis language, while the Minimal Description Length (MDL) bias [17] suggests a trade-off between the complexity of the hypothesis language and that of the resulting representation of the data.

The area of ML focusing 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 [19]. In this case, initial input is required from experts, but the changes carried out by the learner can be non-trivial, and result in large improvements [20].

Inductive Logic Programming (ILP)is a form of ML where both data and hypotheses are expressed in first order logic [15]. 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.

IV. Learning Branch Prediction

The purpose of this section is to explore the issues in learning branch prediction characteristics. This is described in the following subsections that describe how test cases can be generated which are then be used within a learning framework to determine how the branch predictor affects the execution of software.

A. Test Case Generator for Branch Prediction

An output of the test case generator is the set of branch prediction results for each individual branch which can then be amalgamated by the learner to gain the desired result. Each output forms a tuple consisting of <current address, branch prediction, actual branch>. These represent an identifier (*current address*) for the branch being considered, whether the branch is correctly predicted or not (*branch prediction*), and whether the branch was taken or not (*actual branch*). From this tuple all the information needed is available to determine the set of states, and associated transitions, within a branch predictor assuming there are an appropriate set of test cases.

To learn the characteristics of a branch prediction unit requires each of the possible state transitions to be covered by the test cases independent of its initial state. Clearly given knowledge of the particular branch predictor then a sufficient and necessary (i.e. minimal) set of test cases can be generated. In the case of the two-bit predictor in Figure 1 there are eight state transitions. However without this knowledge the only option is to perform comprehensive testing or to use search-base testing techniques. The situation is eased due to the following characteristics of the data being learned from.

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 [14]).
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 [7]. The whole process of learning and testing can be automated in a *closed loop machine learning* style [9].

The most important of these characteristics is the fact we have positive only learning means the results only get better with more data. Hence there is no harm in there being too much data other than an increase in computation time, as long as the solution remains tractable. Figure 2 shows the set of test cases used where *IS* is a variable representing the initial state of the predictor.

```
for (IS $\in$ initial_states)
  for (i=1; i $\leq$ 12; i++)
    for (j=1; j $\leq$ 12; j++)
      simulate loop bias (IS, t, i, j);
```

Fig. 2. Test case generator

The purpose of the procedure *simulate loop bias* is to provide output target predicates in the correct format with an appropriate initial state and bias. It produces output based on executing a nested loop with either a $Taken(t)$ or $NotTaken(n)$ bias. The parameters passed to it are: *IS* - initial state; *bias* - t or n; *i* - loop count for the outer loop; and *j* - loop count for the inner loop

An example output from the test case is given in Figure 3. It shows the target predicates for an outer loop bound of 2, an inner loop bound of 3 and an initial state ST. In this example the target predicate ttp has the following information associated with it - test case number, outer loop bound and overall loop count which are 6, 2 and 3 respectively. It is noted that the loop count (or difference in

```
% ttp(CaseID,NumberOfMispredictions,
%     NumberOfPredictionsMade).
% tp(Case,Index,BranchPrediction,BranchTaken).

% Case 15:
% Outer Loop has bound 2, Inner Loop has a
% bound 3, 2 x 3 = 6 predictions to make,
% Initial State is Strongly Taken.
ttp(15,2,6).

tp(15,1,t,t).
tp(15,2,t,t).
tp(15,3,t,n).
tp(15,4,t,t).
tp(15,5,t,t).
tp(15,6,t,n).
```

Fig. 3. Example Output from Test Case

bounds) for any given loop is equal to the number of branch predictions that must be made. The target predicate tp has the following information (in the order shown) - test case number, overall loop iteration number, branch prediction and actual branch. For example $tp(15, 3, t, n)$ refers to test case number 15, loop iteration 3, a branch prediction speculating the branch is taken whereas the branch is actually not taken.

By using the test case generator suggested with both a maximum inner loop bound and a maximum outer loop bound of 12, branch predictors of order up to 12 can be learned as the test case guarantees all states transitions are covered. In practice it is unlikely this number of test cases will be needed due to the complexity of branch predictors that are used. However this high number will serve to demonstrate that significant numbers of test cases can be used within a reasonable amount of computation time. Clearly the number of test cases can easily be increased, as they are automatically generated, if the learner can not make a conclusion prediction.

