Fast Cache Simulation for Host-Compiled Simulation of Embedded Software

Kun Lu, Daniel Müller-Gritschneder and Ulf Schlichtmann
Institute for Electronic Design Automation
Technische Universität München, Munich, Germany

***Abstract*—Host-compiled simulation has been proposed for software performance estimation, because of its high simulation speed. However, the simulation speed may be significantly lowered due to the cache simulation overhead. In this paper, we propose an approach that can reduce much of the cache simulation overhead, while still calculating cache misses precisely. For instruction cache, we statically analyze possible cache conflicts and perform cache conflicts aware annotation for host-compiled simulation. Within loops, the conflicts are dynamically captured by tagging the basic blocks instead of performing the expensive cache simulation. In this way, a vast majority of the cache accesses can be saved from simulation. For data cache, aggregated cache simulation is used for a large data block. Further, the data locality can be bound by considering the data allocation principle of a program. Experiments show that our approach improves the speed of host-compiled simulation by one order of magnitude, while providing the cache miss numbers with high accuracy.**

I. Introduction

Over the years, the cost and effort of software development has the tendency to outweigh the hardware development in the design of system-on-chip (SoC). For software development, virtual prototypes (VPs) have been widely used to enable early software performance estimation, debug and verification. To simulate the SW programs before the actual hardware is available, instruction set simulators (ISS) are often used. An ISS interprets the instructions of a *cross-compiled* program as the target processor would do. However, besides the high modeling effort, the simulation speed of ISSs is usually not high enough for simulating long software scenarios, design space exploration or real-time simulation. To overcome this problem, researchers have proposed *host-compiled simulation* as an alternative and faster way of software performance estimation [1]–[11], [15] In this approach, the programs are first annotated and then directly compiled for the simulation host. The annotated information is usually extracted from the cross-compiled target binary and models the performance aspects of a program, such as timing and memory accesses. In this way, the software performance can be estimated with relatively high accuracy.

One problem in host-compiled simulation is that the high overhead of cache simulation may significantly lower the simulation speed. To illustrate this problem, consider the basic annotation process of host-compiled simulation in Figure 1. First, the source code is cross-compiled to the target binary code. Then the basic blocks, i.e. nodes, in the control flow graphs (CFGs) of the source and binary code are mapped against each other. For each binary basic block, execution cycles and memory accesses are extracted, which are then annotated in the corresponding source code basic block, based on the CFG mapping. Finally, the annotated source code is directly compiled for and executed on the simulation host. During the simulation, instruction and data caches are simulated with the annotated addresses. When a cache miss happens, the cache miss penalty will be added to the simulated cycles. On one hand, cache simulation is indispensable for an accurate model. On the other hand, it may cause the problem of excessive annotation, which may diminish the speed-up offered by host-compiled simulation. As of now, existing approaches use *in-place* cache simulation. This means that cache accesses are annotated to the basic block that contains them, as shown in Figure 1. However, *in-place* cache simulation may cause too much simulation overhead. In the shown example, there are 53 cache accesses to be simulated within a source basic block. Since this basic block is in the inner loop, a large loop iteration count will incur very high cache simulation overhead. Besides, when used in transaction level models (TLM), a transaction can be called after a cache miss to simulate the cache refilling. This causes additional overhead due to synchronizing and simulating those transactions. The overhead of *in-place* cache simulation can slow down the host-compiled simulation to a great extent. When there are many cache misses, the gain of simulation speed over ISS simulation can be significantly redueced. Additionally, we point out that *non-functional* cache

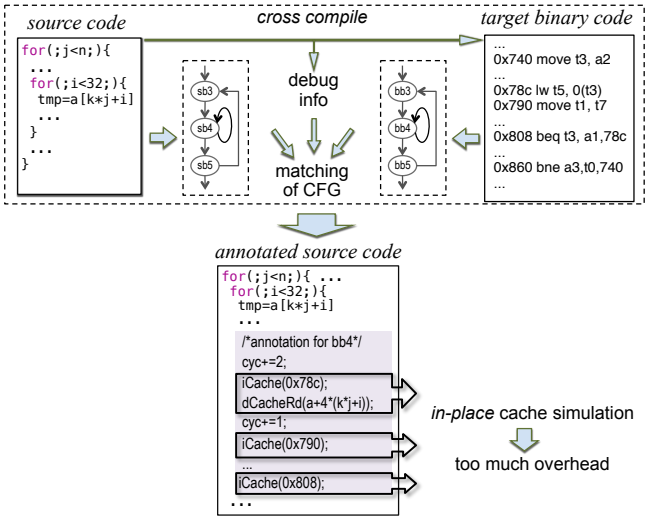


Fig. 1. Basics of host-compiled simulation with *in-place* cache simulation.

