Learning Bayesian Networks for Improved Instruction Cache Analysis

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

***Abstract*—As modern processors can execute instructions at far greater rates than these instructions can be retrieved from main memory, computer systems commonly include caches that speed up access times. While these improve average execution times, they introduce additional complexity in determining the Worst Case Execution Times crucial for Real-Time Systems. In this paper, an approach is presented that utilises Bayesian Networks in order to more accurately estimate the worst-case caching behaviour of programs. With this method, a Bayesian Network is learned from traces of program execution that allows both constructive and destructive dependencies between instructions to be determined and a joint distribution over the number of cache hits to be found. Attention is given to the question of how the accuracy of the network depends on both the number of observations used for learning and the cardinality of the set of potential parents considered by the learning algorithm.**

***Index Terms*—bayesian networks; machine learning; instruction caches; worst case execution time**

I. Introduction

For many computing applications, correct execution of a program is equivalent to the correct functional behaviour of the system. However in the domain of Real-Time Systems, programs must not only have correct functional behaviour but also correct temporal behaviour. For example, flight controls in an aircraft must be guaranteed to respond within a given period of time. Most commonly, such temporal constraints consist of a deadline before which a task must be guaranteed to terminate. In order to create schedules that guarantee these deadlines are met, it is first necessary to obtain the Worst Case Execution Time (WCET) of a task, that is, the greatest number of clock cycles that the task could execute for given any input and any initial hardware state.

Due to the infeasibly high number of possible combinations of hardware state and inputs for many realistic programs, it is impossible to determine the WCET exactly through exhaustive testing. Similarly, the halting problem precludes the exact determination of this quantity through reasoning about the program structure and hardware configuration [1]. Instead the process of static analysis is commonly used, which mathematically reasons about the behaviour of the program and hardware, overcoming the halting problem by specific generalisations of the mathematical operations which are guaranteed not to underestimate the real execution time [2]. While this aproach is typically used academically, practical issues mean that measurement-based approaches are commonly needed industrially. In any case, static analysis almost inevitably leads to pessimism in the WCET estimate, resulting in wasted resources or unnecessarily fast and expensive hardware being purchased. This paper presents an approach using Bayesian Networks which seeks to overcome one source of this pessimism.

The rate at which processors can execute instructions has long since outstripped the rate at which those instructions can be retrieved from a computer's main memory. One common solution to this problem has been the introduction of instruction caches, which are small, fast areas of memory that sit between a computer's processor and its main memory. Instructions which it is believed will be soon needed by the processor are placed in the instruction cache, reducing the amount of time needed to retrieve these instructions for execution. While this results in a decrease in typical execution time, it makes reasoning about the temporal behaviour of programs much more difficult; the number of clock cycles to fetch an instruction for execution can now vary by an order of magnitude depending on whether or not it is already in the cache when required. It is therefore important to estimate accurately which instructions are in cache when needed in order to avoid pessimism in the WCET estimate. It is this problem that this paper seeks to address.

Traditional approaches to this problem have been based on creating mathematical models of caches and modifications to the static analysis techniques [3]. However, these have problems, as will be discussed in Section II. We therefore propose an alternative technique which seeks to remove the need for the manual creation of such models by the automated learning of an appropriate model from suitable data. More specifically, we outline a method by which a Bayesian network is learned from data obtained during some example executions of the program. This Bayesian network can then be utilised to determine which instructions have an effect on the caching behaviour of which other instructions, or that can be used directly to determine a joint probability distribution over the number of instructions not present in cache when needed. Through basic knowledge of the functionality of caches, we are able to vastly reduce the number of potential Bayesian network structures that form valid models of the cache, making it feasible to exhaustively search for the network with maximum likelihood given the

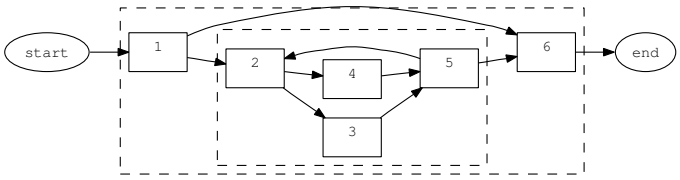

Fig. 1. The control flow graph for an example program.

observed data.

