Instruction Cache Prediction Using Bayesian Networks

Mark Bartlett and **Iain Bate** and **James Cussens**[1]

Abstract. Storing instructions in caches has led to dramatic increases in the speed at which programs can execute. However, this has also made it harder to reason about the time needed for execution in those domains where temporal behaviour of code is important. This paper presents a novel approach to predicting which instructions will be found in the cache when required using machine learning. More specifically, we demonstrate a method in which a Bayesian network is inferred from examples of a program running and is then used to predict the presence of instructions in the cache when the same program is run with unknown inputs.

1 INTRODUCTION

In modern computers, the CPU is capable of executing instructions at a far higher rate than those instructions can be delivered from memory. One solution that has been adopted is to introduce an instruction cache between the CPU and the main memory. This region of fast memory stores recently executed instructions and can supply these instructions to the CPU at a much more rapid rate if required again. While such caches decrease the typical execution times of programs, they also complicate the task of reasoning about these execution times. In most environments, this is not necessary but in Real-Time Systems tasks typically have deadlines which they must meet. In order to schedule such systems, an upper-bound on the Worst Case Execution Time (WCET) of each task is needed. The tighter this bound is determined, the less resources will be wasted in the system [4].

Determining the WCET estimate consists of maximising an integer linear function subject to constraints imposed by the program structure. For a system with cache, one formulation is as follows.

$$\text{Total Execution Time} = \sum_{i \in I} T_i C_i + T_{miss} \times C_{miss} \qquad (1)$$

where T_i is the time to execute i, C_i is the number of times i is executed, T_{miss} is the extra time taken to fetch an instruction from main memory and C_{miss} is the number of cache misses. Constraints between the various C_is are found from the structure of the code, which can be represented in a control flow graph, such as in Figure 1.

As the difference in time between fetching an instruction from main memory and fetching it from the cache is so great, determining which instructions are present in cache can make a huge difference to the WCET estimate. At present, cache analysis is performed by building a detailed mathematical model which is used to track what is in the cache after each instruction [2]. There are problems with such an approach though, for example, a detailed specification of the cache must be known. Additionally, different cache contents along different paths either creates a lack of precision in the model or leads to a combinatorial explosion in possible cache contents.

[1] Department of Computer Science, University of York, UK, email: firstname.lastname@cs.york.ac.uk

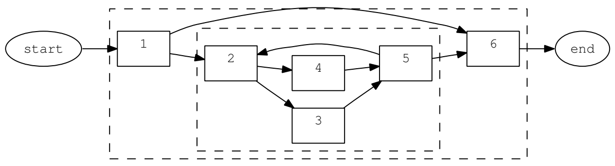

Figure 1. The control flow graph for an example program.

We propose a novel technique in which machine learning is used to determine which instructions are fetched directly from the cache. In this method, rather than deduce which instructions result in cache hits based on mathematical models, we *induce* this information from examples of the behaviour of the code when executed. By training a learner using observations of which instructions cause cache hits we seek to make subsequent predictions of cache hits and misses when the program is run with previously unseen parameter values.

The technique employed for this is to learn a Bayesian network [3] which records the influence of whether one instruction is a cache hit on the likelihood of a cache hit for other instructions in the program. 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.

2 LEARNING THE BAYESIAN NETWORK

The goal is to obtain observations of the actual caching behaviour of a program and then use these to train a Bayesian network, from which a distribution of the number of cache misses can be found and used in producing a probabilistic bound on the execution time.

Program traces for this work are obtained from a hardware simulator. For the simple chips used in embedded systems, these simulators give accurate approximations of the real chips, but are easier to monitor, allow for the collection of more detail about the chip and permit the hardware to be modified with minimal effort.

The output of the simulator is a sequence of (instruction, cache behaviour) pairs. It is necessary to extract appropriate variables from this sequence for learning. Therefore variables are defined to consist of the address of instruction executed, indexed by the iteration of the loop in which they occurred. As some instructions are likely to depend on what has happened on the final iteration, variables recording the behaviour of instructions on the last iteration of the loop, the second to last, etc. are also created. All variables take the values *hit*, *miss* or *not-executed* corresponding to their observed cache behaviour. Each execution of the program therefore becomes a collection of assignments of these values to each of the variables representing each instruction on each loop iteration. A dataset represented in this way is suitable input for the Bayesian network learning.

In this dataset, some instructions will exhibit the same behaviour whenever they are executed in each execution of the program. As these convey no information on how other instructions may be affected if their value were different, they are removed from consideration when constructing the Bayesian network. The fixed behaviour of these instructions are included directly in the WCET calculation.

For each variable, a node is created in the network, and parents that may affect it are learned from other variables. For any Bayesian network, the structure for which the actual data was most likely to have been observed is the one that maximises the following quantity [1].