978-3-9815370-0-0/DATE13/©2013 EDAA

models are used in host-compiled simulation, meaning that no actual data are transferred and stored in the cache.

A. Our contribution

We propose an approach that is capable of removing much of the cache simulation overhead while providing precise numbers of cache misses. It identifies a vast majority of cache accesses that do not need to be annotated and performs cache-conflict aware annotation for host-compiled simulation. Consider a simple example: assume we are reading an instruction 100 times consecutively. Then only the first read requires instruction cache simulation. The remaining 99 reads do not need to be simulated, because the instruction is already cached. Likewise, only a few cache accesses need to be simulated for loops that execute an instruction sequence repeatedly. Further, our approach can handle all possible execution paths of the control flow to accurately capture the cache misses due to cache conflicts. Specifically, it presents efficient strategies for both instruction and data cache simulation.

For instruction cache simulation, our approach performs cache miss analysis for all loops at annotation time. The addresses of the instructions are known after compilation. Thus, given a particular cache configuration, it can usually be determined statically whether there are internal instruction cache conflicts within a loop body. If not, then the instruction cache accesses need only to be simulated once, e.g. after the loop body. If there are internal conflicts, our approach identifies the execution paths causing those conflicts. It then annotates a variable in corresponding source code basic blocks to count the number of conflicts. Then after the loop body, the number of cache misses is calculated using the count variables. Hence, no instruction cache simulation needs to be performed within the loop body. This considerably reduces the overhead of instruction cache simulation.

For data cache simulation, *aggregated* cache simulation is used. This means that data cache simulation is performed for a large address range, instead of for each accessed byte or word. The address range is the range of a large data block, e.g. an array. If the size of the data block does not exceed the data cache capacity, then there would be no internal data cache conflicts among the accesses to the data block. Thus, these data cache accesses need to be simulated only once irrespective of the actual access count and order. Furthermore, our approach considers the data allocation principle used by a program. All data accessed by a program locate in the stack, heap or data section in the memory. Correspondingly, we can bound the data locality to simplify data cache simulation and avoid *in-place* data cache simulation. This is also useful when the data addresses can not be determined, due to complex pointer dereferences or data structures.

B. Related work

Since its advent [1], *host-compiled simulation* of embedded software has been continually researched over the years [2]–[11], [15]. Substantial progress has been made regarding the back annotation of timing information [2]–[6]. In order to perform realistic performance estimation, memory accesses must also be annotated to enable cache simulation [7]–[11], [15]. Although researchers are aware of the problem of cache simulation overhead, all existing approaches use *in-place* cache simulation. No concrete approach has been proposed to specially tackle this problem in the area of *host-compiled simulation*. Analytical cache miss equations have been developed for compiler or memory hierarchy optimization [12]. The concept of symbolic execution [13] is used for statically estimating the cache misses or timing for WCET analysis.These are not proposed for and can not directly be used in host-compiled simulation. Motivated by [13], we investigate the concept of cache miss equation and symbolic execution to reduce cache simulation overhead for fast host-compiled simulation.

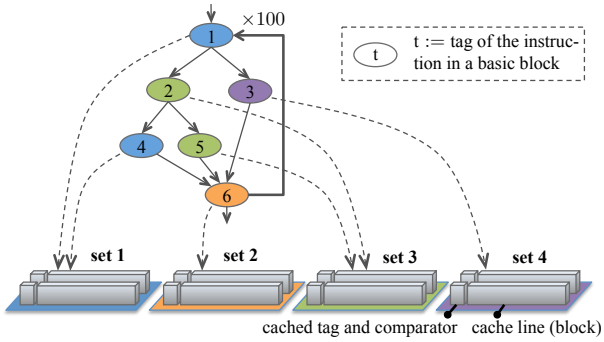


Fig. 2. Cache miss calculation - an example.

In the following, Sec. II and III present our approach. Sec. IV gives experimental results, followed by the conclusion.

