Probabilistic Instruction Cache Analysis using Bayesian Networks

Mark Bartlett, Iain Bate, James Cussens and Dimitar Kazakov
Department of Computer Science
University of York
York, UK
Email: firstname.lastname@cs.york.ac.uk

***Abstract*—Current approaches to instruction cache analysis for determining worst-case execution time rely on building a mathematical model of the cache that tracks its contents at all points in the program. This requires perfect knowledge of the functional behaviour of the cache and may result in extreme complexity and pessimism if many alternative paths through code sections are possible. To overcome these issues, this paper proposes a new hybrid approach in which information obtained from program traces is used to automate the construction of a model of how the cache is used. The resulting model involves the learning of a Bayesian network that predicts which instructions result in cache misses as a function of previously taken paths. The model can then be utilised to predict cache misses for previously unseen inputs and paths. The accuracy of this learned model is assessed against real benchmarks and an established statistical approach to illustrate its benefits.**

***Index Terms*—instruction cache; worst-case execution time (WCET); Bayesian network**

I. Introduction

Improvements in processors have led to a great increase in the rate at which they can execute instructions. Unfortunately, the rate at which they can be supplied with instructions from main memory has not kept pace. For this reason, modern systems (including embedded systems) commonly feature instruction caches between the processor and main memory from which instructions can be more quickly retrieved for execution. This reduces the average latency for retrieval of instructions, but introduces variability that makes reasoning about the temporal behaviour of the system much more difficult.

Knowledge of the worst-case execution time (WCET) of a task is a crucial part of scheduling any real-time system. In the general case, determining the exact WCET is impossible mathematically [1] and infeasible through exhaustive testing [2]. Therefore estimates of this quantity are needed instead.

Where failure to meet a deadline can result in catastrophic behaviour (hard real-time systems), estimates which are guaranteed to be at least as large as the real WCET are needed. Static analysis [2] is used to mathematically derive these safe estimates. Such a process will however generally introduce pessimism, as much as 33% according to a recent case study [3]. For applications where occasional deadline misses are not so critical (soft real-time systems), having to provide so much additional unused processing power may be unnecessarily expensive. Rather, testing may be used to provide an estimate of the WCET. Any non-exhaustive testing approach cannot be guaranteed to overestimate the real WCET but is likely to be less pessimistic.

Estimating WCET using static analysis presents particular problems in systems with instruction caches. Instructions found to be in cache when required will be retrieved much faster than those not in cache, perhaps by orders of magnitude. Therefore any instruction that is pessimistically assumed to be out of cache, when in fact it may not be, will result in a large amount of pessimism being added to the estimate.

This paper therefore presents an alternative approach to cache analysis based not on creating a cache model through deduction from the specification of the hardware, but rather from induction from observations of the behaviour of the cache in use [4]. The core aspect of this approach is to derive information on the behaviour of the cache through logging execution traces and then to automatically generalise this to a model which can be used to predict cache hits and misses for any path. In doing so, we seek to avoid any problems due to lack of knowledge of the hardware or incompleteness in reasoning about the possible cache state. As with any measurement based approach, however, the correctness and safety of the model generated cannot be guaranteed. It is therefore only suited to soft real-time system applications.

The model built by the approach is based on a Bayesian network [5]. This records the conditional dependencies between the caching behaviour of each instruction of interest and other instructions in the program. A process of learning is used to automatically select the network which best explains the behaviour of the cache seen in the program traces gathered. This model differs in two main ways from a traditional cache model. First, the model is probabilistic. Rather than just predict either a cache hit or miss, it assigns a probability based on the behaviour of other instructions in the program. Such a method allows the use of all information available, even when it is clear that insufficient information is known to correctly classify the instruction with certainty, allows smooth degradation of quality when not completely correct, and gives a primitive measure of the confidence with which the label is assigned. Second, the model is of the use of the cache by a particular program, whereas a traditional cache model is of just the cache itself.

The specific contributions of this paper are twofold. First, a

TABLE I
LABELS FOR CACHING BEHAVIOUR OF AN INSTRUCTION.

Label	Meaning
Always Hit	The instruction is always in the cache when needed.
Always Miss	The instruction is never in the cache when needed.
First Hit	The instruction is in the cache when first needed, but never when needed after that.
First Miss	The instruction is not in the cache when first needed, but always when needed after that.
Conflict	The instruction cannot be statically determined to have any of the other labels.

previously published version of the technique [6] is modified to allow it to be used in WCET analysis. The earlier model mixed the prediction of control flow and cache analysis. We show here how the learning and utilisation of the model must be altered to allow control flow analysis to be performed external to the model, with the model being used to predict cache behaviour for a given program path. This allows standard static control flow analysis to be performed and the cache model to be incorporated as part of the WCET analysis in a similar way to normal static cache analysis. Second, a thorough evaluation of the technique is presented with regards to establishing its accuracy. This examines in particular the size of dataset needed to create a good model. A comparison is also performed with another measurement-based approach to cache prediction based on extreme value statistics, which is an adaption of a previously published method [7].

The rest of this paper proceeds as follows. Section II presents previous related work on cache analysis and Bayesian networks. Section III presents details on the process of acquiring data and learning the cache model from these, before Section IV presents results from the application of this technique. Finally, Section V concludes.

II. BACKGROUND AND RELATED WORK

A. Cache Analysis

Static cache analysis for real-time systems was first proposed by Mueller [8] and much modern analysis has extended this technique [9]. At the heart of this approach is the use of data flow analysis to determine the state of the cache before each execution. As influential as this technique, has been the idea of labelling each instruction to summarise its behaviour on repeated executions. Having determined the cache state before each instruction, a label is then applied to the instruction to categorise it. The classifications possible are shown in Table I.

