Fast Instruction Cache Simulation Strategies in a Hardware/Software Co-Design Environment

MARCELLO LAJOLO
Politecnico di Torino
laolo@polito.it

LUCIANO LAVAGNO
Politecnico di Torino
lavagno@polito.it

ALBERTO SANGIOVANNI-VINCENZELLI
University of California at Berkeley
alberto@eeecs.berkeley.edu

Abstract—
Cache memories are one of the main factors that affect software performance, and their use is becoming increasingly common even in embedded systems. Efficient analysis of the effects of parameter variations (cache dimensions, degree of associativity, replacement policy, line size, ...) is at the same time an essential and very time-consuming aspect of embedded system design, whose complexity increases when multi-tasking and real-time aspects must be considered.

We propose a new simulation-based methodology, focused on an approximate model of the cache and of the multi-tasking reactive software, that allows one to trade off smoothly between accuracy and simulation speed. In particular, we propose to accurately consider intra-task conflicts, but approximate inter-task conflicts by considering only a finite number of previous task executions. The rationale for this choice can be found in a common pattern in embedded systems, where a "normal" data flow results in a regular intra-task common flow, interrupted from time to time by some urgent event, that pessimistically can be considered as disrupting the cache behavior.

The approach is conservative because re-execution of a task after a large amount of time will always be considered as not in cache, and the simulation speed-up is considerable, as shown by theoretical analysis and experimental results.

I. INTRODUCTION

The ever increasing importance of software in the design of embedded systems, coupled with the requirement to provide high performance and use state-of-the-art processors, mean that cache memories are becoming a significant aspect of modern embedded system design.

Cache design requires careful understanding of their behavior for a given application running in its environment. Poor choices of cache parameters may result in low or (even worse) highly unpredictable system performance. There have been numerous studies on cache performance based on trace-driven simulations [1]. Due to the cost of such exact simulations, it is often impractical to use a trace that covers the execution of a significant part of the system behavior. Collecting very long program traces has also been recognized as a very difficult task [2]. Another factor that adds even more complexity to serious cache studies is the fact that cache behavior varies heavily among different benchmarks [1], and hence among different application scenarios.

One alternative for reducing the cost of cache performance analysis is to perform simulations based on sampled portions of a workload [3]. A good sampling technique may allow us to derive meaningful cache behavior from a relatively small portion of the complete workload. The selection of memory references can be based either on the time or on the location of cache accesses.

Typically, all the approaches described above provide only raw, global information such as overall cache miss ratio. To improve cache performance, on the other hand, we need to better understand the effect of cache architecture, coding style, and software optimizations on cache performance [4]. These are the reasons why a hardware/software co-design methodology there is the strong need of automatic, efficient and easy to use strategies for simulating cache behavior with the possibility of easy tuning the trade-off between speed of the analysis and relative accuracy.

In our approach, we use a mixed strategy, in which we build an abstract model of each piece of code, that represents the memory addresses that it will use. That model can be customized to suit different memory allocation strategies (in case the designer wants, for example, to reserve some portions of the cache to some specific task) and different caching policies. The hardware/software co-simulator then models quite accurately the conflicts between all basic blocks of each task, but only approximates inter-task conflicts in order to save simulation memory and time. In this work we concentrate on instruction cache modeling, since unlike many data references, the address of each instruction is known statically (at compile time) if the code is statically linked (physically locked in memory i.e. the layout of object code into memory is fixed during execution). Note that this assumption is almost realistic because generally the implementation of the software part of an embedded system is statically linked [5], and virtual memory causes large and unpredictable delays. An extension of the strategy to data caches for statically allocated data is straightforward. We are currently investigating the problem of modeling arrays and pointers as well. Like [4] we have the possibility to visualize cache statistics during software execution and we can also visualize critical code sections with poor cache performance.

The paper is organized as follows. In section II we provide an overview of our co-design environment, and in particular of the co-simulation platform. In section III we present the architecture of our cache simulator. In section IV we present an example, showing the effectiveness of our
strategy. In section V we conclude the paper and discuss future work.

II. OVERVIEW OF THE CO-DESIGN ENVIRONMENT

POLIS [6] is a hardware/software co-design environment targeted to control-dominated embedded systems. The internal model of computation is based on Co-design Finite State Machines (CFSM) which are extended finite state machines with asynchronous buffered communication. The choice of an asynchronous type of interaction between CFSMs is due to the need to support a "neutral" high-level specification of hardware and software components in which the execution delay of a CFSM transition is unknown a priori. It just assumes an unbounded delay for the software response and the communication between CFSMs is not performed with shared variables (as in the classical composition of FSMs), but by means of events. An event is a uni-directional communication that can be sent and received, and that can carry a value.

POLIS can synthesize both a software and a hardware implementation for each CFSM, as well as a Real-Time Operating System that coordinates the execution of the software CFSMs (one task for each CFSM). In the simulation phase, both hardware and software modules are simulated in the PTOLEMY environment [7], using the synthesized software as a functional model of each CFSM.