II. Basic Idea and Preliminaries

A. Basic Idea

The proposed idea is that cache simulation does not need to be performed for a vast majority of the cache accesses. Consider the control flow graph of an exemplary code block in Figure 2. Without loss of generality, we assume the instructions within one basic block are cached to the same cache set and they have the same tag[1]. Further, assume a two-way associative instruction cache is used and all basic blocks in the loop are visited. As shown, both the instructions containing tag 1 and 4 are cached into cache set 1. Since there are two cache lines in each cache set, the capacity of the cache set 1 suffices to store these instructions. Therefore, no matter how many times the loop iterates, there are no more cache misses at cache set 1, except the first two cold misses. Similar analysis holds for other cache sets. In terms of annotation, instruction cache simulation can be pushed outside of the loop body, e.g. after the loop body, to capture the cold misses. In this way, the overhead of cache simulation is almost eliminated. If there are cache access conflicts within the loop, the number of conflict misses can be accurately calculated, without performing the costly cache simulation on each access. To do this, one needs

[1]Even if a basic block contains instructions with different tags, the analysis in Sec. III-A still holds, since the principles of deriving the access graph of the cache sets and calculating the cache miss counts remain the same.

to consider cache configuration, execution paths of the control flow, loop structure, etc. Details are given after introducing some preliminary terms.

B. Preliminaries

Here we give several terms that are used later in the strategies of fast cache simulation. These strategies are valid for caches with least-recently used (LRU) or first-in-first-out (FIFO) replacement policy.

Must-access: if a basic block is visited by all paths from a program point A to program point B, then a memory access within this basic block is a must-access. For example, the access to tag 1 in Figure 2 is a must-access. For loops, the must-access basic blocks are those that are within the loop body and post-dominate the loop head node. The dominance, post-dominance and loop analysis among the basic blocks can be found in [5].

May-access: if a basic block may be visited by the paths from a program point A to program point B, then a memory access within this basic block is a may-access. For example, the access to tag 4 in Figure 2 is a may-access. For loops, the may-access basic blocks are those that are within the loop body and do not post-dominate the loop head node.

Mutually exclusive may accesses: for two may accesses between two program points, if there is no path that visits both these two may accesses without passing the loop back edge, then they are mutually exclusive. For example, the accesses to tag 4 and 5 in Figure 2 are mutually exclusive.

Access trace of a cache set: a sequence of addresses accessing a set. In this paper, we only consider the tag part of an address. For cache set 1, an exemplary access trace is:

`..., 1,1,1,3,6,3,6,3,4,6,3,6,1,1, ...,`

where all the numbers are tags.

Access graph of a cache set is a graph that models possible access traces of this cache set.

Cold misses are the cache misses that are caused by the accesses to certain tags for the first time. The number of cold misses can not be reduced irrespective of the cache size.

Conflict misses are the cache misses that are caused by previous eviction of existing cache lines (blocks) due to accesses to other tags.

Lemma 2.1: Assume LRU or FIFO replacement policy, there are no conflict misses for a given access trace of a cache set, if the number of different tags in this trace is smaller than the number of cache lines (blocks) in this cache set.

Theorem 2.1: For the accesses to a continuous address region, there are no conflict misses, if the size of this region is smaller than the cache size.

Proof: Within a continuous address region, the number of tags cached to a cache set is smaller than the number of cache lines (blocks) within the cache set. According to Lemma 2.1, there are no conflict misses. ■

III. Strategies for Fast Cache Simulation

In this section, we first show the proposed strategies for fast simulation of both instruction and data caches. Then we show how the source code is annotated and discuss the corresponding timing simulation.

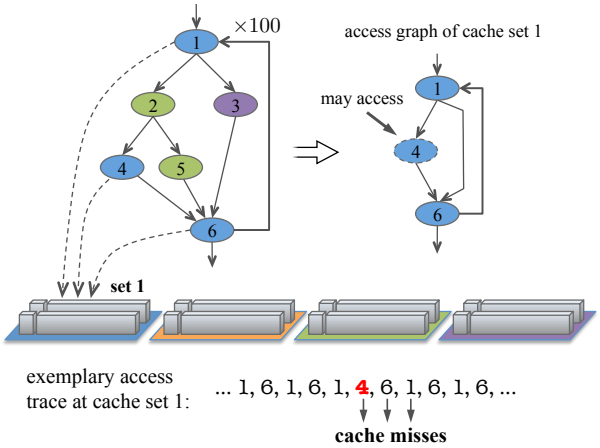


Fig. 3. Conflict misses in loops.