Building on this, two parameters affecting the learning of the model are studied. Firstly, we assess the question of how the quality of the learned network is related to the amount of data from which it is created. Secondly, we look at the trade-off between the accuracy of the learned network and the number of potential parents considered for each node in the network during the learning process. We evaluate these two parameters in part by assessing the determinism of the learned network. We know from the functionality of a cache that the perfect network would consist of entirely deterministic nodes, hence the degree to which the learned network possesses this property is a good guide to quality of the learning.

The rest of this paper is organised as follows. Firstly, section II presents an overview of the existing approaches to WCET and cache analysis along with the issues that motivate the need for this new approach. Section III then presents details of this approach, focussing on the techniques used for the construction of the appropriate Bayesian network. Section IV evaluates the technique on an examples and studies the role of two parameters associated with the learning. Finally, section V concludes.

II. Background and Related Work

A. Worst Case Execution Time Analysis

There are two major approaches to determining the WCET of a program. For systems in which the WCET must be absolutely guaranteed (such as safety critical systems), static analysis [2] is used to reason about the code, approximating the behaviour of the code in a way that will give an overestimate of the real WCET. This is safe, yet introduces waste from the pessimism in the estimate. Where missing deadlines is permissible but undesirable, such as in audio-visual application, testing [2] allows a tighter estimate of the WCET to be found, but this will in general occasionally be exceeded during use.

Hybrids between the two approaches exist, in which high-level analysis of possible paths through the programs are combined with low-level timing information about these paths obtained by execution [4]. A recent trend in this area has been to attempt probabilistic WCET analysis [5] in which rather than attempt to define an upper bound on the WCET, an estimate is obtained such that it will only be exceeded with a certain probability. By the correct choice of this probability, a trade-off between high utilisation of the hardware and infrequently missed deadlines can be obtained. This paper demonstrates that the joint probability distributions represented by Bayesian Networks can be useful in determining these probabilities.

For both the static analysis and hybrid methods, the task of determining the WCET estimate ultimately consists of maximising an integer linear objective function subject to constraints imposed by the program structure.

$$\text{Total Execution Time} = \sum_{i \in I} T_i C_i \tag{1}$$

where I is the set of all instructions in the program, T_i is the time to execute instruction i and C_i is the number of times instruction i is executed in the program. Constraints between the various C_is are found by determining the structure of the program in terms of all paths that can be followed through the code. Such a structure is known as the control flow graph. For example, for the program whose control flow graph is shown in figure 1, it is possible to deduce amongst other constraints that $C_1 = 1$, $C_2 = C_5$ and $C_2 = C_3 + C_4$.

B. Caches in Real-Time Systems

Most processors, including increasingly the simple processors found in embedded systems, now feature fast access caches between themselves and the main memory in order to decrease the time spent waiting for instructions to be fetched. Instruction caches function by retaining a given amount of the most recently executed instructions, on the assumption that what has been recently used is most likely to be used again soon. Instruction caches typically do not cache single instructions at a time but rather, when an instruction must be fetched from main memory, will retrieve and store a contiguous set of instructions including the one requested. For example fetching the instruction at address 1001 may cause those at 1000, 1002 and 1003 to be retrieved and cached as well. These fixed sets of instructions are known as cache blocks. Most caches are arranged such that cache blocks are always mapped to the same particular address (or small set of addresses) in the cache, evicting whatever is currently present.

While these caches vastly improve the typical performance of computer systems, they make reasoning about the WCET of code executed on them significantly more complex. One simple solution for static analysis is to assume that all instructions are fetched from main memory every time they are needed. This guarantees the safety of the estimate, but makes it so overly pessimistic that it becomes practically useless [6].

Static cache analysis that sought to reason about caches rather than ignore them was first introduced by Mueller [3]. The technique consists of assigning a label to each instruction in the program stating whether or not it can be found in the cache when needed. These labels are shown in Table I. In order to determine the correct labelling for each instruction, the program is examined in conjunction with a model of the cache to be used. Through determining what may or may not be in the cache before the execution of each instruction, the correct label may be determined. For those instructions assigned the conflict label, the analysis was unable to determine that

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.

the instruction either was or was not in memory and thus potentially pessimistic assumptions must be made, e.g. all accesses are cache misses.