An alternative approach is to use abstract interpretation to determine the cache contents [10]. This approach conducts *must* analysis (to see what will always be in cache when needed), *may* analysis (to find what could be in cache) and *persistence* analysis (to see what will not be evicted once loaded). Based on these analyses, a label from Mueller's classification is again assigned to each instruction.

The core idea in all of these previous approaches to cache analysis is to construct a model of the way in which the cache operates and then track, explicitly or implicitly, the contents of the cache before and after each instruction. This requires a full understanding of the operation of the cache and sophisticated human intelligence to devise the techniques used for analysis. As a result, good analysis of caches in more complex systems (such as multicores, MPSoCs and cache hierarchies) constantly lags behind the state-of-the-art systems. Our aim is to devise a more automated approach to cache analysis which functions through a pragmatic analysis of the cache in operation. Hopefully such a technique would be more robust to changes in the underlying hardware. An additional perceived advantage of such a technique is less pessimistic estimates of WCET for those domains where optimistic values can be occasionally tolerated.

In addition to work on analysing caches for WCET, there is a related body of work on making caches more temporally predictable for simpler analysis, for example by locking their contents [11]. The work presented in this paper relates to analysing systems as they are, not modifying the hardware or software. Such ideas are therefore not pursued or commented on further here.

B. Bayesian Networks

A Bayesian network is a way of representing a joint probability distribution over a set of random variables V. The structure of the network is a directed acyclic graph (DAG) consisting of a set of nodes, one corresponding to each variable, and a set of directed edges between these nodes. An edge in the network from V_1 to V_2 indicates that the value of variable V_1 may influence the probability distribution of V_2. In other words, the probability distribution of the variable associated with each node is conditionally dependent on the variables associated with each of its parents. The exact nature of this conditional dependence is recorded in a probability function of the values of the node's parents. In the case of discrete random variables (as will be used throughout this paper) the probability function maps each permutation of a parent set's values onto a distribution over the allowable values of the variable.

For processes that are understood, it is possible to find the structure of Bayesian networks analytically. However, in some situations, the relationship between variables is not known *a priori*. In this case, it is possible to use machine learning to generate a network structure from a set of observations of the variables. This process searches for the network whose structure best explains the observed data. It can be shown (assuming there is no prior reason to prefer one network structure over another) that this is the network for which the following quantity is maximised [12].

$$\prod_{i=1}^{n}\prod_{j=1}^{q_i}\frac{\Gamma(N'_{ij})}{\Gamma(N'_{ij}+N_{ij})}\cdot\prod_{k=1}^{r_i}\frac{\Gamma(N'_{ijk}+N_{ijk})}{\Gamma(N'_{ijk})} \qquad (1)$$

where there are n variables, q_i is the number of combination of values for the candidate parent nodes of the ith variable, r_i is the number of possible values for the ith variable, $\Gamma(\cdot)$ is the

gamma function (an extension of the factorial function to real numbers), N_{ijk} is the number of observations of that particular combination of values in the dataset, $N_{jk} = \sum_{i=1}^{n} N_{ijk}$, N'_{ijk} is a constant commonly set to $\frac{1}{q_i r_i}$, and $N'_{jk} = \sum_{i=1}^{n} N'_{ijk}$.

Intuitively, the first Π is a product over each of the nodes in the network, the second Π is a product over all possible tuples of values that the parents of that node can take, and the final Π is a product over all values that the node itself can take. In the current paper, the number of nodes is fixed, as are the set of values taken by the node. Hence the only way in which the quantity can vary in order to be maximised is through changing the set of parents of a node. This leads to the number of possible tuples of values, q_i, changing (as this will be equal to the number of possible values of a node to the power of the number of parents) and also to different values of N_{ijk} from the dataset being used. In short, the task of maximising the equation becomes one of searching for the best possible set of parents for each node.

Once a network structure has been determined, it is trivial to produce the corresponding conditional probability table for the node. Each entry in the conditional probability table is set to the number of times that the particular combination of parents‘ and node values were observed together in the dataset divided by the total number of times that the same combination of parents‘ values were observed, i.e. the empirical relative frequency of that value being observed given the values of the parents.

C. Bayesian Networks for WCET

As far as we are aware only one technique, other than that modified and extended for this paper, has been previously proposed using Bayesian networks for WCET analysis [13]. In the presented technique, Bayesian networks are used to learn and record the interaction of the execution time of various parts of a program in relation to each other. The output is then presented to the system designer who can use it to aid construction of the system. The approach differs considerably to that presented here, most notably in recording timing interactions at a much higher level. In the technique outlined here, such interactions will be at the single instruction level. Additionally, the previous technique records timing interactions due to any hardware component as well as software dependencies. The technique to be presented focuses more tightly on cache analysis, allowing other possible sources of timing dependency to be treated using separate established software and hardware analysis methods.

III. A Bayesian Network Cache Model

The technique presented in this paper is a modified version of the cache model proposed in [6]. However, as presented, that technique allowed only for the distribution of cache hits or misses on any unknown execution of the program to be determined. Utilising cache analysis for WCET estimation however requires something different; the instructions which result in cache hits or misses must be identified, not just the total number of these events. This paper shows how this can be accomplished by modifying the learning process used to create the cache use model.

To make this paper self-contained, and as the technique in [6] has not previously been published in the Real-Time Systems community, we first outline that technique here. The problems with this are then made explicit before the changes necessary are presented.