A. Fast instruction cache simulation

At annotation time, the instruction addresses are known. For a given cache configuration, it can be determined which cache set an instruction will be cached to. Thus we can obtain the access graph for each cache set from the control flow graph. For a specific execution of the access graph, the number of cache misses at a cache set is deterministic and thus can be accurately calculated. In the following analysis, we assume 2-way associative caches are used, with LRU or FIFO replacement policy.

1) Handle a single loop: Without loss of generality, we show how to calculate the number of cache misses at one cache set, considering several cases.

Case 1) Cache set capacity is not exceeded. The number of tags is not larger than the number of cache lines. This is the simplest case, in which no conflict misses will happen. The example in Figure 2 shows one such case.

Case 2) Cache set capacity is exceeded, which results in conflict misses. The number of conflict misses can be precisely calculated under the following conditions.

Case 2a) The must-accesses fill up the cache set and the conflict misses are caused by a may-access or several mutually-exclusive may-accesses. Consider the example in Figure 3. For the code block under analysis, an access graph of cache set 1 is constructed. In this graph, accesses to tag 1 and 6 are must-accesses, while those to tag 4 are may-accesses. In every loop iteration, accesses to tag 1 and 6 will fill cache set 1 to capacity. Thus, a cache miss occurs each time for each access to tag 4. After this miss, tag 6 will be evicted. Thus, the next access to tag 6 gives a miss and evicts tag 1. As depicted, three consecutive misses are caused. In total, the number of cache conflict misses for this code block at cache set 1 is calculated as

$$M_{set1} = (N_{way} + 1) \times cnt_{tag4} = 3 \times cnt_{tag4}, \qquad (1)$$

where $N_{way} = 2$ is the number of cache lines in a cache set and cnt_{tag4} is the count of accesses to tag 4. To capture the count of conflict misses, we accumulate the execution count cnt_{tag4} of the basic block containing the access to tag 4. Then after the loop body, this count is substituted in (1) to calculate M_{set1}. Now assume tag 5 is also cached into cache set 1. Since accesses to tag 4 and tag 5 are mutually exclusive, we

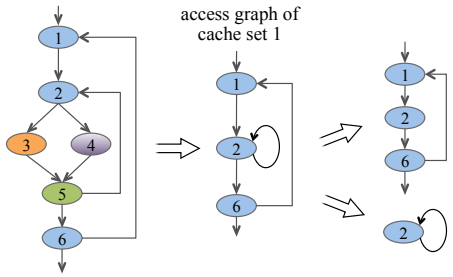


Fig. 4. Consider nested loops.

have the number of conflict misses as

$$M_{set1} = 3 \times (cnt_{tag4} + cnt_{tag5}). \tag{2}$$

Case 2b) The must-accesses exceed the cache set capacity. This leads to the most conflict misses. Assume in a loop body there are three must-accesses to tag 1, 4, 9 cached at cache set 1. Then the access trace of cache set 1 within this loop is:

```
..., 1,4,9,1,4,9,1,4,9, ...
```

Every access in this trace will cause a miss, due to similar reasons of those misses in Figure 3. If there are other may-accesses at cache set 1, then each may-access will also give a miss. For example, assume a may-access to tag 3, one possible access trace is:

```
..., 1,4,9,1,3,4,9,1,4,9,1,3,4,9, ...
```

Each access to tag 3 will cause a miss. The total number of misses at cache set 1 is calculated as:

$$\begin{aligned} M_{set1} &= cnt_{tag1} + cnt_{tag4} + cnt_{tag9} + cnt_{tag3} \\ &= 3 \times cnt_{loop} + cnt_{tag3}, \end{aligned} \tag{3}$$

where $cnt_{loop} = cnt_{tag1} = cnt_{tag4} = cnt_{tag9}$ is the loop iteration count.