For systems with caches we can now modify the function that WCET analysis seeks to maximise (equation (1)) to become

$$\text{Total Execution Time} = \sum_{i \in I} T_i C_i + T_{miss} \times C_{miss} \qquad (2)$$

where T_{miss} is the extra time taken to fetch an instruction from main memory instead of cache (the cache miss penalty) and C_{miss} is the number of cache misses in the program.

There are problems with this technique however. The first and most obvious is that for some instructions it is impossible to state whether they will be in the cache or not when required. For example, the contents of the cache will most likely be different depending on which way the program has gone through an `if then else` statement. This means that a pessimistic assumption will need to be made about the cache contents after the statement, making the WCET estimate less accurate, or that both potential cache contents will need to be stored, leading to a combinatorial explosion in computational complexity.

We believe that more robust, probabilistic techniques which do not rely on detailed knowledge of the hardware are clearly needed. This position is backed by others [2].

C. Bayesian networks

Bayesian networks are a form of graphical model that record a joint probability distribution over a set of random variables [7]. A Bayesian network consists of a directed acyclic graph (DAG) in which each node corresponds to a random variable, and edges between nodes indicate conditional dependencies between the appropriate variables. A conditional probability distribution for each variable is associated with the corresponding node in the network.

While Bayesian networks can be constructed analytically for problems which are fully understood, it is also possible to infer both the structure of the network and the associated conditional probabilities from data [8]. For any Bayesian network without a prior probability over the network structure, the structure for which the actual data was most likely to have been observed is the one that maximises the following quantity.

$$\prod_{i=1}^{n} \prod_{j=1}^{q_i} \frac{\Gamma(N'_{ij})}{\Gamma(N'_{ij} + N_{ij})} \cdot \prod_{k=1}^{r_i} \frac{\Gamma(N'_{ijk} + N_{ijk})}{\Gamma(N'_{ijk})} \qquad (3)$$

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, 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 $N'_{jk} = \sum_{i=1}^{n} N'_{ijk}$.

As Bayesian networks can be used to reflect causal structures, they seem well suited for the task of cache prediction; whether or not an instruction is present in cache is directly related to what instructions have recently been executed. Though ultimately the contents of the cache are entirely deterministic, the inability to reason completely about them means a probability distribution over the current state conditional on that which has previously occurred is perhaps the fullest and most useful representation possible.

Bayesian networks have previously been used in an attempt to aid WCET analysis [9]. In that work, a Bayesian Network was learned at a much higher level of analysis than in the current paper. The aim was to characterise the way in which the execution time of various parts of the code affected the execution time of other code sections. Such timing effects could be due to cache interactions as studied here, but may also have been due to other hardware interactions such as through the state of the pipeline or branch predictor, or due to software interdependencies such as infeasible paths.

III. INSTRUCTION CACHE ANALYSIS USING BAYESIAN NETWORKS

Having identified problems with the current approaches to cache analysis and the use of probabilistic machine learning as a potential solution, it is necessary to develop a method to test this approach. The method presented here expands considerably on a technique that has been presented elsewhere [10] and develops that previous approach to increase the accuracy of the learned networks.

Our approach consists of executing the program on a subset of its possible inputs and then inferring the Bayesian network which best explains the observed patterns of cache hits and misses in this dataset. This network represents a joint probability distribution over the cache hits and misses of each instruction in the program and can therefore be used to estimate the probability of a given number of cache miss penalties being incurred. The rest of this section now explains how the relevant data is obtained, processed and then used to infer the appropriate Bayesian network. The issue of how this network can be used to estimate the number of cache misses and WCET is addressed in [10].

A. Obtaining observations

In order to learn a Bayesian network replicating the caching behaviour, it is necessary to first obtain observations of the

behaviour. There are many ways that such data can be obtained [11]. As the data required consists of the sequence of instructions executed and whether they were fetched from the cache or from the main memory on each execution, two major options are possible.

The first is some manner of hardware instrumentation, such as using a logic analyser, to monitor the address buses between CPU and cache, and cache and main memory. However, this is time consuming to set-up and ties one to a fixed hardware architecture. A simpler method is to obtain results from a hardware simulator, such as M5 [12]. For the simple chips used in most embedded systems, these simulators give highly accurate approximations of the real chips, but are easier to monitor, allow for the collection of far more detail about the operations occurring on chip (within the pipeline, for example), and permit the hardware to be modified with minimal effort.