A. Overview of the Technique

The basic model will now first be explained at an intuitive level before going on to explore in greater depth the various aspects in the following sections.

The model constructed is a Bayesian network in which nodes represent instructions of the program under consideration and the edges between these nodes represent an interaction between the behaviour of the two instructions. That is to say, an edge from instruction i to instruction j records that when instruction j is required, whether or not instruction i was in cache when required (or not executed at all) will have a bearing on the likelihood that instruction j is in cache. This relationship may be due to instructions i and j being in the same cache block or due to i and j being loaded into the same cache line. In fact, as the network will be learned from observations, it is not necessary to determine why the behaviour of one instruction affects another, merely that there is a statistical correlation observed between the two behaviours. The exact nature of the relationships between the instructions linked by edges is recorded in the conditional probability table at each node.

Obviously, a cache is a fully deterministic hardware device, hence the choice of a probabilistic representation for the model may appear unusual. However, such a representation allows a model to be constructed when the interaction between instructions is partially but not fully understood. The conditional probability tables in the Bayesian network do, of course, allow for fully deterministic relationships to be described where the data support this. In addition, a Bayesian framework allows the selection of the provably best model of the cache given the observed data; without any reason to suspect that one model of the cache is more likely than another, the Bayesian model of the cache is certainly the best that could be determined from the available data.

B. Data Collection

The cache model is learned from the observation of the cache in use on a particular program. It is therefore first necessary to obtain these observations.

The required information for each observation is a trace of the execution of the program featuring the instructions fetched in order and whether each was in cache when required, or had instead to be first retrieved from main memory. Multiple runs are obtained by altering the input vectors given to the program under examination.

The methods for and issues surrounding collection of data for real-time systems are very rehearsed and need not be explored in depth here [14].

C. Processing Observations

Bayesian networks are learned from a rectangular array of data in which columns represent the variables that will form nodes in the network and each row records the outcome of a single associated observation of each of these variables. For the cache model application presented here, this means that each row must be the outcome of a single run of the code for a random input and each column is the cache behaviour of a single fetch of a particular instruction. The data values in the table then state what the observed cache behaviour of that instruction was on that run.

There are two issues to deal with in order to construct this table. First, the matter of instructions within (nested) loops, which may be fetched multiple times within a single execution; the table can record only a single value for each variable, not a list of values for all the fetches. Second, the subject of missing values; some instructions will appear in one execution of a program but not in others, meaning that for some executions there is no observation of whether that instruction was in cache when fetched because it was never fetched.

Instructions which may execute multiple times are treated by separating them into multiple variables, one for each execution of the loop. For example, instruction I is split into I_1, I_2, I_3, etc. denoting the cache behaviour of the instruction on the first iteration of the containing loop, on the second iteration, on the third and so on. This generalises to nested loops by introducing a subscript for each of the containing loops. The control flow graph can be used to determine appropriate variable assignments for each recorded instruction execution in a program trace. These variables are then assigned the value of HIT or MISS for each run, depending on whether the instruction was fetched from cache or main memory respectively.

Those variables so far described relate to the instructions' behaviour on the nth iteration. It can be observed that some cache interactions between instructions are likely to be related to what occurred on the final iteration of a loop, where this may occur on different iterations in different traces. Variables are therefore introduced for each instruction on the last iteration, the second to last iteration, the third to last, etc. Let the variable I_{-n} refer to the behaviour of instruction I on the nth from last iteration of the loop. Again this notation is extended to multiple subscripts for nested loops. These variables are given values exactly as with the other variables.

The solution to the problem of missing values mentioned above is to allow variables to take two additional values. In addition to HIT and MISS, variables also take the values of NO_ITERATION if they were not encountered because the loop they were in never had that many iterations on that run of the program, and NOT_EXECUTED is assigned to variables whose loop was encountered, but which were not executed on that iteration. In addition to filling in the blanks in the table, these instructions allow for an important link between the control-flow behaviour of the program and the caching behaviour. The reason an instruction is or is not in cache when required may be linked to the path along which the instruction has been reached; these values allow for such relationships to be expressed.

Using all of these variables, each trace can thus be transformed into a mapping of a global set of variables to a particular set of values that fully represents the caching behaviour of that execution. From this dataset, a Bayesian network can then be learned.

D. Learning the Bayesian Network

In the dataset obtained as described above, some variables will take the same value in every single observation. These variables therefore have no predictive power in determining the value of other variables nor are affected by the other variables either. Such variables have their constant values noted and are pruned from the dataset used for learning. When the value of such variables is needed for the subsequent cache analysis, the value seen in all observation is used directly. Note that it cannot be confirmed that the same value would be seen on all possible executions of the program, only those that have been obtained. There is however no way of determining what may occur on any other executions, hence the values must be treated as constants. This is a similar idea to the concept of dynamic invariant detection [15].

Having reduced the dataset to just those instructions that have different values in different observations, a Bayesian network which seeks to explain the causal relationships between these variables can be learned. Intuitively, for each variable, we seek to find the variables whose value can affect the value of that variable. This is performed by exhaustively searching through the power set of all other variables, attempting to maximise a function of the likelihood that those parents are the variables true parents. The function used for this is that previously given in Equation (1).

As explained in Section II-B, the appropriate conditional probability tables for each node can then be found through the relative frequency of events in the observed dataset.