B. Branch Predictor Learner

The task of identifying the type of branch predictor used begins with describing the behaviour of the three types of predictor contemplated. A branch predictor is an automaton in which each state is associated with a prediction, *taken* or *not taken*, that will be made when a request is received; transitions between states are triggered (and labelled) by the subsequent notification of the actual outcome of the branching action predicted (which, again, is either *taken* or *not taken*). Such an automaton can be conveniently described as a list of 4-tuples. The first three elements of the 4-tuple list the current state, transition label, and new state. The fourth argument makes explicit the prediction associated with each state. So, for example, the two-bit predictor from Figure 1 is represented by the predicate `twoBitPredictor/4` in Figure 4.

The first two arguments are taken as input, and the latter two produced as output, but there is an important notion of time that is not obvious from this mode description: only the first argument (current state) is used to make the prediction (argument 4), which is given by the current state alone; then the actual branch taken, provided as input (argument 2), is used to determine the next state of the automaton (argument 3). For instance, a two-bit predictor in *strongly taken* (st) state will predict the branch to be taken, and if this is not the case, i.e., the branch is not taken, its state will change to *weakly taken* (wt) (see line 7 of `twoBitPredictor/4`). One-bit and zero-bit predictors can be described in the same way, the only difference being the reduced number of states, which means there is no more hysteresis in the one-bit predictor behaviour – it reverses the prediction it makes after the first mis-prediction. The zero-bit predictor has only one state, hence one prediction, which we have arbitrarily set to *taken*. The maximum range of states for all three predictors is listed in predicate `state/1`.

```
% twoBitPredictor:
%       In       In           Out          Out
%       t        t + delta    t + delta    t
% (CurrentState,BranchTaken,NextState,Prediction)

twoBitPredictor(sn,n,sn,n).
twoBitPredictor(sn,t,wn,n).
twoBitPredictor(wn,n,sn,n).
twoBitPredictor(wn,t,wt,n).
twoBitPredictor(wt,n,wn,t).
twoBitPredictor(wt,t,st,t).
twoBitPredictor(st,n,wt,t).
twoBitPredictor(st,t,st,t).

% oneBitPredictor:
% (CurrentState,BranchTaken,NextState,Prediction)

oneBitPredictor(sn,n,sn,n).
oneBitPredictor(sn,t,st,n).
oneBitPredictor(st,n,sn,t).
oneBitPredictor(st,t,st,t).

% zeroBitPredictor:
% (CurrentState,BranchTaken,NextState,Prediction)

zeroBitPredictor(st,n,st,t).
zeroBitPredictor(st,t,st,t).

% maximum range of states
state(sn).
state(wn).
state(wt).
state(st).
```

Fig. 4. Logic programming representation of predictors

The ILP learner Progol4.4 is given a choice of three background predicates, `topZeroBP/3`, `topOneBP/3`, and `topTwoBP/3` that can be used to “explain” the observations listed in `ttp/3`, that is, the number of mis-predictions for a given training case. Each of the three predicates has the same three arguments: *Case*, *Number of mis-predictions*, and *Total number of predictions made* (which, in turn, are the same as the arguments of `ttp/3`). The predicate `topTwoBP/3` is defined in such a way, so that it only succeeds if for a given case A, there is an initial state IS, such that a two-bit predictor would replicate exactly the behaviour of the predictor recorded in `tp/4`. To use the example of case 15 in Figure 3, the predictor whose model we try to unveil, has generated the following sequence of predictions, listed in argument 3 of `tp/4`: *t, t, t, t, t, t*. Only 4 of these 6 predictions have been successful, as can be seen from argument 4 of the same predicate, showing the actual program behaviour: *t, t, n, t, t, n*. Predicate `topTwoBP/3` will try each of the four possible initial states sn, wn, wt, st in turn, and make the six consecutive predictions that a two-bit predictor with this initial state would prescribe. For the predicate to succeed, the two-bit predictor has to replicate in each step the prediction listed in argument 3 of `tp/4`, while using argument 4 of the same predicate to update its state. In the case of success, the total number of mis-predictions is computed and output in argument 2 of `topTwoBP/3`.