B. Processing the observations

The output of the simulator is a sequence of instruction, cache behaviour pairs whereas Bayesian Networks perform analysis over a given set of variables. It is therefore necessary to extract appropriate variables from this sequence.

An obvious representation of variables is to construct a variable for each occurrence of each instruction. The first occurrence of instruction 3 is therefore 3_1, its second occurrence 3_2 etc. However, this creates a problem with the order in which variables occur. For example, consider the program whose control flow graph is shown in figure 1. On one execution of this program, the program may execute instruction 3 on the first iteration of the loop then instruction 4 on the next, while on a separate execution, instruction 4 may be first executed then instruction 3. For reasons that will be explained in section III-C, the order of instruction execution is useful in building the Bayesian network. A variable representation is therefore needed that shows the first execution of 3 occurs before the first 4 in the first example and after it in the second.

Therefore we choose a representation in which variables are instead indexed not by the number of occurrences of that instruction but instead by the iteration of the loop in which they occurred. This necessitates the introduction of an additional value for variables indicating that they were not executed on that iteration. Returning to the example in the previous paragraph, we can now have four variables in each of the runs 3_1, 3_2, 4_1 and 4_2, where 4_1 and 3_2 take *not-executed* values in the first run and 3_1 and 4_2 take that value in the second.

In addition to these instruction variables, it should be noted that the behaviour of some instructions are likely to depend on what has happened not on the xth iteration, but on the final iteration. Again, considering the example given in figure 1, if the loop can execute a variable number of times, we might reasonably expect the cache behaviour of instruction 6 to depend on what has happened on the final iteration. This may mean it sometimes follows 5_1, 5_2 $\ldots 5_n$ (where n is the maximum number of iterations). Learning such a structure is intuitively complex and will lead to very few observations of the behaviour for various 5_i when i is a large value.

We therefore introduce additional variables for instructions in loops, numbering them from the final iteration backwards. This creates variables recording the behaviour of instructions on the last iteration that the loop was executed, the second to last, etc. For a given instruction, X, these variables will be labelled with negative indices, X_{-1}, X_{-2} $\ldots$ respectively. In the previous example, this now means that the instruction immediately preceding instruction 6 will be 5_{-1} regardless of how many loop iterations occur in any given run of the program.

Each execution of the program therefore becomes a collection of assignments of either *hit*, *miss* or *not-executed* to each of the variables representing each instruction on each loop iteration. A dataset of executions represented in this way then becomes suitable input for the learning of a Bayesian network.

Once such a dataset is collected, some instructions in it will be observed to always exhibit the same behaviour whenever they are executed in each execution of the program. For example, in figure 1, assuming that instruction 1 is not in the cache when the program begins, it will always have the value of *miss* regardless of the execution of the rest of the program. As these instructions convey no information on how other instructions may be affected if their value were different, they can be removed from consideration when constructing the Bayesian Network. It may or may not be that the same behaviour would indeed be observed for all possible inputs; without an exhaustive execution of all inputs, such a guarantee could not be made using a testing-based approach. The fixed behaviour exhibited by these instructions can be included directly in the calculation of the WCET at the end of the process.

C. Learning the Bayesian network

As the cache state does not depend solely on what instruction has just been executed but also on the path taken to reach that point, we cannot construct a Bayesian network for the cache that exactly replicates the structure of the control flow graph. In other words, the outcome of a cache fetch is not a Markov process of the current instruction, but depends on potentially all previously executed instructions. Therefore, a tree structure for the Bayesian network in which each node represents an instruction execution and also the path along with the program had travelled to reach that instruction would be possible, but would also suffer from a combinatorial explosion in the presence of loops in the control flow graph. For example, in figure 1 there is one path that arrives at instruction 3 on the first iteration of the loop, 2 paths by the second, 4 paths on the next, then 8, 16, 32 .... If the program could iterate thousands of times, such a network would be too large to store, and would require an excessive number of examples in order to produce conditional probability tables for each node.

Instead, the structure of the Bayesian network is created by a learning process. For each variable identified in the previous

stage, a node will be created in the final network, and parents that may affect it are learned from other variables. For each node, all sets of potential parents are generated and the set which maximises a variant of the likelihood score given by the equation in Section II-C is found by exhaustive search.