Those variables which have negative indices are treated differently from those explained above. Recall that these are defined as the occurrence counting backwards from the last iteration. The associated nodes can therefore be created analytically. For each such node, the parent nodes are set to all those nodes that could refer to the same instruction when counting from the first iteration. For example, I_{-1} will have as parents I_1, I_2, I_3, $\ldots I_n$ where n is the largest number of iterations seen. The value taken by such a node can then also be set deterministically. For example, in the previous example, I_{-1} will have its value defined to always be equal to the I_i with the highest index and which has a value of HIT or MISS.

The final cache model therefore consists of two parts, a list of instructions-behaviour pairs for those instructions whose caching behaviour was always observed to be constant and a Bayesian network which describes causal relationships for those instructions exhibiting different behaviours in different executions of the program.

E. Incorporating Domain Specific Knowledge

The process of learning the optimal Bayesian network for a particular dataset is NP-hard in the number of variables in the general case [12]. As even simple programs typically consist of thousands of instructions and may iterate over these thousands of times, the number of variables to consider for learning the Bayesian network may be infeasibly large. For this reason, it is necessary to use some simple domain-specific observations to reduce the task to a manageable level.

It makes sense that instructions can only be affected by those that may have happened earlier in time; whether the second instruction to be fetched was in cache cannot affect whether the first instruction fetched was there, for example. Using the control flow graph, we can construct a partial ordering over the order in which instructions can execute. From this, a restricted set of instructions can be found which can occur before each instruction. The set of parents to consider for each node when learning the Bayesian network can then be restricted to this set. Particularly for instructions late in the program, this reduces the set of parents to consider hugely.

A second similar restriction can also be introduced. Caches are more likely to contain instructions that have been recently executed. The search for the best set of parents for a node can therefore be restricted to those that could have been executed within a certain period before the current instruction. As with the above refinement, this set of potential parents can be found through the same partial ordering and control flow graph. However, in contrast to the previous refinement. such an alteration may decrease the quality of the network learned. More recent instructions are only more likely to affect an instruction, but it is possible that instructions from much earlier in a program may still have an effect. Reducing the number of parents to consider in this way is therefore a trade-off between the tractability of the learning and the quality of the outcome.

F. Problems with the Existing Approach

The process outlined above produces a model from which a predicted distribution of the number of cache hits and misses for a particular program can be obtained. This can be constructed through simple Monte Carlo sampling of the nodes of the Bayesian network subject to the associated conditional probabilities. An estimate of the worst-case value can then be found from the tail of this distribution. However, such estimates cannot be incorporated into standard WCET analysis in the same way that the results of a static cache analysis can for two reasons.

First, the instructions which cause cache hits and misses must be known, not just the total number of these that occur. For an extremely simple processor, it would be possible to find the WCET in the absence of a cache and just add on the number of cache misses multiplied by the additional time incurred by fetching from main memory as opposed to the cache. However, in the presence of a pipeline within the processor, this becomes unsuitable; the overall temporal effect of a cache miss at one program point may have a completely different effect to that at another due to the instructions transversing the pipeline.

Second, the existing technique conflates parts of the control flow analysis and low-level analysis. While extracting the number of cache misses from the Bayesian network, the probability that given instructions execute are implicitly used through the conditional probabilities assigned to the NOT_EXECUTED or NO_ITERATION values. Determining whether or not instructions execute is part of the calculation and should not also occur using an alternative method during the cache analysis. Instead a model is required that predicts whether or not an instruction is in cache given that it executes. Determination of what does execute can then be correctly restored to the control flow and calculation phases.

A change to the Bayesian network learning algorithm allows this distinction to be made as outlined below.

G. Modification of the Approach

The network will ultimately be utilised to predict whether instructions are cache hits or misses given that they execute. It will not be used to predict whether instructions are executed or not, this information is best determined using existing techniques at other stages of the WCET analysis and will be supplied as input to the network. This means that the network should never predict a NO_ITERATION or NOT_EXECUTED value for a variable; if the variable is to take this value for a particular execution, it will be determined elsewhere and fixed in the network at analysis time.

One consequence of this is that only variables whose behaviour has been observed to be both HIT and MISS in the dataset will need to have their behaviour explained by other variables. An instruction that is always either NO_ITERATION or HIT, for example, can have its behaviour on a particular run determined just by knowing whether it was executed or not. There is therefore no need to learn parent nodes in the Bayesian network for such a variable.

Using this observation, the dataset (obtained through the same observation and processing as before) is split into three parts; those instructions which exhibit constant behaviour, those which exhibit constant behaviour if they are executed and those whose caching behaviour may vary. The first of these are recorded in a list, and will predict that behaviour whenever that instruction is seen as before. The second are also recorded and will predict the given cache behaviour if that variable is actually executed, but are also used as potential parents in the Bayesian network learning process though will not have parents learned for themselves. The final set of instructions are entered into the Bayesian network learning as both nodes whose parents must be learned and which may be used as parents of other nodes in turn.

The Bayesian network explaining the causal relationships between the caching behaviour of instructions can then be learned from the appropriate variables. Intuitively, the aim is to discover for each instruction in the final set mentioned above, the set of instructions whose behaviour may influence

whether or not that instruction is a cache hit or miss. This is done by maximising a slightly modified quantity related to that given in Equation 1. As just stated, a network is desired in which variables cannot predict that they are executed or not, but can make use of the fact other instructions have or have not executed in predicting their own caching behaviour. For this reason, the best network is the one that maximises the following quantity.

$$\prod_{i=1}^{n}\prod_{j=1}^{q_i}\frac{\Gamma(N'_{ij})}{\Gamma(N'_{ij}+N_{ij})}\cdot\prod_{k=1}^{r'_i}\frac{\Gamma(N'_{ijk}+N_{ijk})}{\Gamma(N'_{ijk})} \qquad (2)$$

where r'_i is the set {HIT,MISS} (i.e. only the possible values of i when i is known to be executed) and all other symbols are as defined as for Equation (1).

