Analysis of DSP–Kernel Software by Implicit Cache Simulation

Gabriele Luculli *
Advanced System Technology
STMicroelectronics
gabriele.luculli@st.com

Alberto Sangiovanni–Vincentelli
Department of EECS
University of California at Berkeley
alberto@eecs.berkeley.edu

Abstract

We introduce a new approach to performance analysis of DSP–kernel software, based on high-level abstractions, called Implicit Cache Simulation. The method can take into account any kind of instruction cache as well as code allocation effects. We show that no loss of estimation accuracy is implied by the proposed abstractions. Moreover the speed of the method is such that it can be efficiently used as a system–level design tool. We compare implicit cache simulation with the trace–driven simulation approach, commonly used in industry. Experimental results show that our method is 4 times faster in the average and up to 11 times faster than trace-driven simulation.

1 Introduction

In this paper, we focus on embedded system applications where a micro-processor or a digital signal processor is used to implement DSP functions.

Embedded micro-processors make extensive use of pipelining and caching features to improve performance by exploiting instruction level parallelism of the computation and spatial–temporal locality of the code. Unfortunately these features are in contradiction with another key aspect of embedded systems, especially for real-time applications: predictability. To evaluate the worst-case as well as the average performance of DSP code, *system-level evaluation* is becoming more and more important [8]. New methods and software tools are badly needed.

In this paper we propose a system-level method, named *implicit cache simulation*, to estimate accurately *instruction cache performance* for DSP applications. This estimate can be used both to improve cache design and code writing.

Performance analysis depends on both software characteristics and cache architecture. To exploit application characteristics, the analysis method has to reflect all the relevant properties of the software. In particular, we noted that:

- The DSP code body often consists of one or more nested loops with constant upper bound of the iteration count, i.e., it has the *loop–kernel structure.*
- DSP code often has a single execution path (there are not if–then–else constructs). This property is important because it ensures that *infeasible paths* [10] are not present. The code is always executed in a straight–line fashion from the entry point to the ending point. This results in a deterministic behavior of the code at the control–flow graph abstraction level and there is no source of uncertain behavior in the code semantic that could result in a loss of performance estimation accuracy. This property has been called the *single feasible path property* [6].

Our approach makes use of the properties quoted above and hence it is not general. However, we obtain several advantages by restricting our analysis to software that has the loop-kernel structure and the single feasible path property. In particular, our approach has the following important properties:

- It is parametric with respect to all cache design parameters (cache size, line size, replacement policy, associativity, pre-fetch policy), thus allowing effective analysis of alternative cache architectures;
- It takes into account the allocation map of the software code in main memory for accuracy since recent research has shown that cache miss rate strongly depends on code allocation [12, 14].

ILP-based [11] and static simulation [7] are two elegant approaches that are applicable to general software structures. However, as typical of static approaches, the analysis is overly conservative and hence it may yield rather useless

*This research was carried out at Scuola Superiore S.Anna, Pisa, as part of G. Luculli 's PhD program.

0-7695-1086-8/01 $10.00 © 2001 IEEE

Simulation of a sequence of memory references:

$$\mathcal{U} : \hat{C} \times S^* \rightarrow \hat{C}$$
$$\mathcal{U}(\hat{c}, \langle s_1, s_2, ..., s_m \rangle) \equiv \mathcal{U}(\ldots \mathcal{U}(\mathcal{U}(\hat{c}, s_1), s_2), \ldots)$$
$$\mathcal{E} : \hat{C} \times S^* \rightarrow Z^+$$
$$\mathcal{E}(\hat{c}, \langle s_1, s_2, ..., s_m \rangle) \equiv \mathcal{E}(\hat{c}, s_1) + \mathcal{E}(\mathcal{U}(\hat{c}, s_1), \langle s_2, ..., s_m \rangle)$$

Simulation of a sequence of basic blocks:

$$\mathcal{U}(\hat{c}_I, \langle B_1, B_2, ..., B_k \rangle) \equiv \mathcal{U}(\ldots \mathcal{U}(\mathcal{U}(\hat{c}_I, \mathcal{P}(B_1)), \mathcal{P}(B_2)), \ldots)$$
$$\mathcal{E}(\hat{c}_I, \langle B_1, B_2, ..., B_k \rangle) \equiv \mathcal{E}(\hat{c}_I, \mathcal{P}(B_1)) + \mathcal{E}(\mathcal{U}(\hat{c}_I, \langle B_1 \rangle), \langle B_2, ..., B_k \rangle)$$