Using the equation given in Section II-C directly would allow for all possibilities of value to be predicted for a node (with certain probabilities). In fact, we wish to create a network which assigns probabilities of each node being either a *hit* or *miss*, but does not make predictions about whether an instruction is executed or not. Such an issue is part of the flow analysis of the problem and is determined by another part of the WCET analysis. What is in fact desired is a network in which each node predicts the probability of a *hit* or *miss* given that the instruction is executed, but that still makes use of whether other instructions are *not-executed*.

The appropriate quantity that each node, i, seeks to maximise in its choice of parents is therefore:

$$\prod_{j=1}^{q_i} \frac{\Gamma(N'_{ij})}{\Gamma(N'_{ij}+N_{ij})} \cdot \prod_{k=1}^{r'_i} \frac{\Gamma(N'_{ijk}+N_{ijk})}{\Gamma(N'_{ijk})} \tag{4}$$

where r'_i is the set {*hit*,*miss*} (i.e. the possible values of i except *not-executed*) and all other symbols are as defined in section II-C.

Conditional probability tables in the nodes are then computed directly from the frequencies of the observations of each combination of the values of the parents and the variable itself in the dataset. Again these tables are computed such that they represent the conditional probability of an outcome given that the outcome is not *not-executed*.

The exception to this learning is the nodes calculated from the final iteration backwards i.e. those nodes defined above as having negative indices. These nodes may be used as parents of other nodes, but are added to the Bayesian network without having their own parents learned, but rather have their parents set as all nodes that refer to the same number instruction and which have positive indices. For example, the parents of 4_{-1} (the last iteration of instruction 4) will be set to $\{4_1, 4_2, 4_3 \ldots 4_n\}$ where n is the greatest number of iterations observed. The conditional probability distribution of these nodes is set based on the probabilities of each parent node and the probability of that parent node being the parent with the largest index which does not have a value of *not-executed*.

As even simple programs may consist of thousands of instructions and the learning of the Bayesian network is NP-hard in the general case [8], we introduce domain specific knowledge into the learning process. This significantly reduces the search space of potential Bayesian networks and hence makes the learning feasible. The cache contents depend on those instructions that have already been executed. Therefore, the set of potential parents for each instruction can be limited to only those which occur at some point before the instruction being considered. This can be done without adversely affecting the network learned, indeed it may improve the network by removing from consideration parents that could not possibly be parents in the real world but that nevertheless have fortuitous correlations with the node. It is for this reason that the representation of variables adopted in section III-B was chosen. The partial ordering of instructions that this representation permits makes it possible to determine exactly those variables which occur before others and hence may be parents to those later nodes.

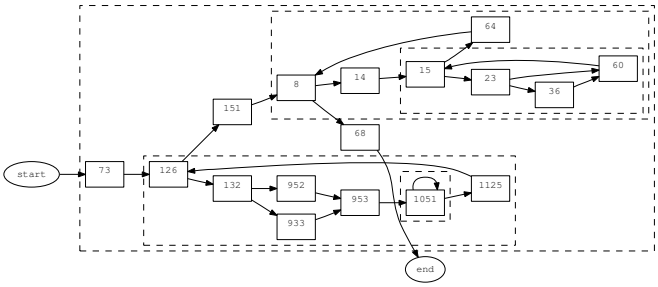

Fig. 2. The control flow graph for the bubblesort program.

Observing that the cache contents depends most on those instructions most recently executed, we may also limit the set of parents to consider to those instructions that have been executed within a given number of steps previous to this instruction along all possible execution paths. Unlike the previous limiting of parents to consider, this restriction may result in some loss of precision of the learned network as the functioning of caches occasionally allows for some long since executed instructions to still be present in the cache. In choosing the number of previous instructions to consider, there is a trade-off to be made between the accuracy of the learned network and the speed at which it is learned. This trade-off is studied in greater detail in the next section.

The control flow graph of the program under consideration can be used to identify those instructions whose potential execution occurs within a set distance before the instruction of interest.

IV. Results and Evaluation

The technique was evaluated on a bubblesort routine adapted from the collection of benchmarks held by Mälardalen University[1] which are commonly used in evaluating WCET techniques. Specifically, we use a version of the code which takes an input vector containing between 10 and 100 integers in the range [0,10] and rearranges them in ascending numerical order. This simple program inherently features different paths through the main loop for different inputs (as, at each comparison between two values, the program may take the swap or not swap path). Through choosing to sort a variably lengthened input, the number of iterations through the main loops also varies between different runs. The control flow graph associated with this program is shown in figure 2.