Once the structure of the network has been determined, the conditional probability tables associated with each node can be computed. In each case, the functions are set directly from the relative frequencies of the variable-value pairs observed in the dataset. Specifically, for each assignment of values to its parents, the probability assigned to a variable being a HIT is equal to the number of times it was observed as being a HIT in the dataset for that combination of parent values, divided by the number of times it was observed to be either a HIT or MISS for those parent values. The probability of a MISS is always assigned to be 1 – p(HIT).

The result of this modified learning process is a Bayesian network in which nodes will only ever predict that their associated variable is a cache hit or miss. As desired, it has become impossible for the model to predict an instruction will not execute. Such information can be supplied to the model when it is used to predict what is in the cache, by fixing nodes to either of the unexecuted values. This information on whether other instructions have been executed will then still be usable in predicting caching behaviour of other instructions as before. Details of how the model is used for cache analysis can be found in Section IV.

H. Scalability

Recall that the task of learning the Bayesian network consists primarily of determining the structure of the network and that this occurs through finding the set of parents of each node in turn. As stated above, the search for parents will only consider other nodes which may have occurred within a set limit before the node in question. Therefore, on average, the number of parents to be considered for each node will be approximately constant regardless of the position in the program that the node corresponds to (assuming that there is no reason to believe the branching of code should change in any systematic way with code length). As the parents of each node are found independently for each node, the complexity of the learning is therefore linear in the number of nodes.

The number of nodes is itself based primarily on the length of the code and the number of repetitions of loops. Under (the possibly incorrect) assumption that code length is not generally correlated with the number of repetitions of the loops within it, this leads to an approximately polynomial complexity in the length of the program code to the power of the typical loop nesting depth of instructions. In typical real-time code, this probably implies complexity of between linear and cubic in the length of the program code.

As the branching factor and depth of loop nesting are related closely to the task of a particular piece of code, it is difficult to make a useful, more specific estimate of the scalability. It follows however that the technique is obviously at its fastest and most useful when analysing code that is short or has low branching factor or shallow loop nesting.

In practice, the total number of nodes to be learned is far fewer than might be expected as a great number exhibit constant behaviour and can therefore be removed from consideration in constructing the Bayesian network. All instructions which are at neither the beginning of a cache line or basic block will be found to have consistent behaviour for any input for example. Results in the following chapter demonstrate this factor.

TABLE II
SUMMARY OF BENCHMARK PROGRAMS USED.

Name	Description	Inputs
binary_search	Binary search in a set list of numbers	One integer in the range 0–20
bsort	Bubble sort an input vector	Between 10-20 integers in the range 0–10
factorial	Find the factorial of the input	One integer in the range 0–20
isort	Insertion sort an input vector	Between 10-20 integers in the range 0–10
janne_complex	A program with complex looping	Two integers in the range 0–30
petri	Simulation of a petri net	One integer in the range 0–200
select	Select the nth largest number	One integer in the range 0–20

IV. RESULTS AND EVALUATION

A. Datasets

Evaluation of the technique is not straightforward. The most direct method of evaluation would be to compare the model with an idealised Bayesian network for the cache and program to see how well they compared. However, it is extremely difficult to create such a perfect network by hand, and if it were possible to create it automatically, then this learning technique would be completely unnecessary. Such an evaluation is therefore only possible for a small toy example [4].

Instead, the quality of the model must be assessed indirectly by looking at the predictions it makes. To this end, a traditional machine learning approach was followed. Two datasets are independently generated. The first, the training dataset, is used to construct the model. Predictions from the learned model are then compared to the second, the test dataset, to assess the quality of the learning. In fact, a range of training and test datasets are generated as described below.

A selection of seven benchmarks from the Mälardalen

TABLE III
MEAN OF THE SQUARED ERROR BETWEEN THE ACTUAL NUMBER OF CACHE MISSES FOR EXECUTIONS IN THE TEST DATASET AND THE PREDICTED VALUES

Benchmark	Training Set Size	
	100	1000
binary_search	0.045	0.047
bsort	4.647	1.419
factorial	—	1.018
isort	9.559	4.370
janne_complex	10.075	7.623
petri	0.000	0.000
select	0.041	0.040

TABLE IV
EARTH-MOVER'S DISTANCE BETWEEN THE DISTRIBUTION OF CACHE MISSES IN THE TEST DATASET AND THE PREDICTED DISTRIBUTION USING THE BAYESIAN NETWORK.

Benchmark	Training Set Size	
	100	1000
binary_search	0.002	0.001
bsort	0.016	0.013
factorial	—	0.023
isort	0.026	0.024
janne_complex	0.032	0.034
petri	0.000	0.000
select	0.001	0.001

benchmark suite[1] were chosen to evaluate. These have been adapted to take input values in order to make them amenable to measurement-based analysis. The benchmarks chosen and the inputs they take are summarised in Table II.

For each benchmark, three datasets were created; a training set of 100 examples. a training set of 1000 examples and a test set of 1000 examples. Each example in these datsets consists of a single trace of the program being executed with inputs uniformly randomly chosen from the given ranges. For some of the benchmark marks with smaller input ranges, this means that most, if not all, possible inputs were represented in the datasets; for the sorting routines, only a very tiny portion of possible inputs were seen.