Summing up the considered cases, we have

$$M_{set_i} = \begin{cases} 0, & case1 \\ (1 + N_{way}) \times \Sigma cnt_{may}, & case2a \\ \Sigma cnt_{must} + \Sigma cnt_{may}, & case2b \end{cases} \tag{4}$$

The total number of cache misses is:

$$M = \Sigma_{i=1}^{S} M_{set_i}, \tag{5}$$

where S is the number of cache sets.

2) Handle multiple loops: For loops that are sequentially executed, (4) can be used to handle them independently. For nested loops, we first decouple them to single loops and then apply (4) to estimate the cache misses for each of them. Consider the example in Figure 4. For loop 1, based on its access graph of cache set 1, The number of conflict misses for loop 1 is $3 \times cnt_{loop1}$. Within loop 2, there is no conflict misses at cache set 1. Thus for these two loops, the total number of cache misses at cache set 1 is $3 \times cnt_{loop1}$. Similar principle can be used to handle more complex nested loops.

B. Fast data cache simulation

Unlike instruction accesses, the addresses of data accesses are usually unknown statically at annotation time. Although it is possible to obtain the precise data access addresses for host-compiled simulation [15], it may be inefficient to simulate every data cache access. Here we present a fast and approximate way of data cache simulation.

Firstly, our approach uses aggregated data cache simulation for an accessed data block. This means that the address range of the data block is considered as a whole to simulate data cache misses. This is normally useful when looping over large data blocks. Consider the following example:

```
1   char tmp;
2   for (i=0; i < N-1; i++){
3     for (j=i; j < N; j++){
4       tmp=buf[j];
5       ...
6     }
7   }
8   dcReadDataBlock(bufAddr, N); //dc simulation
```

If a data cache simulation is annotated for every byte after line 4, then approximately N^2 cache accesses will be simulated. Instead of doing this, data cache simulation is annotated after loop with the whole buffer size. Assume the buffer size is smaller than the data cache size, there will be no internal cache conflicts (see Theorem 2.1) and the number of cache misses is correctly simulated. Additionally, in aggregated data cache simulation, the address is increased by the size of a cache line (32 bytes). So only $\lceil N/32 \rceil$ cache accesses need to be simulated. The buffer size and address is obtained by using the Cparser [14] and a tool from [5].

Secondly, the data locality of certain data accesses can be bound by considering the principle of data allocation in the execution of a program. Data are allocated in three major memory sections, which are the stack, heap and data sections. By distinguishing the corresponding sections of the accessed data, the data locality can be bound for data cache simulation. Consider an example for data allocated in the stack:

```
1   sp = 0x1000;  //init of sp
2
3   void setkey(){
4     sp-=530;  //530: size of local stack
5     char state[16][16];
6     char buf[256];
7     for (i=0;i<16;i++){
8       ptr= i<8 ? state[8-i] : state[i];
9       for (j=0;j<16;j++){
10        buf[i*j] = ...
11        *ptr = ...
12        ptr++;
13      ...
14
15    dCache(sp, 530, numDCAccesses); //dc simulation
16    sp+=530;  //restore stack pointer
17  }
```

The local variables *state* and *buf* are allocated in the local stack of the function. The size of a function's local stack can be obtained from the cross compiled binary. The actual addresses of the variables in the stack are tracked by the stack pointer *sp* register. Assume a downward growing stack, the *sp* register is decreased and increased at the function entrance and exit, respectively. The actual stack address of a function depends on when it is called. To trace the dynamic value of the *sp* register, a global variable *sp* is annotated and updated at the function entrance and exit. Hence we can use *sp* for data cache simulation. In this example, since only local variables are used,

the accessed data addresses are bound within $[sp, sp + 530]$. If *state* and *buf* are defined as global variables, they will be allocated by the compiler in the data section. Their addresses are fixed at compile time and can be provided explicitly by the debugger. In both case, the address range can be used to bound the number of data cache misses.

The presented approximate data cache simulation is that it can not handle internal data cache conflicts. Besides, not all data in the range are accessed, hence over estimation of cache misses is possible. However, it still gives good approximation, as long as the data range is smaller than the cache size. If data cache conflicts are present, it resorts to the in-place data cache simulation for the part of code where conflicts may occur.