One important characteristics of the software synthesized by POLIS from a CFSM, that we exploit extensively in our approach, is that the model of execution of CFSMs is globally asynchronous, but locally synchronous. This means that the (acyclic) path of execution for a given invocation of each task is fully determined by the presence/absence status of its input events, and by their value. Note, however, that this characteristics is only used to accelerate the simulation, and the approach can be used for non-synthesized software by considering basic blocks separately. In this work we also consider only uniprocessor architectures, and hence ignore cache coherency issues.

III. INSTRUCTION CACHE MODELING

The cache simulator that we present in this paper allows one to perform the evaluation and debugging of cache architectures for a given embedded system application.

The aim of our approach is to dramatically increase the simulation speed in respect to cache simulators that model each cache line individually, by using an abstract model of the cache behavior, that still provides valuable information to the system designer, in the form of hit/miss ratios at various level of granularity, in order to facilitate the analysis of performance-critical sections of the code and optimize the memory layout for cache performance. We first describe the cache model, and then briefly discuss those parameters.

A. The conflict graph model

The picture in figure 1 shows an example of a conflict in the mapping of two different basic blocks in the instruction cache (each box is a basic block, and each edge represents a possible flow of control). Basic block 6 is in conflict with 50% of basic block 1. The key observation is that the conflicts in instruction cache accesses can be predicted at compile time so that, in the case of direct mapped instruction cache, the layout in cache of a given path after its execution can be statically determined. In this case the information about the conflict between basic blocks 6 and 1 is taken into account and stored for further use.

For example, the cache layout shown on the right hand side of figure 1 demonstrates that after the execution of each possible path in the task (1,2,4,6), (1,2,4,5,6),(1,3,4,6) or (1,3,4,5,6), we will never find the first part of basic block 1 in the cache as well as the entire basic block 6. Hence the instruction references contained in that portion of basic block 1 and in basic block 6 will always result in instruction cache misses.

![Fig. 1. An example of a conflict in instruction cache mapping](image)

The basic steps of our instruction cache simulation strategy are:

1. Prior to execution, we add to each node (basic block) information about the start and end addresses of the associated portion of object code in main memory.

   This information can be either set manually, or provided by a software size estimator, or extracted from the output of the compiler. Then we associate a variable called hit_rate to each node, that represents the percentage of the instruction references inside the basic block that are currently estimated to be in the cache.

2. During execution, the first time a certain path of a given task is executed, the instruction cache layout of the path in memory is computed and stored in an internal data structure (see Figure 2).

3. Then the conflicts between the current path and the previously executed paths are computed. The number of previous conflicts to consider is determined by the HISTORY_LENGTH model parameter, as described in Section III-B, in order to limit the computational complexity of the model. The rationale for this choice is that our model should be conservative, and hence "old" blocks should be predicted to be not in cache.

4. Finally the percentage of in-cache blocks for the current task is used to determine the memory access overhead, and the hit_rate variable for it is updated.

In the case of set-associative instruction caches the only difference is that the layout is stored in terms of cache sets instead of cache lines.
In figure 2 we show an example of path-conflict graph that we build after the execution of the path (1,2,4,6) for the task of figure 1. The computation of the instruction cache layout of this path is straightforward, because it is sufficient to go back from the last node to the first one while taking into account bidirectional conflicts between nodes.

The analysis at step 2 can be optimized based on the above mentioned properties of software synthesized from a CFSM (that it can only have a finite number of paths, and for each execution the path is completely determined by the values of the task inputs). During step 2 we statically predict the behavior of a large portion of the instruction cache references for a given path in the task, assuming a specific cache configuration, and we store in a hash table the layout of each path for the following steps. A path can be found in the table based on the task and the path identifiers.

During step 3 we build the task-level cache interference graph. This graph has a node for each task path, and an edge for each conflict higher than a user-selected threshold (if the threshold is set to zero then the complete graph is built). The edges are labeled with the percentages of corresponding instruction reference conflicts. This graph is built incrementally during co-simulation (every time a new path is executed).

In step 4 we dynamically perform a cache simulation by using the task-level interference graph built at the previous stage of the cache analysis. In this manner we approximate the cache behavior by considering interferences between a limited number of basic blocks. However, since we use the cache model inside a co-simulation environment that can model pre-emption, we also take into account (approximately) the effect of preemption, interrupts and so on.

During simulation we can dynamically show to the user the cache statistics, both within each task (granularity at the basic block level), and for the entire system (granularity at the task level). In picture 4 there is an example of graphical statistics that we can generate for a given task in the system. The x-axis shows the basic block number, while the y-axis shows the hit rate. The task-level instruction cache statistics look similar, but in this case on the x-axis we have an integer task identifier.