In each case, the same simulation platform was used. The M5 simulator was used in its default setting for the ALPHA processor, with a single instruction cache configured to have a total size of 4kB, block size of 32 bytes and 2-way associativity. The cache replacement algorithm used was Least-Recently Used. The hardware was reset to its initial state between runs to ensure independence.

B. Evaluation

The task of the model is to predict which instructions are or are not in cache when required. It is not the task of the model to predict whether or not they are required; that is a control flow task which would be dealt with at a separate part of the WCET analysis. Furthermore, we seek accuracy of prediction, not behaviour in the worst case; the task of cache analysis is to produce as accurate information as possible on the cache behaviour which can then be utilised to produce an estimate of worst-case execution time of the program. The evaluation, therefore, is carried out as follows. First, the cache use model is learned, as described in Section III, from a given training dataset. Following this, the ability to predict the cache behaviour in each trace in the corresponding test datasets is assessed. To do this, the execution path information is extracted from that trace (which instructions are or are not executed) and supplied to the Bayesian network which fixes any instructions which did not execute at their appropriate value (NOT_EXECUTED or NO_ITERATION). Finally, the instructions that were executed have their cache behaviour predicted; from the previously observed behaviour in the case of those instructions believed to have constant behaviour, and through sampling the Bayesian network with a Monte Carlo method for those instructions featuring in the network to yield a probability of each instruction being in cache. This yields a predicted probability for each individual instruction executed of it being in cache when required. Combining these probabilities can then produce a probability distribution over the number of predicted cache misses for a trace with the same execution path as the given test trace.

The overall result of the testing is a set of pairs, one for each test example. First, the actual number of cache misses observed in that trace. Second, the model's prediction of the probability distribution of the number of cache misses for a trace with that execution path.

The actual number of cache misses for each trace can be compared to the expected (i.e. mean) value of the corresponding predicted probability distribution. This is shown in Table III. Values in this table show the mean squared difference between the real and predicted expected number of cache misses in the dataset. The smaller this value, the more precisely the predictions match the real data. Formally, the quantity shown is

$$\frac{\sum_{dataset} (actual - expected)^2}{|dataset|} \tag{3}$$

Results were obtained for both two training set sizes and also for several different degrees of sampling the Bayesian network. The greater number of samples taken, the closer the distribution obtains corresponds to the real joint distribution recorded by the network.

As can be seen, the values are generally low, showing high correspondence between the model's predictions and the real values. It should be noted that the values shown in the table are squares of the actual discrepancies,so a value of 9, say, could result from only 3 misclassifications of an instruction's cache behaviour in each entire example execution. Given the number of instructions in the programs, these values are incredibly small.

The table shows that an increased training set size improves the quality of the models predictions, particularly for those examples with the highest divergence from the actual values. There is also a slight improvement in the quality of the predictions when a greater number of samples are taken, but this is not particularly dramatic.

[1] http://www.mrtc.mdh.se/projects/wcet/benchmarks.html

The missing values for factorial when trained with 100 examples in this table (and subsequent ones) are due to a problem in variable assignments. In the test set, there are instructions that were not encountered at all in that training set. This means that the algorithm for converting traces into variables, which was outlined in Section III, is unable to correctly associate variables in the test set with the ones used in the model in order to present them for classification. This issue illustrates the fact that instruction coverage must be achieved by the method in the training set. The question of whether random inputs or a more targetted approach to training set generation would be preferable remains open for future work.

The comparison presented above does not take into account any part of the distributions of the predictions other than their expectation. It is possible however to conduct a second form of evaluation which examines the entire distribution. Rather than compare each test example separately, all the test values can be collated into a distribution over the number of cache misses in the test set and all the predicted distributions can be combined into a single predicted distribution similarly. These two distributions can then be compared for similarity using various measures. Table IV shows a comparison between these two distributions using the well-known Earth Mover's Distance. Values have been normalised by dividing by the maximum distance theoretically possible for each example, in order to allow for comparisons between different benchmarks. A value of 0 shows identical distributions; a value of 1 maximally dissimilar distributions.

The table shows a very similar pattern to that in Table III. This suggests that not only is the expected value of the number of cache misses that the model predicts close to the real value, but that the complete distribution of the values is also close to the true distribution.

C. Extreme Value Statistics

Edgar [7] presents a method that can be used to estimate the WCET of a program given a set of observations of the actual execution times. While that presented method was applied to the full execution time of a program, there is no reason that it cannot be used, almost unmodified, to estimate the worst-case number of cache misses instead. We do so here to compare with the Bayesian network approach.

From a set of observations of the number of cache misses in a series of traces, the probability that the real worst-case value is less than x is equal to the following quantity [16].

$$\frac{G(x - max) - G(max)}{1 - G(max)} \tag{4}$$

where G is a Gumbel distribution with shape parameter σ (the sample standard deviation $\times \frac{\sqrt{6}}{\pi}$) and location parameter μ (the sample mean – Euler's constant $\times \sigma$) and max is the maximum value observed in the sample.

While such a method is potentially useful for estimating the actual worst-case number of cache misses, it cannot be used in the as many situations as the Bayesian network model can. The output of this statistical method is an estimate over all cache misses in the program. While this may be used in a simple system, more complex hardware requires more detailed information as explained earlier.

For the above reason, it is not possible to use extreme value statistics to compare with the results presented in Tables III and IV however a separate form of evaluation can be performed for which the two can be compared.