In this paper, the aim of the experiments was to assess how the quality of the learning depended on two different factors.

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

Firstly, the effect of the number of observations of the program executing on the final network. Generating traces of simulated program is very time consuming and hence it would be better to avoid performing more simulations than necessary. On the other hand, without sufficient data, any learning process is likely to notice spurious correlations in the dataset that would not appear in the full set of all possible inputs. Reaching a trade-off between these two competing pressures is essential.

The second issue to study is number of nodes to consider as parents when learning the structure of the Bayesian network. In section III-C that it was stated that the most recently executed instructions would, in general, have the greatest effect on whether or not an instruction could currently be found in the cache. It was then noted that limiting the instructions considered as potential parents of each node to just those recently occurring was a possible way to reduce the number of parent sets to consider and hence make the search for the best parents feasible. However, as it is theoretically possible for any previously executed instruction to affect the current contents of the cache, such a reduction has the potential to adversely affect the accuracy of the learned network. The issue of the number of parents considered and how this affects both the time taken for learning and learned network will therefore be assessed here.

Experimental observations for these experiments were obtained using the M5 simulator in its standard configuration for the ALPHA processor. The settings for the instruction cache used was a cache size of 1kB, cache blocks of 32 bits (equating to 8 instructions) and a cache associativity of 4. These settings were chosen such that the whole program didn't fit in the cache at once (making analysis unnecessary) but are otherwise arbitrary chosen typical values.

An earlier version of this technique has been previously evaluated against the program with the control flow graph shown in figure 1 and found to correspond reasonably to the ideal Bayesian network, albeit with some overfitting [10]. As the bubblesort is a real example for which an ideal Bayesian network was not known and very difficult to derive, it is impossible to evaluate the network based on how closely it corresponds to the ideal network. Instead it will be studied in terms of two of its properties; the network structure and the conditional probability tables associated with the variables.

Table II shows data on the structure of Bayesian networks learned with various combinations of dataset size and for various numbers of parent nodes considered. The parent nodes are limited by only allowing those instructions that may have been executed less than X instructions previous to the instruction of interest to be considered. This quantity is referred to henceforth as the parent limit.

The data in table II shows two trends in the data. As the parent limit increases, the number of nodes in the network that are assigned no parents decreases and the number being assigned a single parent increased. In all cases, no nodes were found to have more than one parent for reasons that will be discussed later. This trend is true for all data sets studied but particularly pronounced for the smaller data sets.

TABLE II
NUMBER OF NODES WITH A GIVEN NUMBER OF PARENTS.

Dataset size	Number of Parents	Parent Limit				
		1	2	4	8	16
10	0	275	182	122	91	16
	1	0	93	153	184	259
20	0	453	303	207	158	46
	1	0	150	246	295	407
30	0	579	384	0	194	64
	1	0	195	321	385	515
40	0	687	457	307	231	93
	1	0	230	380	456	594
50	0	705	468	314	236	121
	1	0	237	391	469	584
100	0	814	540	362	272	199
	1	0	274	452	542	615

Secondly, as the size of the dataset is increased, the number of nodes in the Network also increases. This is partially due to the discovery and execution of instructions in the larger data sets that have not been seen in the smaller data sets. This may occur either through encountering a longer input list than the previously longest seen, or through executing instructions that were not executed on particular iterations in the smaller data sets. The increase is also due to the increased chance of observing some instructions exhibiting multiple caching behaviours in a larger data set and hence being included in the network, when in smaller data sets some instructions appeared to have non data dependent behaviour.

The functionality of a cache is deterministic therefore the ideal Bayesian network would actually have all conditional probabilities set to either 0 or 1. As well as being a more accurate model, this property would make it easier to incorporate results into a WCET formulation. In addition to examining the structure of the Bayesian network, this property can also be examined. Table III shows similar data to table II but this time focussing on the number of entries in the nodes' conditional probability table whose behaviour is either deterministic or probabilistic. For behaviour that is deterministic, we cannot say for sure that the behaviour learned is correct; we can however state that any probabilistic behaviour is definitely capable of being improved upon by the idealised correct network.