Figure 1. Extensions of the abstract functions $\mathcal{U}$ and $\mathcal{E}$

estimates especially for cache design. We deem accuracy to be the key aspect in any estimation technique, provided that speed is adequate to perform extensive design space exploration and our technique guarantees 100% accuracy for the software constructs considered. The well-known trace–driven simulation approach [15] has similar accuracy properties. However, to obtain satisfactory levels of accuracy needs long simulation runs. We show that our method is always significantly faster: sometimes an improvement of an order of magnitude is gained in processing time, even when trace-collection time is not considered. While our method, in principle, is not limited to direct–mapped caches (in fact set–associative caches with any kind of replacement policies can be analyzed as well), in the current implemented version of the implicit cache simulator just the direct–mapped cache has been considered.

The paper is organized as follows: in Section 2, a detailed analysis of previous and related work is carried out. In Section 3, the technique is presented. In Section 4, a set of experimental results is analyzed and in Section 5 some concluding remarks are offered.

2 Previous and Related Work

Performance estimation problem for cached architectures is not a new problem: a number of possible solutions have been proposed [11, 10, 7, 6].

The simplest approach is trace–driven simulation [15]. A sequence of memory references is the input to the simulator that mimics the way a real cache processes the instruction flow. During simulation, statistics are collected. Trace–driven simulation has the advantage of providing the exact miss rate when a complete address trace is processed. However, address traces are typically very large. Thus collecting detailed address traces and then processing them is a very time consuming process. Several minutes, even hours are often necessary to complete processing a single program.

Li et al. [11] proposed an analytical method for bounding the performance in the case of a direct–mapped and set–associative instruction cache [9]. The main idea is to formulate the estimation problem as an integer linear programming problem, where the cost function is the Worst Case Execution Time (WCET) of the code. Integer linear programming is an NP-complete problem and computation time may be quite large. Cache line contention can create a serious problem to the method since every time two distinct parts of code come into conflict in the same cache line, a new constraint is added to the linear integer program, thus resulting in large integer programs. Such problem is even worse for the set–associative caches [9].

Mueller et al. [7] develop a method, called *static cache simulation*, that statically predicts the behaviour of a portion of the instruction cache references for a given program with a specific cache configuration. Static analysis is very interesting for system–level evaluations because it permits both a high–level abstract view of the program and an analytical approach to bound cache effects. However, static cache simulation has a serious drawback: estimation accuracy strongly depends on the specific program, so that a fair comparison of different software implementations is difficult. In fact, accuracy can be often quite low (i.e. programs in which the estimation error was of 100% were reported in literature).

In general, these approaches assume that code is written in C or assembly language with bounded loops but without any other particular structure [10]. Recently, it has come to the attention of the embedded system community that DSP software found in existing applications is often very structured. For example, DSP–kernel algorithms are often implemented by code with the single feasible path property. Control–oriented applications are often implemented as finite state machines implying the absence of loops [4]. We make extensive use of this observation to overcome the limitations of the previous approaches.

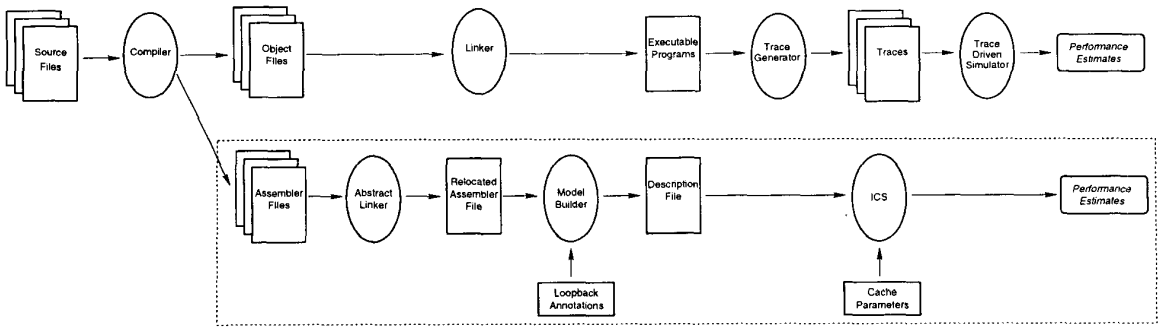

Figure 2. Overview of Implicit Cache Simulation (dashed box) and Trace–Driven Simulation approach