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

where there are n variables, q_i is the number of combination of values for the candidate parent nodes, r_i is the number of possible values, $\Gamma(\cdot)$ is the gamma function, N_{ijk} is the number of observations of that combination of values in the dataset, $N_{jk}=\sum_{i=1}^{n}N_{ijk}$, N'_{ijk} is a constant set to $\frac{1}{q_i r_i}$, and $N'_{jk}=\sum_{i=1}^{n}N'_{ijk}$. Conditional probability tables in the nodes are computed directly from the frequencies in the dataset. The exception to this learning is the nodes calculated from the final iteration backwards. These nodes may be used as parents of other nodes, but have their parents set as all nodes that refer to the same instruction when counted forward.

As even simple programs have thousands of instructions and the learning Bayesian networks is NP-hard in general [1], domain specific knowledge is used to reduce the search space. 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 those which occur before it. Additionally, as the cache contents depend most on the most recent instructions, the parents to consider can be limited to those that may have been executed within a given number of steps previously. This second restriction may result in some loss of precision of the learned network as some long since executed instructions may occasionally still be present.

3 RESULTS

The results presented here are obtained from one of our experiments, in which the technique was evaluated on a program whose control flow graph is shown in Figure 1. The number of iterations of the loop for any execution was uniformly distributed between 0 and 100 and the probability of instruction 3 or 4 on each iteration was 50%. The caching behaviour of each instruction was set so that instructions 1, 2 and 5 did not clash with any other blocks, and 3 and 6 were placed in the same cache block, which clashed with the block holding 4. Execution of the code was simulated 1000 times. The output of this code was processed as described and a Bayesian network thus learned. The maximum number of previous potentially executed instructions to consider as parents was set to the most recent 15.

The Bayesian network learned featured the correct network with some additional links added due to over-fitting. For each instruction, the correct parents were always included in the set of learned parents, but an extra parent was nearly always added. Instruction 1 and all iterations of instructions 2 and 5 were identified as having nothing to add to the network and filtered out prior to learning as desired.

The maximum number of iterations of each instruction occurs when the loop executes the maximum number of times. Assume that

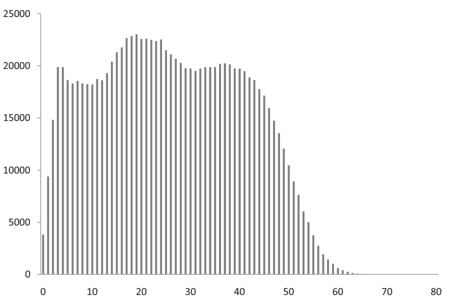

Figure 2. Frequency distribution for the total number of cache misses for instructions 3, 4 and 6 over 1,000,000 simulations.

Table 1. Number of cache misses at given percentile levels.

Percentile	90	99	99.9	99.99	99.999
Number of Hits	46	55	60	64	68

all instructions take a single clock tick and that the penalty for fetching an instruction from main memory is 10 clock cycles. The number of cache misses due to instructions 1,2 and 5 can be taken from what was observed during the processing of the execution outputs; 1 will always be a miss and 2 and 5 will be misses only on their first execution. Substituting this information into Equation (1) yields.

$$\text{Maximum Execution Time} = 332 + 10 \times C_{miss_{3,4,6}} \qquad (3)$$

As we wish to express a 'maximum' execution time that is a probabilistic estimate of the actual quantity, we use the Bayesian network to estimate the maximum number of cache hits that are likely to occur for instructions 3,4 and 6. The probability distribution described by the Bayesian network is sampled 1 million times using a Monte Carlo simulation to estimate of the number of cache misses of these instructions when executed on unknown inputs. This is shown in Figure 2. From this distribution, we can select a given percentile level of the number of misses, depending on how rarely the 'maximum' execution time may be exceeded. Statistics for various percentile levels are given in Table 1. These figures can be substituted into Equation (3) to yield execution times. For example, if the 99.9 percentile is required, a WCET estimate of 932 clock cycles is obtained.

4 CONCLUSIONS

This paper has presented a new domain for the application of Bayesian networks; cache prediction for worst case execution time analysis. We have shown how such networks may be inferred from data and then be utilised in obtaining useful estimates of the worst case temporal behaviour likely to be seen in practice.

REFERENCES

[1] David Heckerman, Dan Geiger, and David M. Chickering, 'Learning Bayesian networks: The combination of knowledge and statistical data', *Machine Learning*, **20**(3), 197–243, (1995).

[2] Frank Mueller, *Static Cache Simulation and its Applications*, Ph.D. dissertation, Florida State University, Tallahassee, Florida, 1994.

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

[4] Reinhard Wilhelm, Jakob Engblom, Andreas Ermedahl, Niklas Holsti, Stephan Thesing, David Whalley, Guillem Bernat, Christian Ferdinand, Reinhold Heckmann, Tulika Mitra, Frank Mueller, Isabelle Puaut, Peter Puschner, Jan Staschulat, and Per Stenström, 'The worst-case execution-time problem — Overview of methods and survey of tools', *Transactions on Embedded Computing Systems*, **7**(3), 1–53, (2008).