In soft real-time systems, deadline misses can be tolerated as long as they do not happen too often. The concept of probabilistic hard real-time systems has also been formulated [17]. These systems must meet their deadlines at least a specific proportion of the time. In both these settings, it may be possible, or even desirable, to determine a WCET estimate that is not actually the worst-case, but rather will not be exceeded too frequently. Such an estimate may allow slower, cheaper hardware to be utilised. In such a case, rather than the actual worst-case behaviour, the behaviour at a given high percentile of the distribution of the behaviour is required. For the current application, this means determining a number of cache misses such that there are a greater number of cache misses observed in only a small number of cases.

The Bayesian network model can be used to estimate such percentile levels. The distribution over the number of cache misses which was obtained above (for comparison using the Earth Mover's Distance) can have given percentile levels found. The method using the extreme value distribution distribution presented above can also be used to estimate these values. In this case, the curve can be fitted to the same training data set as the Bayesian network is learned from and the results compared. Finally, as the real values are unknown, the actual numbers of cache misses in the test dataset can be used as a proxy for the population distribution.

Table V shows that in a few cases, the Bayesian network model exactly matches the 'true' behaviour and there are two cases where the Bayesian network produces an estimate above the actual values and one case where the extreme value statistic method does likewise. In most cases however, the two methods both produce slightly higher estimates than the observed values.

The Bayesian network produces lower estimates than the extreme value method in nine cases, and higher estimates in only four. As the real values are not known however, it is impossible to know whether these are tighter estimates or too optimistic. Nevertheless, in all but two cases, the lower estimates produced by the Bayesian method equal or exceed the ones obtained from the real data.

The most striking difference between the two methods is for the petri example. Here, the extreme value method produces a huge estimate for the number of misses, much higher than either the real test dataset or the Bayesian network approach. In fact, as seen in the earlier tables, the Bayesian network approach produces exactly correct predictions for this benchmark. The problem for the extreme value method appears to be that the number of cache misses is extremely variable, hence high upper bounds are produced. Within this

TABLE V
UPPER PERCENTILES ON THE DISTRIBUTION OF NUMBER OF CACHE MISSES IN THE TEST DATASET OBTAINED UNDER VARIOUS METHODS. ACTUAL ARE THE REAL VALUES FROM THE TEST DATASET, EVD ARE THE VALUES OBTAINED USING THE EXTREME-VALUE DISTRIBUTION, AND BN ARE OBTAINED USING THE BAYESIAN NETWORK APPROACH.

Benchmark	Method	Training Set Size	Percentile Level					
			0.9	0.99	0.999	0.9999	0.99999	0.999999
binary_search	actual	—	46	46	46	46	46	46
	evd	100	48	49	50	51	52	53
		1000	48	49	50	51	52	53
	bn	100	46	47	47	47	47	47
		1000	46	46	46	46	46	46
bsort	actual	—	56	56	56	56	56	56
	evd	100	59	61	63	65	67	69
		1000	59	61	63	65	67	69
	bn	100	60	65	73	75	75	75
		1000	58	60	62	64	66	66
factorial	actual	—	42	42	44	44	44	44
	evd	100	42	42	42	42	42	42
		1000	45	46	47	48	49	50
	bn	100	—	—	—	—	—	—
		1000	43	43	44	44	44	44
isort	actual	—	55	55	55	55	55	55
	evd	100	58	60	62	64	66	68
		1000	58	60	62	64	66	68
	bn	100	59	66	78	82	85	88
		1000	57	60	63	65	67	72
janne_complex	actual	—	46	46	46	46	46	46
	evd	100	49	51	53	55	57	59
		1000	48	50	52	53	55	56
	bn	100	46	56	63	63	63	63
		1000	50	53	54	54	54	54
petri	actual	—	6169	6491	6491	6491	6491	6491
	evd	100	11190	15704	20202	24698	29194	33690
		1000	11177	15710	20230	24749	29267	33785
	bn	100	6169	6491	6491	6491	6491	6491
		1000	6169	6491	6491	6491	6491	6491
select	actual	—	82	84	84	84	84	84
	evd	100	90	96	101	107	112	117
		1000	90	95	100	106	111	116
	bn	100	79	80	81	81	81	81
		1000	79	80	81	81	81	81

variability though, the cache state of individual instructions is actually extremely consistent, the variability coming in the number of iterations that occur. For the Bayesian network method, this regularity can be recorded and utilised; for the extreme value method that has no concept of where misses occur in the program, the information is not usable.

The effect of the dataset size on the extreme value method is virtually nil. In three cases the larger training dataset results in no changes, three times in a very marginal tightening of the estimate and once in an increase in the estimates when the smaller sample was actually optimistic. In contrast, the larger dataset sizes make quite significant changes to the Bayesian network method in several cases.

D. Detailed Analysis

A further evaluation was performed to determine the role of training set size and to understand the accuracy of various parts of the model derived. The technique described above was applied to the analysis of a single program, the bubblesort routine. This benchmark was chosen as interesting cache behaviour was seen to occur, related to the section of code which swaps a pair of adjacent numbers in an early trial. At each comparison, whether this swap occurs affects the subsequent cache contents.

Evaluation was performed as follows. First, the training data set was used to identify instructions which appeared to exhibit fixed behaviour and those for which different traces showed different cache outcomes. A Bayesian network was learned for the latter as described above. Following this, the test data set was used to determine the accuracy of the technique at predicting the number of cache misses that would occur in each of the (previously unseen) test traces. The variables in the test trace were split into three classes depending on their behaviour in the training set and evaluated as follows.

Variables not seen during training.
: For any variables not seen during training, we make an assumption that the variable would cause a cache miss (which may not actually be safe due to timing anomalies). Whether it actually had caused a miss was also noted.

