Annotation and Analysis Combined Cache Modeling for Native Simulation

Rongjie Yan
State Key Laboratory of Computer Science
Institute of Software
Beijing, China,100190
yrj@ios.ac.cn

De Ma
Institute of Microelectronic CAD
Hangzhou Dianzi University
Hangzhou, China, 310013
made@hdu.edu.cn

Kai Huang Xiaoxu Zhang Siwen Xiu
Institute of VLSI Design
Zhejiang University
Hangzhou, China, 310013
{huangk,zhangxx,xiuswen}@vlsi.zju.edu.cn

Abstract— To accelerate the speed of performance estimation and raise its accuracy for MPSoC, we propose a static analysis and dynamic annotation combined method to efficiently model cache mechanism in native simulation. We use a new cache model to statically analyze segmental profiling results to speed up simulation, and utilize a dynamic annotation technique to exactly trace the addresses of local variables. Experimental results show the efficiency of the proposed techniques for more accurate system performance estimation.

I. Introduction

Rapid growth of silicon processing technology and embedded applications provide more opportunities and challenges for multi-processor system-on-chip (MPSoC) based design. The integration of more processor components brings high performance with concurrency capability and long market period with flexible programmability [10, 9]. The MPSoC design is naturally processor-centric, and thus software-centric. As a consequence, providing techniques to obtain fast performance estimation of software (SW) running on those MPSoC architectures becomes increasing important.

Nowadays, even equipped with powerful execution units and deep pipeline architecture, memory latency still impacts the accuracy of processor performance estimation greatly. Almost all advanced processors apply cache mechanism to overcome the gap between fast processor pipeline and slow memory access. Therefore, we have to take cache effects into account to exactly estimate the performance of MPSoCs. A great variety of estimation approaches have considered cache as a key element in system performance estimation. Almost all of the instruction set simulators (e.g., ConvergenSC [2], Realview[1], and MPARM [5]) have applied detailed cache models to give accurate SW execution cycles. The function of the cache models is to manage memory access performed by their processor simulators. Though the simulation using ISS could reflect the accurate execution cycles of a system, it is not suitable for early performance estimation for the low speed. Recently, native simulation based approaches have been proposed to achieve a good tradeoff among simulation speed, accuracy and retargeting ability [11, 8]. In native simulation, software runs on a host machine natively and its execution time on the target machine is calculated using annotation or analysis technique. Most native simulation techniques use tag-search based cache models [15] or purely statistical models [11]. The method in [15] has adopted a classical cache model, with the binary code being divided into cache lines. During native simulation, whenever a line tag changes or a basic block terminates, the cache model is queried to check if a line is present. This model performs a sequential search over the whole set to check if it is hit or miss. Accessing and searching the cache model incur a much higher simulation cost. Castillo et. al. propose an improved fast Instruction Cache modeling technique [6] to avoid tag-searches by defining a two-dimension array structure for each basic block. During simulation, when an execution flow reaches the end of a basic block, we check the corresponding array element to know whether the basic block is in cache. This mechanism provides constant hit time detection. However, the cache updating granularity based on basic block level is quite small. It would become time-consuming for software codes with many branches. Even worse, few data cache modeling techniques have been considered in native hardware and software co-simulation. Compared with instruction cache model, it is difficult for data cache model to obtain the addresses of the data variables during pre-processing, for they are allocated dynamically. Pedram et. al. propose a method to obtain addresses for global data variables [13], which is only a small percentage of the total data variables. Improved techniques [7] trace addresses of all variables from native simulation for their data cache model. In this case, many annotations will be inserted into the source, and the cache model should be updated every time when data memory access happens, which cause great simulation time overhead. To reduce cache updating frequency, we have presented a segment-based technique to model instruction and data cache [12]. To model data cache accurately, it tries to establish the address offset tables for local variables in functions by analyzing the assembly code in the target processor and tracing the address of stack pointer in function callings according to the execution flow of native simulation. However, it is difficult to exactly trace the address of these local variables only with static analysis, and inaccurate address offset table may lead to wrong estimation results. Moreover, it does not handle the addressing problem of heap space.