3 Implicit Cache Simulation

Implicit Cache Simulation (ICS) [13] is a "hybrid" method in the sense that it borrows from the trace-driven approach the idea of evaluating cache performance by simulation and it borrows from formal techniques the idea of representing the behavior of the cache with abstract models for software execution. The abstract model is application dependent and it is extracted every time a new program is run on the architecture under evaluation. Common to all software models is the exploitation of the kernel-loop and of the single feasible path properties.

There is a well known theory of static program analysis called *Abstract Interpretation* [5] based on the idea of computing the properties of programs by approximating their concrete semantics onto approximate (or abstract) semantics. In general the abstract semantic is formulated in term of abstract objects and abstract functions. Abstract interpretation is the process of evaluating the abstract semantic on specific abstract objects which can be obtained by a specific program. The evaluation process is usually based on fix-point approximations.

Our approach formulates an abstract model in term of abstract objects and abstract functions. This aspect allows to prove the correctness of the abstraction process by formal methods (as it is usually done in the abstract interpretation theory), a method is proposed in [3, 7] for the performance estimation problem. The two methods differ because our evaluation process is based on the *simulation* of the abstract model rather then its interpretation by fix-point approximations, thus resulting in a fast and accurate approach.

A description of our abstract models of the programs and the direct–mapped cache with n lines follows. Main memory is represented by a set of memory blocks $S = \{s_0, s_2, ..., s_{m-1}\}$. $S' = S \cup \{nil\}$, where *nil* is the empty element. We represent a program by an annotated control–flow graph $G(V, E)$ (see Figure 3). Each node is labeled with the name of the basic block that is represented, and with the size (in bytes) of such basic block. The loopback edges are labeled with the loop bounds. We assume that for each basic block the sequence of memory references is statically known (i.e. there is no dynamic memory management). It is then possible to represent such sequence by a mapping from control flow nodes to sequences of memory blocks $\mathcal{P}$: $V \rightarrow S^*$. The cache is represented by a set of n lines $L = \{l_0, l_2, ..., l_{n-1}\}$ and by two abstract functions: the cache eval function $\mathcal{E}$ and the cache update function $\mathcal{U}$.

Definition 1 (abstract cache state) *An abstract cache state is a mapping* $\hat{c} : L \rightarrow 2^{S'}$. $\hat{C}$ *denotes the sets of all abstract cache states. The initial cache state* $\hat{c}_I$ *maps all the cache lines to* $\{nil\}$.

Definition 2 (cache update) *A cache update function is a mapping* $\mathcal{U}$: $\hat{C} \times S \rightarrow \hat{C}$ *that describe the new cache state for a given cache state and a referenced memory block:* $\mathcal{U}(\hat{c}, s_x) = \hat{c}[l_i \mapsto s_x]$ *where* $i = x\%n$.

Definition 3 (cache eval) *A cache eval function is a boolean function* $\mathcal{E}$: $\hat{C} \times S \rightarrow \{0, 1\}$ *that describes the cache miss/hit effect for a given cache state and a referenced memory block:*

$$\mathcal{E}(\hat{c}, s_x) = \begin{cases} 1 & : \ \hat{c}[l_i] = nil \ or \ \hat{c}[l_i] \neq U(\hat{c}, s_x)[l_i] \\ 0 & : \ \hat{c}[l_i] = U(\hat{c}, s_x)[l_i] \end{cases}$$

where i=x%n.

A path is a sequence of nodes in the control flow graph, $\langle B_1, ..., B_k \rangle$ with $B_i \in V$, $(B_i B_{i+1}) \in E$, $i = 1..(k-1)$. The simulation of a path is done by applying the $\mathcal{U}$ and $\mathcal{E}$ functions to each element of the path (a formal description is reported in Figure 1).