The most notable result from table III is that all values in the conditional probability tables of nodes without parents are not 0 or 1, while almost all those in probability tables with a parent are. This suggests that nodes without parents have parents missing, while those nodes with a single parent are mostly likely to have sufficient parents identified. As the parent limit is increased and more nodes become available to act as parents to a particular node, the number of underparented probabilistic nodes decreases and nodes gain parents and become deterministic.

This table also explains why nodes with greater numbers of parents were not learned. One parent was almost always sufficient to explain all the observed behaviour deterministically for this example. There was no need therefore for additional parents to be added to the network.

TABLE III
NUMBER OF ENTRIES IN CONDITIONAL PROBABILITY TABLES WHICH ARE DETERMINISTIC (D) OR PROBABILISTIC (P).

Dataset size	Number of Parents	Parent Limit									
		1		2		4		8		16	
		D	P	D	P	D	P	D	P	D	P
10	0	0	0	0	0	0	0	0	0	0	0
	1	0	0	186	0	306	0	368	0	517	3
20	0	0	72	0	48	0	32	0	24	0	8
	1	0	0	300	0	492	0	590	0	808	8
30	0	0	207	0	138	0	92	0	69	0	24
	1	0	0	390	0	642	0	770	0	1015	11
40	0	0	296	0	197	0	133	0	101	0	35
	1	0	0	460	0	760	0	912	0	1178	10
50	0	0	351	0	234	0	156	0	117	0	49
	1	0	0	474	0	782	0	938	0	1145	19
100	0	0	360	0	239	0	161	0	122	0	76
	1	0	0	548	0	904	0	1084	0	1202	25

TABLE IV
DISTANCE FROM AWAY FROM AVERAGE PARENT NODE.

Dataset size	Parent Limit				
	1	2	4	8	16
10	0	1	1.6	2.7	7.7
20	0	1	1.7	2.8	7.7
30	0	1	1.7	2.8	7.7
40	0	1	1.7	2.8	7.5
50	0	1	1.7	2.8	7.5
100	0	1	1.7	2.8	7.3

Those parented nodes which feature probabilistic entries are all found when the parent limit was at the greatest value studied. There are two explanations for this. Firstly, some of the correct parents may have now been available, so that some correct but imperfect correlation was found, while other correct parents were still laying beyond the parent limit threshold. This would mean the nodes' parents were underspecified. Secondly, as the number of potential parents increases, spurious relationships may have been detected between the larger number of variables leading to erroneous parents being assigned.

Finally, table IV shows the average number of instructions between nodes and their parents for all experimental conditions considered. As can been seen, as the parent limit is increased, the average of the parents' distance increases. This is intuitive as more potential parents (correct or spurious) at a greater distance become available as this quantity is increased. In contrast, increasing data set size as very little effect on this quantity. The values may be slightly smaller for large data sets at high parent limits, but this does not appear significant from the available data.

V. CONCLUSIONS

In this paper, we have shown how a Bayesian network that models a cache could be automatically constructed with no prior knowledge of the cache's functionality. The control flow graph representing the program has been used in order to guide the search and make the learning far more feasible, despite the fact that it does not possess the Markov property itself with respect to the cache state. It was shown that as the number of potential parent nodes that were considered in the learning process increased, the network became more deterministic, suggesting increased correctness of the network structure. However, the greatest marginal benefits from increasing the number of potential parents came at the lowest levels. This means there may well be a good compromise between the number of parents considered (and hence the feasibility of the learning) and the quality of the learned representation.

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

[4] R. Kirner, I. Wenzel, B. Rieder, and P. Puschner, "Using measurements as a complement to static worst-case execution time analysis," in *Intelligent Systems at the Service of Mankind*. UBooks Verlag, 2005, vol. 2.

[5] 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.

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

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

[8] 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.

[9] 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.

[10] M. Bartlett, I. Bate, and J. Cussens, "Instruction cache prediction using bayesian networks," in *19th European Conference on Artificial Intelligence (ECAI 2010)*, 2010.

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

[12] N. L. Binkert, R. G. Dreslinski, L. R. Hsu, K. T. Lim, A. G. Saidi, and S. K. Reinhardt, "The M5 simulator: Modeling networked systems," *IEEE Micro*, vol. 26, pp. 52–60, 2006.