The predicates `topOneBP/3` and `topZeroBP/3` are defined in a similar way. When presented with the data in the form of `ttp/3` and `tp/4` predicates, Progol can now learn rules of type `ttp(A,B,C) :- topTwoBP(A,B,C)`, where the predicate on the right hand side (in the body of the clause) can be one of the three described. As all examples in a data set are generated by the same predictor, we expect that Progol would learn a single rule that would cover (account for) the entire data set. This is indeed the case, as our experiments, that are described in the following section, show.

Some of the Progol code is listed in Figure 5. The `set/2` definitions specify constraints pruning the search of the hypothesis space, setting aspects of the preference bias, and specifying learning with no counter-examples. The *mode* declarations list the predicates that can appear in the head and body of a clause (RHS and LHS of a rule) learned; *type* declarations introduce user-defined types. The background knowledge shown contains the full definition of `topTwoBP/3`; those of `topOneBP/3` and `topZeroBP/3` differ only in two lines of code, replacing the two calls to `twoBitPredictor/4` with calls to `oneBitPredictor/4`, respectively `zeroBitPredictor/4`.

V. Evaluation

The purpose of this section is to show how the branch predictor learner can distinguish between different types of hardware.

A. Evaluating the Ability to Learn Different Hardware Architectures

In section IV-B the method of learning different types of branch prediction mechanisms was described. Here an evaluation is presented that illustrates their ability to distinguish between the different types of mechanisms using the test cases given in section IV-A.

Test cases were produced based on the nested loop code example given in section IV-A 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 that was to identify whether the test data was a positive match for one of the known predictor types.

For each of the test cases the type of branch predictor was correctly learned. The results are summarised in Table II. The

```
%%%%%%%%%%%%%%%%%%%%
% search parameters
%%%%%%%%%%%%%%%%%%%%
:-set(posonly)? :- set(c,1)?
:- set(h,10000)? :-set(r,100000)?
:-set(inflate,101)? :- set(nodes,1000)?

%%%%%%%%%%%%%%%%%%%%
% mode declarations
%%%%%%%%%%%%%%%%%%%%
modeh(1,ttp(+int,+int,+int))?
modeb(1,topTwoBP(+int,+int,+int))?
modeb(1,topOneBP(+int,+int,+int))?
modeb(1,topZeroBP(+int,+int,+int))?

%%%%%%%%%%%%%%%%%%%%
% type declarations
%%%%%%%%%%%%%%%%%%%%
state(sn). state(wn). state(wt). state(st).

%%%%%%%%%%%%%%%
% BK Knowledge
%%%%%%%%%%%%%%%
% topTwoBP(Case,NoMispredictions,MaxIndex)
topTwoBP(Case,NoMispredictions,MaxIndex) :-
   twoBP(Case,0,NoMispredictions,MaxIndex).

% twoBP(Case,MispredictionsSoFar,
    TotalNoMispredictions,MaxIndex)
twoBP(Case,MispredictionsSoFar,
    TotalNoMispredictions, MaxIndex) :-
    state(InitState),twoBPhelp(InitState,
    Case,1,MaxIndex,MispredictionsSoFar,
    TotalNoMispredictions).

%              IN       IN    IN     IN     IN
%       IN
% twoBPhelp(CurrState,Case,Counter,Index,Predict,
    BranchTaken)

twoBPhelp(CurrentState,Case,MaxIndex,
    MaxIndex,MispredictionsSoFar,
    TotalNoMispredictions) :-
    twoBitPredictor(CurrentState,
    ActualBranchTaken,_,Prediction),
    tp(Case,MaxIndex,Prediction,
    ActualBranchTaken),countMispredictions
    (MispredictionsSoFar,Prediction,
    ActualBranchTaken, NewNoMispredictions),
    TotalNoMispredictions = NewNoMispredictions.

twoBPhelp(CurrentState,Case,Counter,MaxIndex,
    MispredictionsSoFar,TotalNoMispredictions) :-
    twoBitPredictor(CurrentState,ActualBranchTaken,
    NextState,Prediction),
    tp(Case,Counter,Prediction,ActualBranchTaken),
    IncrementedCounter is Counter + 1,
    countMispredictions(MispredictionsSoFar,
    Prediction,ActualBranchTaken,
    NewNoMispredictions),!,twoBPhelp
    (NextState,Case,IncrementedCounter,
    MaxIndex,NewNoMispredictions,
    TotalNoMispredictions).

countMispredictions(MispredictionsSoFar,
    PredSameAsBranchTaken,PredSameAsBranchTaken,
    MispredictionsSoFar).
countMispredictions(MispredictionsSoFar,
    Prediction,ActualBranchTaken,
    NewNoMispredictions) :- \+ Prediction =
    ActualBranchTaken,
    % '\+' stands for 'not'
    NewNoMispredictions is MispredictionsSoFar + 1.
```