An overview of tools and interfaces involved in implicit cache simulation is depicted in Figure 2. The path in the

dashed box represents our approach, the other represents the trace–driven simulation approach. We start with a set of assembly files, which can be obtained by compilation of the source programs or with source files, and we extract a description file. This file contains a description of the abstract model of the original programs. The cache simulator ICS takes in this description and simulates the abstract model. An example of partial simulation is reported in Figure 4 where the evaluation of the sequence $\langle B_3, B_6 \rangle$ is shown in detail. The abstract cache state at the input of basic block B_3 is indicated as $In[B3]$ whether $Out[B3]$ is the abstract cache state at the output. $Miss\_Count$ is the evaluation of the $\mathcal{E}$ function on the current state. The simulation of the sequence $\langle B_3, B_6 \rangle$ results in an increasing of the misses number by one and a new cache state, where the third cache line is filled by B_6 and a part of B_7. It is interesting to note that B_7 will never be referenced in the future because it is the first basic block of the control–flow graph. Every time there is a reference to B_6 there is also a partial load of B_7 in the cache because B_6 and B_7 are allocated contiguously in main memory. A better allocation of the SYN_BFIR kernel or a reordering of the basic blocks could avoid such effect, thus improving the miss rate.

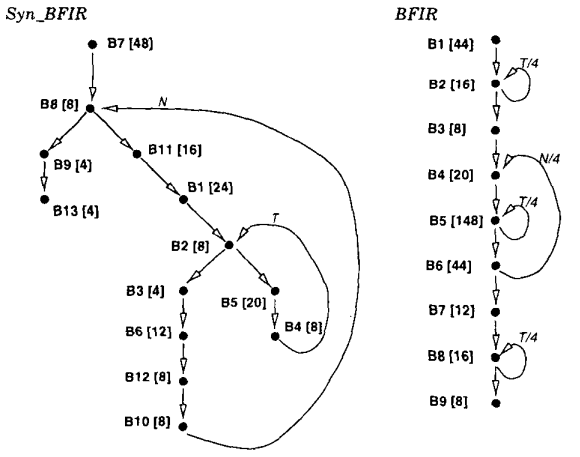

Figure 3. Annotated control–flow graph of two benchmarks

The use of simulation as evaluation process has several advantages. First of all, the accuracy of the estimates *does not depend* on the cache parameters thus allowing a comparison of the estimates of a program for different caches. Hence, ICS can be effectively used for the design and the tuning of the instruction cache, while previous approaches [7, 3] cannot be used in this mode because their accuracy depends on cache parameters and it is not predictable.

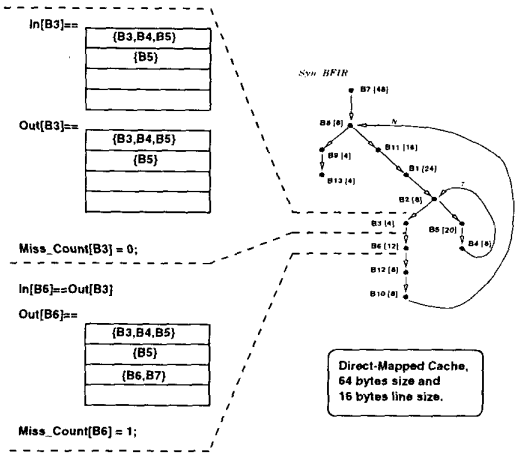

Figure 4. Partial simulation of the syn_bfir benchmark (basic block B3 and B6)

Moreover, different kind of caches, such as set–associative caches with several replacement policies or fetching policies, can be taken into account just changing the implementation of the abstract cache states and the abstract functions. This is easy to do because modeling of the cache and program behaviors are disjoint, while it is not possible in other methods [11] since they are based on a common formalism for both models. Finally, implicit cache simulation is very fast. In the next section we show that the processing time is in the order of a few seconds for our benchmark suite.

4 Experimental Results

Our approach guarantees that the "estimated" number of misses is actually an exact value (i.e. accuracy 100%) as in the case of trace-driven simulation. Together with accuracy, we have fast processing time since we do not need any trace capture stage and we execute an abstract model during simulation instead of the actual software. We now illustrate some experimental results to demonstrate the power of the approach.