Variables which exhibited fixed behaviour.
: For each variable, whether it was believed to be a cache miss, based on the training set, was noted along with whether it actually was a cache miss.

TABLE VI

COMPARISON OF NUMBER OF ACTUAL AND PREDICTED CACHE MISSES IN THE TEST DATA SET. COLUMNS SHOW THE NUMBER OF MISSES RESULTING FROM VARIABLES NOT SEEN IN THE TRAINING DATA; INSTRUCTIONS BELIEVED TO EXHIBIT FIXED BEHAVIOUR; INSTRUCTIONS KNOWN TO HAVE VARIABLE BEHAVIOUR.

Number of Training Examples	Previously Unseen		Believed To Be Fixed		Known To Be Variable		Total		Pessimism
	Actual	Predicted	Actual	Predicted	Actual	Predicted	Actual	Predicted	
100	0	4	5237	5259	309	314	5546	5577	0.56%
200	0	4	5233	5245	313	324	5546	5573	0.50%
300	0	0	5231	5231	315	333	5546	5564	0.32%
400	0	0	5231	5231	315	335	5546	5566	0.37%
500	0	0	5231	5231	315	334	5546	5565	0.35%
600	0	0	5231	5231	315	330	5546	5561	0.27%
700	0	0	5231	5231	315	332	5546	5563	0.31%
800	0	0	5231	5231	315	330	5546	5561	0.27%
900	0	0	5231	5231	315	328	5546	5559	0.23%
1000	0	0	5231	5231	315	328	5546	5559	0.23%

Variables which appeared in the Bayesian network
: A probability distribution over the number of cache misses was found as in the earlier experiments. The actual observed number of misses for these variables in the traces was collected as well.

Having obtained these data, the total number of cache misses that occurred in the whole of the test data set and the total number that were predicted to have occurred were found by summing over the data set. The distributions that occurred from the Bayesian network were converted into a single number by taking a weighted sum.

The data thus collected is displayed in Table VI. The results show that technique predicts the number of cache misses with a high degree of accuracy (>99%) for even the smallest training data set. Furthermore, in this case, all variables were seen to be executed in quite small data sets. An overwhelming number of the instructions were found to have the same behaviour in all cases, and these account for a large part of the accuracy of the overall accuracy. This perhaps points to a strong suitability for dynamic invariant detection [15] in the field. For those variables featuring in the Bayesian network, there appears to be a slight trend towards increased accuracy for the largest data sets.

V. CONCLUSIONS

This paper has presented an approach to cache analysis differing considerably from those currently used. At the core of the approach is the learning of a Bayesian network representation of the cache, which records the influence of instructions on each other. This model is constructed automatically from the program with minimal human intervention and no need for a specification of the cache nor for detailed understanding of the program semantics. As this is a measurement-based approach, the obtained results are only valid for soft-real time systems, however the accuracy of the model's predictions for a range of standard benchmarks are extremely close to the real behaviour.

REFERENCES

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

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

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

[4] M. Bartlett, I. Bate, and J. Cussens, "Instruction cache prediction using Bayesian networks," in *Proceedings of the 19th European Conference on Artificial Intelligence (ECAI 2010)*, 2010, pp. 1099–1100.

[5] J. Pearl, *Probabilistic Reasoning in Intelligent Systems: Networks of Plausible Inference. Representation and Reasoning Series*. San Francisco, California: Morgan Kaufmann, 1988.

[6] M. Bartlett, I. Bate, and J. Cussens, "Learning Bayesian networks for improved instruction cache analysis," in *The Ninth International Conference on Machine Learning and Applications (ICMLA 2010)*, 2010, to Appear, Available on Request.

[7] S. Edgar, "Estimation of worst-case execution time using statistical analysis," Ph.D. dissertation, University of York, 2002.

[8] F. Mueller, "Static cache simulation and its applications," Ph.D. dissertation, Florida State University, Tallahassee, Florida, 1994.

[9] ——, "Timing analysis for instruction caches," *Real-Time Systems Journal*, vol. 18, no. 2-3, pp. 217–247, 2000.

[10] H. Theiling, C. Ferdinand, and R. Wilhelm, "Fast and precise WCET prediction by separated cache and path analyses," *Real-Time Systems Journal*, vol. 18, no. 2-3, pp. 157–179, 2000.

[11] I. Puaut, "WCET-centric software-controlled instruction caches for hard real-time systems," in *ECRTS '06: Proceedings of the 18th Euromicro Conference on Real-Time Systems*. Washington, DC, USA: IEEE Computer Society, 2006, pp. 217–226.

[12] D. Heckerman, D. Geiger, and D. M. Chickering, "Learning Bayesian networks: The combination of knowledge and statistical data," *Machine learning*, vol. 20, no. 3, pp. 197–243, 1995.

[13] M. Zolda, "INFER: Interactive timing profiles based on Bayesian networks," in *8th International Workshop on Worst-Case Execution Time (WCET) Analysis*, R. Kirner, Ed., 2008.

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

[15] M. D. Ernst, "Dynamically discovering likely program invariants," Ph.D., University of Washington Department of Computer Science and Engineering, Seattle, Washington, Aug. 2000.

[16] S. Edgar and A. Burns, "Statistical analysis of WCET for scheduling," in *Real-Time Systems Symposium, 2001. (RTSS 2001)*, 2001, pp. 215–224.

[17] G. Bernat, A. Colin, and S. M. Petters, "WCET analysis of probabilistic hard real-time systems," in *Proceedings of the 23rd Real-Time Systems Symposium (RTSS)*, 2002, pp. 279–288.