![Figure 2: Intra-task analysis: path-conflict graph](image1)

![Figure 3: Inter-task analysis](image2)

![Figure 4: An example of statistics report](image3)

**Table I**

<table>
<thead>
<tr>
<th>Parameter Name</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>CACHE_SIMUL</td>
<td>0 = FAST, 1 = FULL</td>
</tr>
<tr>
<td>CACHE_SIMUL_STYLE</td>
<td>output file for traces generation</td>
</tr>
<tr>
<td>CUSTOM_LAYOUT</td>
<td>num. of prev. paths to consider</td>
</tr>
<tr>
<td>HISTORY_LENGTH</td>
<td>minimum conflict value to consider</td>
</tr>
<tr>
<td>THRESHOLD</td>
<td>cache size in bytes</td>
</tr>
<tr>
<td>LINE_SIZE</td>
<td>line size in bytes</td>
</tr>
<tr>
<td>ASSOCIATIVITY</td>
<td>1 = direct mapped</td>
</tr>
<tr>
<td>REPLACE_POLICY</td>
<td>random, lru, fifo</td>
</tr>
</tbody>
</table>

**Global cache parameters**

B. Model parameters

Table I shows all the parameters that the user can set to customize the cache performance model during a PTOLEMY co-simulation of the embedded system.

The CACHE_SIMUL parameter is used to turn on and off dynamically the cache model during co-simulation. The CACHE_SIMUL_STYLE parameter is used to select the style to use for cache simulation (FAST corresponds to the FAST cache simulation approach that is described in this paper, FULL is the FULL simulation). TRACE_GEN is used to optionally generate output traces in a textual file, in order to run a more precise batch cache simulation afterwards. CUSTOM_LAYOUT defines the loading strategy for software tasks in memory. If it is set to 1, other task-specific additional parameters which define, for example, the exact position of loading of each task in main memory will be taken into account. HISTORY_LENGTH defines the number of previous paths to consider during inter-task conflicts analysis. THRESHOLD defines the minimum value of conflicts (between basic blocks during intra-task
conflicts analysis and between tasks during inter-task conflict analysis) to consider. \texttt{CACHE\_SIZE} defines the size of the cache in bytes. \texttt{LINE\_SIZE} defines the size of a line of the cache in bytes. \texttt{ASSOCIATIVITY} represents the level of associativity of the cache. \texttt{REPLACE\_POLICY} defines the policy that is used to deal when some entries in main memory map to the same cache line in cache.

IV. EXPERIMENTAL RESULTS

In this section we report some experimental results obtained by applying our fast cache simulation approach to a case study taken from the automotive domain: a car dashboard controller; this is a control dominated application with both hard and soft deadlines. The interested reader can find additional information on this system and on its description in the POLIS environment in [6].

The system is composed by 13 interconnected CFSMs, all described in the ESTEREL language. We have first generated a C description for all modules with the POLIS software synthesis path and then the assembly code for the Motorola 68332 architecture by using the commercial Introl environment. Prior to co-simulation we have also back-annotated the C code generated by POLIS by adding to each basic block the information about start and end addresses, as well as the complete sequence of instructions (as described in section III-A).

Figure 5 plots the number of predicted misses vs. the value of the \texttt{HISTORY\_LENGTH} for four different values of the \texttt{THRESHOLD} with the same hardware/software partition. The continuous line corresponds to the result obtained with the full simulation.

Figure 6 plots the simulation overhead of the approach with respect to the purely functional case (without considering the cache) under 4 different partitioning schemes.

The conclusion that can be drawn from these results is that the accuracy of our approach increases very rapidly by increasing the value of the \texttt{HISTORY\_LENGTH} parameter and reducing the value of the \texttt{THRESHOLD}. In particular, with a value of 4 as \texttt{HISTORY\_LENGTH} and 0.1 as \texttt{THRESHOLD}, the error with respect to the full simulation approach is under 2%; moreover it is not worth to use a value higher than 8 as \texttt{HISTORY\_LENGTH} because it is not characterized by a significant improvement in terms of accuracy.

The simulation time increases only slightly with respect to the purely functional case and becomes significant only when \texttt{HISTORY\_LENGTH} is increased above 8. In particular, for a value of 8 as \texttt{HISTORY\_LENGTH} we have a maximum of 6 times as degradation in performance with respect to the 80 times of degradation obtained with the full cache simulation.

V. CONCLUSIONS AND FUTURE WORK

An approach to the integration of a fast instruction cache simulation mechanism in a hardware/software code design environment has been presented. The analysis is focused on intra-task cache conflicts with only approximate estimates inter-task conflicts, which creates a good trade-off between accuracy and speed in the simulation and analysis for embedded systems. The results show that the approach can be seamlessly integrated in the POLIS environment.

In the future, we are planning to extend cache analysis also to external hand-written C functions called by code synthesized from CFSMs. The challenge now is greater because the code is no longer acyclic, and could include pointers and arrays. We are also working on a cache layout optimization loop that would determine the best position for each basic block automatically, based on memory size and performance constraints.

REFERENCES