For our experiments we used a Pentium II 233 Mhz machine with 32M bytes of RAM memory and Linux 4.2 OS. As benchmarks we have considered some DSP-kernel. We compared the processing time of ICS with the processing time of trace–driven simulation, which has the same level of accuracy, for some cache parameters without trace generation and then we gave some indication of the processing time needed for this task in the trace-driven approach.

Our benchmark suite consists of five kernels which are commonly found in DSP applications. Three bench-

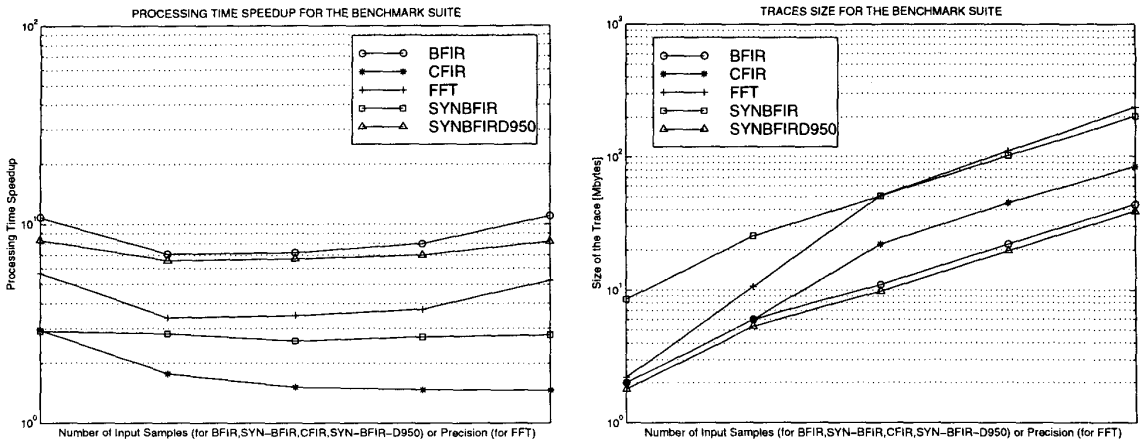

Figure 5. Diagrams of the processing time speedup and of traces size (for trace–driven simulation only)

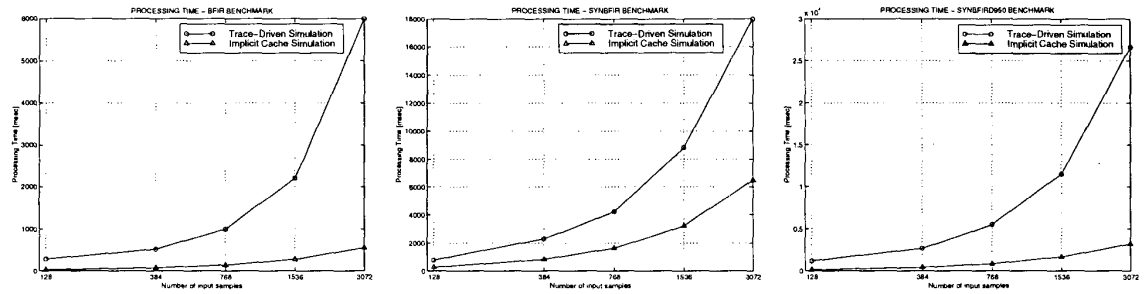

Figure 6. Comparison of the processing time for three benchmarks (bfir, syn_bfir, syn_bfir_d950)

marks (*BFIR, CFIR, FFT*) are present in the ARM Applications Library. They are written in the assembly language of the ARM microprocessor. The last two benchmarks (*SYN_BFIR, SYN_BFIR_D950*) are the implementations of a finite impulse response filter in the C language. The first one has been compiled in an assembly file by the ARM compiler [1], whether the second one has been compiled in the assembler language of the DSP-core ST-D950 [2]. Each benchmark has been tested with several data set: we tested the filters with a variable number of input samples (from 128 up to 3072) and the FFT code with a variable number of points (from 256 up to 16184). A short description of the kernels follows:

- *BFIR* is the implementation of a finite impulse response filter with 120 taps, 16 bits samples and a precision of 32 bits. The code is optimized for the ARM microprocessor, the use of the ARM assembler features make an extremely compact and efficient code.
- *SYN_BFIR* and *SYN_BFIR_D950* are the C language implementations of the previous filter. The assembler codes have not been optimized by the compilers and the memory allocations of the basic blocks are not optimized with respect to the instruction cache (see Figure 3).
- *CFIR* is an implementation of the bfir filter with complex numbers. It is as efficient as the bfir filter.
- *FFT* implements a discrete fast Fourier transform with a variable number of points. The code is highly optimized.