C. Annotation and Simulation

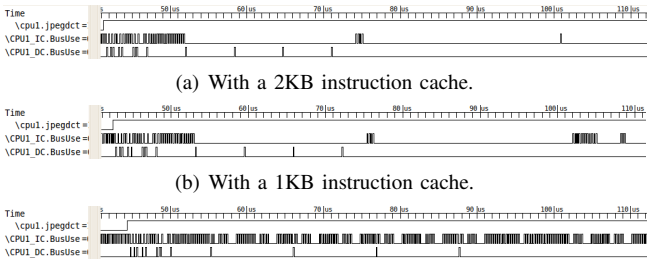
Here we show how the source code is annotated with respect to cache simulation. Then we discuss the corresponding timing estimation. Consider a code snippet in Figure 5. The annotated code lines are shaded. Assume the accesses to the instructions within basic block *bb2* cause conflicts at a certain cache set in a way as in (1). To capture the conflicts, the annotated variable *nbb2* records the execution counts of *bb2* within the loop. Then after the loop body, according to (1), *nbb2* is used to calculate the number of conflict misses that happen during the execution of this loop. Additionally, instruction cache simulation is performed for all the must accesses and the actually visited may accesses. This can simulate the cold misses with respect to the loop. It also simulates the conflict misses between the loop and the codes outside of the loop. Finally, the *sync()* function calculates the simulated time with the accumulated cycles and cache miss numbers. Then it calls the SystemC *wait* statement to synchronize timing. This means the expensive *wait* is not called every time when a cache miss occurs, but only once after a large code block. In this way, the simulation speed can be further improved. As a limitation, our approach does not provide the exact occurrence times of the cache misses. However, for fast software performance analysis, it is often sufficient to use the cache miss numbers for timing estimation. Even for multiprocessor simulation, the timing can also be approximately estimated by knowing the cache miss numbers [16].

```
void mix(){
 sp -= 270;
 ...
 nLoopCnt=0; nbb2=0;...
 for (...){
   nLoopCnt++;
   m=b[i]; //bb1
   cyc+=5;
   if(m>0){  //bb2
     a[i] = x;
     nbb2++; cyc+=2;
   }else{    //bb3
     a[i] = -x;
     nbb3++; cyc+=3;
   }
   tmp=a[i-1];  //bb4
   ...
 }
 //simulate cold misses
 nICMiss+=iCache(0x530); //bb1
 ...
 if(nbb2) //bb2 is visited
   nICMiss+=iCache(0x540);
 ...
 //conflict misses in the loop
 nICMiss+=nbb2*(NUM_WAY+1)+...;
 nDCMiss+=dCache(sp,270);
 sync(cyc, nICMiss, nDCMiss);
 ...
 sp+=270;
}
```

Fig. 5. Annotation example.

IV. Experimental Results

The experiments are performed to evaluate the simulation speed-up and accuracy given by removing the unnecessary cache simulation overhead. Several benchmark programs are simulated on a 4-core Intel machine with 4GB RAM running Linux at 2.33GHz. They are compiled with optimization level $O2$. Firstly, they are simulated by an instruction set simulator (ISS), whose results are used as the reference. Secondly, host-compiled simulation is used with *in-place* cache simulation (HS+ic). Finally, host-compiled simulation with the proposed fast cache simulation is used (HS+fc).



(a) With a 2KB instruction cache.

(b) With a 1KB instruction cache.

(c) With a 512B instruction cache.

Fig. 6. Traced cache misses for *jpegdct*.

The statistics of cache accesses and misses are shown in Table I. First of all, we can see that the results of the host-compiled simulation are very accurate, compared to ISS simulation. The number data and instruction cache accesses and misses are all well estimated. Secondly, after removing a vast majority of unnecessary cache simulation, the proposed approach is still able to give accurate cache simulation results. This can be seen from the almost identical instruction cache miss numbers in HS+ic and HS+fc simulations among all cache configurations. This implies that our approach has successfully captured both the cold misses and conflict misses.

For the program *jpegdct*, the traced cache misses in ISS simulation are given in Figure 6. There are 4 levels of nested loops in the program. Instruction cache conflicts happen within the loop at level 2, when the instruction cache size is changed from 2KB to 1KB. For a 512B cache, conflicts happen within the loop at level 3, an inner loop of the loop at level 2. Thus a lot of instruction cache misses are observed. From Table I we can see that the cache miss rate is over 10%. In the program *aes*, several small functions are called in a nested loop. For a 512B instruction cache, conflicts happen among those functions and the loop body. We first embed the CFG of the called functions into that of the caller functions and then use the proposed approach to capture the conflict misses. For *jpegdct*, *aes* and *edge_detect*, the proposed approach is able to handle the different cache conflict scenarios and provides accurate simulation results. In other programs, no instruction cache conflicts exist. The number of instruction cache misses is mostly caused by cold misses and does not change with the cache configuration.