Fig. 5. Logic programming representation of the two-bit predictor automaton

table features three columns. The first column is the test case which is being evaluated. The second column is the number of instances of the target predicate `ttp/3` from which a rule was learned. The final column is the total execution time. Since there are more combinations of an automaton's initial state and loop bounds, the number of examples to learn from is proportional to the number of internal states of the automation at hand. The results show that more time is taken to learn lower-order predictors. The learning time is influenced by non-logical aspects of the program, such as the order in which the examples and the $modeb$ declarations are listed; both have implications on the order in which the hypothesis space is searched. The exact effect of these is difficult to predict but the results here show that a number of examples sufficient to determine the branch predictor type can be processed in tens of seconds, an acceptable time for this type of task. The results also show that the number of instances needed is relatively small showing the current test technique is sufficient. Future work could also look to optimise the way in which the learner is set.

Predictor	Number of instances	Time (s)
Two Bit	576	16.23
One Bit	288	21.69
Zero Bit	144	31.80

TABLE II
RESULTS FOR LEARNING BRANCH PREDICTOR TYPE

B. Evaluating the Ability to Learn Overall Worst-Case Mispredictions

The next stage of the evaluation is to show how the different components of our solution can be used as part of the overall WCET analysis problem. For this purpose a sort routine is used as it is a simple example to explain and yet has some interesting characteristics - principally the number of loop iterations and hence number of mis-predictions is a function of both the inner and outer loop guards. The sort routine is illustrated by the pseudo-code in Figure 6.

```
for i = 1 to A loop
  for j = 1 to A-i loop
    if a[j] $<$ a[j+1] then
      . . .
  end loop
end loop
```

Fig. 6. Sort routine

The first piece of information to be learned is the number of loop iterations for the inner loop. The method for determining this is detailed in our previous work [8] with a brief summary provided in section II-B. The desired result is in equation (1).