The experimental results are reported in Figure 5, Figure 6 and Figure 7. Figure 6 shows a comparison of the processing time for the benchmarks. Implicit cache simulation is always faster than trace–driven simulation; this effect is more and more pronounced as the number of input samples increases.

The first picture in Figure 5 shows the speedup for the benchmark suite and for each data set. It is clear that the

speedup depends on the actual benchmark; it is always larger than 1 and it climbs up to 11. The speedup is almost constant for each benchmark: it does not depend on the input data set. This is quite important, because it implies that the adoption of an exact, but implicit, simulation of the cache together with the use of an abstract model of computation is a very effective abstraction process. In fact, it yields fast simulation without any loss of accuracy. Averaging over the test cases, ICS is four times faster than trace-driven simulation.

The second picture in Figure 5 shows the sizes of the traces that have been processed by the trace–driven simulator. Most of the reported traces need more than 25 minutes to collect, and some of them need even more than 45 minutes. We can safely conclude that our method outperforms trace-driven simulation when all factors have been taken into considerations.

In conclusion, the Figure 7 shows a simple example of miss–rate evaluation. We used the implicit cache simulator to predict the miss rate for the three different implementations of the BFIR filter. We have considered two different line sizes and four different cache sizes. The results are interesting and quite odd. The ST-D950 yields the best implementation, while BFIR is the worst one. This is to be expected because the ST-D950 is a DSP core processor with a compact instruction set and several hardware devices (e.g. the hardware loop) that reduce the overhead of loop–kernels. It may seem rather strange that the BFIR optimized version of the filter is worse than SYN_BFIR, a C compiled code with no optimization. The main reason is that the SYN_BFIR loop bodies are smaller than the BFIR ones, so that a reduced number of conflict misses is produced even for a small cache of 64 bytes. In the end, a manually speed-optimized BFIR can result in poor performance if the cache is not correctly designed. Finally, an implementation of BFIR needs 256 bytes of cache to be effective while 64 bytes are enough for a good implementation of SYN_BFIR. This is a simple evaluation example that considers just a single loop–kernel function and not a complete DSP application. Nevertheless, we have been able to draw useful conclusions on cache behavior and alternative system designs, in less than a second of processing time!

5 Conclusions

A system–level approach to estimate the performance of instruction caches for DSP–kernel software has been presented. An abstract model of software execution is key to our method. The presence of the cache is taken into account by simulating its behavior. For this reason, the method has been called Implicit Cache Simulation (ICS). Experimental results show that our method is fast enough to be used as a system–level tool where many iterations on different cache architectures and parameters are needed. A comparison with trace-driven simulation shows an improvement of the processing time up to 11 times while maintaining the same level of accuracy. In general, few seconds are necessary to build a new abstract model and to simulate several configurations of the instruction cache as opposed to the long processing time that it may be needed to assemble traces necessary for trace-driven simulation. The accuracy of our method is completely independent on the particular cache architecture and on the software characteristics as long as the software has two properties (loop–kernel structure and single feasible path) that are normally satisfied by DSP-kernel functions.

References

[1] *Jumpstart Reference Manual ver. 2.0.* VLSI Technology Inc., 1994.

[2] *D950-Core User Manual ver. 1.0.* STMicroelectronics, 1997.

[3] M. Alt, C. Ferdinand, F. Martin, and R. Wilhelm. Cache behavior prediction by abstract interpretation. *Lecture Notes in Computer Science*, 1145:52, 1996.

[4] F. Balarin, M. Chiodo, P. Giusto, H. Hsieh, E. Sentovich, B. Tabbara, A. S. Vincentelli, A. Jurecska, L. Lavagno, C. Passerone, and K. Suzuki. *Hardware–Software Co–Design of Embedded Systems: The Polis Approach.* Kluwer Academic Pub, 1997.