Now we examine the gain of simulation speed offered by our approach. First, the upper figure in Fig. 7 shows the simulation performances of HS+ic simulation for the program *jpegdct*. For instruction caches with the same size, the performance slightly decreases when there are more cache lines per cache set, because there are more comparisons of tags for each cache access. For a smaller instruction cache, a clear drop of performance is observed, which is caused by the additional effort after cache misses to synchronize with SystemC kernel and perform transactions. Second, the lower figure shows the speedup of two simulation modes over host-compiled simulation with in-place cache simulation. In one mode, we use host compiled simulation with no annotation

TABLE I
BENCHMARK SIMULATION RESULTS.

SW	sim mode	# of dc access/miss	# of ic access	num. of instruction cache misses for different cache configurations (size-associativity)								
				2KB-1W	2KB-2W	2KB-4W	1KB-1W	1KB-2W	1KB-4W	512B-1W	512B-2W	512B-4W
fir	ISS	32157/424	189980	13	13	13	13	13	13	13	13	13
	HS+ic	32156/427	189731	13	13	13	13	13	13	13	13	13
	HS+fc	32156/500	189731	13	13	13	13	13	13	13	13	13
jpegdct	ISS	27517/99	66877	**46**	**46**	**46**	**229**	**321**	**425**	**2985**	**4105**	**4105**
	HS+ic	27464/100	66895	**46**	**46**	**46**	**228**	**309**	**425**	**2980**	**4100**	**4100**
	HS+fc	27464/157	66895	**46**	**46**	**46**	**228**	**309**	**425**	**2980**	**4100**	**4100**
isort	ISS	16099/26	73139	7	7	7	7	7	7	7	7	7
	HS+ic	16105/27	73177	7	7	7	7	7	7	7	7	7
	HS+fc	16105/27	73177	7	7	7	7	7	7	7	7	7
aes	ISS	2825/49	7551	**71**	**71**	**71**	**72**	**72**	**72**	**202**	**195**	**242**
	HS+ic	2833/51	7564	**71**	**71**	**71**	**71**	**71**	**71**	**198**	**200**	**246**
	HS+fc	2833/76	7564	**71**	**71**	**71**	**71**	**71**	**71**	**198**	**200**	**246**
edge_detect	ISS	163416/281	879263	16	16	16	31	26	26	48	38	38
	HS+ic	163477/279	880443	15	15	15	30	25	26	47	37	37
	HS+fc	163477/279	880443	15	15	15	30	25	26	47	37	37

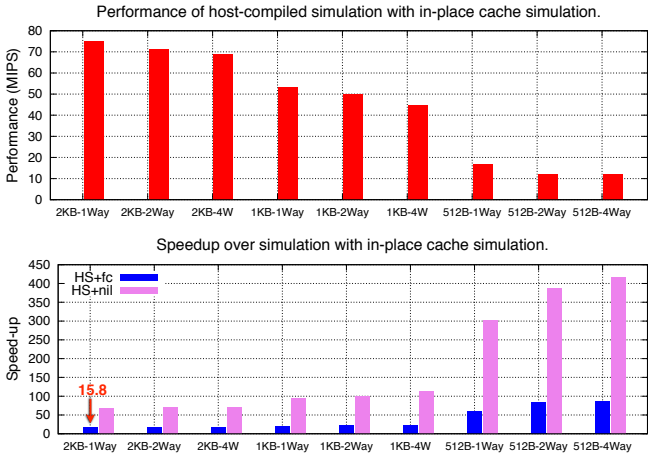


Fig. 7. Comparison of simulation performance for program *jpegdct*.

at all (HS+nil), which gives the upper bound of possible simulation performance. In the other mode the proposed approach is used. We can see that the proposed approach offers a gain of about 16 to 85 over the HS+ic simulation. This is a significant gain, considering that there is no trade-off of simulation accuracy in terms of cache miss numbers. For other programs, similar observations are obtained regarding the relation of simulation performance and the cache simulation overhead. On average, host-simulation with the proposed fast cache simulation yields an average speed-up of 12× over the simulation with in-place cache simulation.