$$lc_{sort} = \frac{A \cdot (A-1)}{2} \qquad (1)$$

We used equation (1) to generate the data to learn from in the shape of pairs of numbers representing the upper bound A and the corresponding number of times the inner loop body is executed, B (see Table III). The variable range was set to $A \in \{1, \ldots, 30\}$. 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 III
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).
  ...
```

Using `sum` and `product` as background knowledge, that is, looking for an equation tying up the data arguments through these two operators, Progol4.4 found a one-rule model in less than four seconds (Table IV). Note C and D are intermediate variables introduced by the learner.

TABLE IV
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.

$$C = A * A \quad (2)$$
$$D = A + B \quad (3)$$
$$C = B + D \quad (4)$$

Note 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 as it can be reduced to be identical to the formula in equation (1). We then assume the methodology in section 4 has been followed, and that the result identified a two-bit predictor has been used. Finally the results from all the stages can be combined. There are three branches to be considered which are as follows:

1) *Main Body* - The **if** statement is executed $\lceil A \cdot (A-1)/2 \rceil$ times. Without knowledge of the data the worst case situation has to be considered which is an alternating case $(tn)^i$ where i is equal to $\lceil A \cdot (A-1)/4 \rceil$.
2) *Outer Loop* - The **for** statement, with loop index variable m, conforms to the pattern $(t^{j-1}n)^i$ where $i = 1$ and $j = A$.
3) *Inner Loop* - The **for** statement, with loop index variable n, conforms to the pattern $(t^{j-1}n)^i$. As the value of j associated with the inner loop is continually changing then a safe assumption must be made, i.e. $i = A$ and $j = A$.

Using the contents of Table I and the information learned so far, the results in Table V can be established for each of the branches considered, where $E = \lceil A \cdot (A-1)/4 \rceil$. Clearly where no knowledge of the initial state is available then the worst-case assumptions have to be made. For example if $A = 10$, the total worst-case predictions would be: Main Body equal to 45, Outer Loop equal to 4, and Inner Loop equal to 13. This makes a total of 62 mis-predictions. If the safe assumption related to the *Inner Loop* was addressed then at most the cost of one branch mis-prediction would be saved. This corresponds to the biggest difference in the number of mis-predictions between the last two rows in Table V. Future work could investigate further additions to the learning to address this error. It should be noted that without any form of learning then mis-predictions would have to be assumed each time a branch is executed. The Inner Loop and Main Body are executed 10^2 times, and the Outer Loop 10 times. This results in 210 mis-predictions which is significantly worse than the figure gained using the method presented in this paper.

Branch	Initial Predictor State			
	SN	WN	WT	ST
Main Body	E	E	$2 \cdot E$	E
Outer Loop;$A = 3$	4	2	1	1
Outer Loop;$A > 3$	3	2	1	1
Inner Loop;$A = 3$	$3 + A$	$1 + A$	A	A
Inner Loop;$A > 3$	$2 + A$	$1 + A$	A	A

TABLE V
NUMBER OF MIS-PREDICTIONS

VI. CONCLUSIONS

This paper has shown how machine learning can be used to identify the branch predictor used by a particular processor. This information was then combined with knowledge gained from program flow analysis and static analysis of branch mis-predictions. The resulting worst-case number of mis-predictions is a lot better than those found if computational intelligence techniques, i.e. machine learning, was not used.

ACKNOWLEDGEMENT

We thank Ralf Reutemann for the examples used.

REFERENCES

[1] I. Bate and R. Reutemann. Efficient integration of bimodal branch prediction and pipeline analysis. In *Proceedings of the 11th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications*, pages 39–44, 2005.

[2] G. Bernat, A. Colin, and S. Petters. WCET analysis of probabilistic hard real-time systems. In *Proceedings of the 23rd Real-Time Systems Symposium*, pages 279–288, 2002.

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

[4] A. Colin and I. Puaut. Worst Case Execution Time Analysis for a Processor with Branch Prediction. *Real-Time Systems Journal*, 18(2/3):249–274, 2000.

[5] L. David and I. Puaut. Static determination of probabilistic execution times. In *Proceedings of the 16th Euromicro Conference on Real-Time Systems*, pages 223–230, 2004.

[6] J. Engblom. 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*, pages 152–159, 2003.

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

[8] 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*, 2006.

[9] 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.

[10] E. Kligerman and A. Stoyenko. Real-time Euclid: a language for reliable real-time systems. *IEEE Transactions on Software Engineering*, 12(9):941–9, 1986. IEEE Transactions on Software Engineering.

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

[12] T. Mitra, A. Roychoudhury, and X. Li. Timing Analysis of Embedded Software for Speculative Processors. In *Proceedings of the 15th International Symposium on System Synthesis*, 2002.

[13] F. Mueller and J. Wegener. A comparision of static analysis and evolutionary testing for the verification of timing constraints. In *Real-Time Technology and Applications Symposium*, pages 144–154. IEEE, 1998.

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

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

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

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

[18] J. Smith. A Study of Branch Prediction Strategies. In *Proceedings of the 8th International Symposium on Computer Architecture*, pages 135–148, Minneapolis, Minnesota, USA, 1981.

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

[20] 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.

[21] J. Wegener, H. Sthamer, B. Jones, and D. Eyres. Testing real-time systems using genetic algorithms. *Software Quality Journal*, 6(2):127–135, 1997.