[5] P. Cousot. Abstract interpretation. *ACM Computing Surveys*, 28(2):324–328, June 1996.

[6] R. Ernst and W. Ye. Embedded program timing analysis based on path clustering and architecture classification. In *Proc. of International Conference on Computer–Aided Design.* IEEE, USA,1997.

[7] C. A. Healy, R. D. Arnold, F. Mueller, D. Whalley, and M. G. Harmon. Bounding pipeline and instruction cache performance. *IEEE Transactions on Computers*, 48(1):53–70, Jan. 1999.

[8] L. Lavagno and A. S. Vincentelli. System–level design models and implementation tecniques. In *Proc. of Conference on Application of Concurrency to System Design*, pages 24–32. March 1998.

[9] Y.-T. Li, S. Malik, and A. Wolfe. Cache modeling for real–time software: Beyond direct mapped instruction cache. In *Proc. of the 17th Real–Time Systems Symposium*, Dec. 1996.

[10] Y.-T. S. Li and S. Malik. Performance analysis of embedded software using implicit path enumeration. In *Proc. of the ACM SIGPLAN 1995 Workshop on Languages, Compilers, and Tools for Real-Time Systems*, NY, USA, Nov. 1995.

[11] Y.-T. S. Li, S. Malik, and A. Wolfe. Performance estimation of embedded software with instruction cache modeling. *ACM Transactions on Design Automation of Electronic Systems*, 4(3), July 1999.

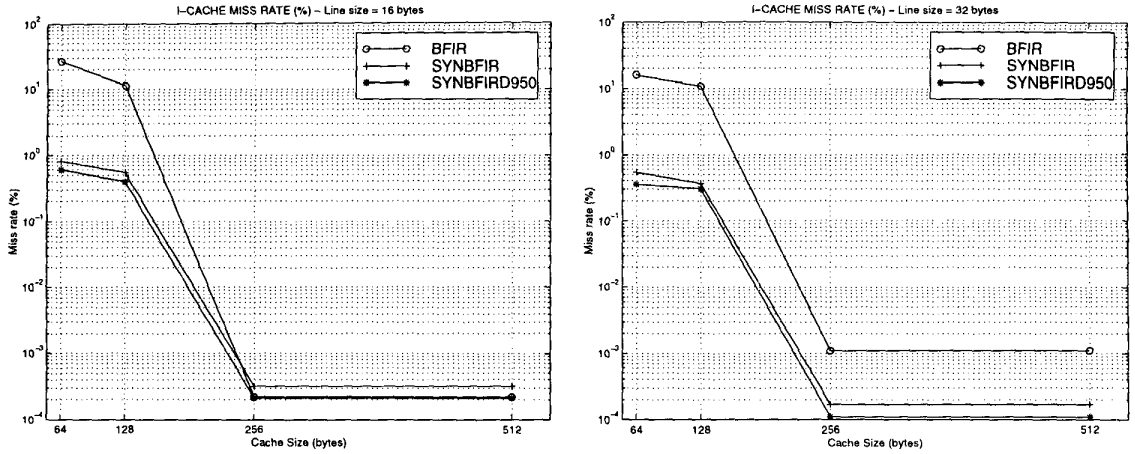

Figure 7. Evaluation of I-Cache miss rates with constant line size by implicit cache simulation

[12] S. Lorenzini, G. Luculli, and C. A. Prete. A fast procedure placement algorithm for optimal cache use. In *Proceedings of 9th Mediterranean Electrotechnical Conference*. IEEE Press, 1998.

[13] G. Luculli and A. Sangiovanni-Vincentelli. A software tool for the timing analysis of embedded software. In *Proceedings of 10th Mediterranean Electrotechnical Conference*. IEEE Press, 2000.

[14] H. Tomiyama and H. Yasuura. Optimal code placement fo embedded software for instruction caches. In *Proceedings of European Design&Test Conference*, pages 96–101. IEEE Computer Society Press, 1996.

[15] R. A. Uhlig and T. N. Mudge. Trace–driven memory simulation: A survey. *ACM Computing Surveys*, 29(2), June 1997.