V. CONCLUSIONS

This paper has shown that a vast majority of the cache simulation can be saved for fast host-compiled simulation, while cache miss numbers can still be well approximated for timing estimation. Our approach statically analyzes possible instruction cache conflicts and performs conflict-aware annotation to dynamically capture the conflicts. Limited symbolic simulation is used, instead of actual cache simulation. Memory allocation mechanism is considered to bound the data locality, which simplifies the data cache simulation. Experiments show that the proposed approach greatly reduces the cache simulation overhead. The simulation speed is comparable to that of native execution without annotation.

ACKNOWLEDGEMENTS

This work is party sponsored by the German Federal Ministry of Science and Education (BMBF) in the project SANITAS (16 M 3088).

REFERENCES

[1] V. Zivojnovic and H. Meyr, "Compiled HW/SW co-simulation ," in *ACM/IEEE Design Automation Conference (DAC)*, 1996.
[2] T. Meyerowitz, A. Sangiovanni-Vincentelli, M. Sauermann, and D. Langen, "Source-Level Timing Annotation and Simulation for a Heterogeneous Multiprocessor," in *Design, Automation and Test in Europe (DATE)*, 2008.
[3] P. Gerin, M. M. Hamayun, and F. Petrot, "Native MPSoC co-simulation environment for software performance estimation," in *International conference on Hardware/Software codesign and system synthesis*, 2009.
[4] S. Stattelmann, O. Bringmann, and W. Rosenstiel, "Dominator homomorphism based code matching for source-level simulation of embedded software," in *International conference on Hardware/Software codesign and system synthesis (CODES+ISSS)*, 2011.
[5] D. Mueller-Gritschneder, K. Lu, and U. Schlichtmann, "Control-flow-driven Source Level Timing Annotation for Embedded Software Models on Transaction Level," in *EUROMICRO Conference on Digital System Design (DSD)*, Sep. 2011.
[6] S. Stattelmann, G. Gebhard, C. Cullmann, O. Bringmann, and W. Rosenstiel, "Hybrid Source-Level Simulation of Data Caches Using Abstract Cache Models," in *Design, Automation and Test in Europe (DATE)*, 2012.
[7] K. Karuri, M.A.A.Faruque, S. Kraemer, R. Leupers, G. Ascheid, and H. Meyr, "Fine-grained Application Source Code Profiling for ASIP Design," in *ACM/IEEE Design Automation Conference (DAC)*, 2005.
[8] Y. Hwang, S. Abdi, and D. Gajski, "Cycle-approximate Retargetable Performance Estimation at the Transaction Level," in *Design, Automation and Test in Europe (DATE)*, 2008.
[9] J. Schnerr, O. Bringmann, A. Viehl, and W. Rosenstiel, "High-performance timing simulation of embedded software," in *ACM/IEEE Design Automation Conference (DAC)*, 2008.
[10] A. Pedram, D. Craven, and A. Gerstlauer, "Modeling Cache Effects at the Transaction Level," in *IESS*, 2009.
[11] H. Posadas, L. Diaz, and E. Villar, "Fast Data-Cache Modeling for Native Co-Simulation," *Asia and South Pacific Design Automation Conference (ASP-DAC)*, 2010.
[12] S. Ghosh, M. Martonosi, and S. Malik, "Cache miss equations: an analytical representation of cache misses," in *International conference on supercomputing (ICS)*, 1997.
[13] S. Chatterjee, E. Parker, P. J. Hanlon, and A. R. Lebeck, "Exact analysis of the cache behavior of nested loops," in *ACM SIGPLAN conference on Programming language design and implementation*, 2001.
[14] pycparser, "http://code.google.com/p/pycparser," 2011.
[15] K. Lu, D. Mueller-Gritschneder, and U. Schlichtmann, "Memory Access Reconstruction Based on Memory Allocation Mechanism for Source-Level Simulation of Embedded Software," in *Asia and South Pacific Design Automation Conference (ASP-DACA)* , 2013.
[16] K. Lu, D. Mueller-Gritschneder, and U. Schlichtmann, " Analytical Timing Estimation for Temporally Decoupled TLMs Considering Resource Conflicts ," in *Design, Automation and Test in Europe (DATE)*, 2013.