To address the above issues, we propose a dynamic annotation and static analysis combined method to establish a fast and accurate cache model in native simulation. The main contribution is to analyze segmental profiling results statically to speed up simulation, and utilize a dynamic annotation tech-

978-1-4799-2816-3/14/$31.00 ©2014 IEEE

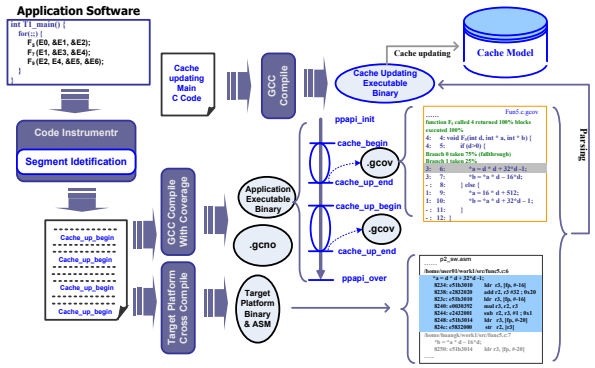

Fig. 1. The workflow of the cache modeling method

nique to trace the addresses of local variables exactly. Compared with existing works, we have mainly adopted three techniques in our cache model. First, we divide the code of an application into several segments according to the control flow of C language, and update the cache model in segments to avoid frequent cache model accessing. Second, for instructions and global or static variables, an address table of the target platform is established by statically analyzing the target-processor assembly code. Third, we insert annotation functions into GCC debugger script to trace the offset address of local variables. These annotation functions are used to establish the address table of dynamic variables by printing the variable hierarchy and their addresses in the host machine during native simulation.

II. The Proposed Cache Modeling Method

To achieve faster cache behavior simulation, we present a new cache modeling method basing on native simulation, as illustrated in Fig. 1. The basic idea is to use gcov [3] to obtain the profiling result of C statements during simulation. Then the status of a cache model is updated according to the analysis of the generated execution flow and execution times of each statement with its assembly code generated by the targeted platform assembler. There are three stages in our cache modeling process:

1. Code instrumentation stage to insert a cache update API at the boundary of each updating unit.
2. Profiling stage to collect profiling results of the code during simulation on a host machine.
3. Analysis stage to analyze profiling results with the assembly code in the target platform and update the cache status.

As Fig. 1 illustrates, to be executed in a host machine, we compile an application with GCC coverage parameters. Meanwhile, extra files suffixed by .gcno are generated by GCC with information to reconstruct the basic block graphs and assign the numbers of source lines to blocks. In Fig. 1, we also present a coverage file named fun5.c.gcov, which is generated by GCC profiling tool gcov during native simulation, with the total execution times of each statement and the percentage of executing different control branches. Based on the profiling results, there are still three problems to be solved in the cache modeling method:

1. What kind of cache model should be adopted to detect cache miss/ hit efficiently?

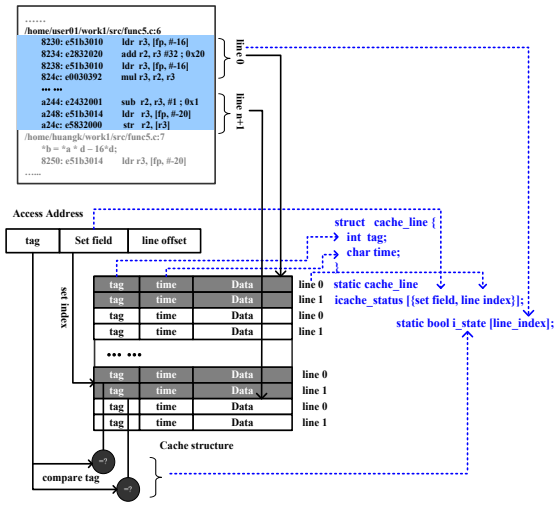

Fig. 2. An example of an instruction cache model

2. How to identify a basic cache updating unit to reduce cache updating frequency.
3. How to update the status of the cache model according to the profiling results.

In this section, we first propose our cache model. Then we review the process of segment identification. The third subsection presents the methods to trace instruction and data addresses, combined with dynamic annotation and static analysis techniques. Finally, we explain the updating mechanism of our cache model.

A. The proposed cache model

Cache memory is composed of multiple lines and each line consists of a certain number of bytes stored sequentially in memory, with a unique identity named tag and its replacement information. Cache lines are arranged in sets, depending on the association degree (e.g., for a 2-way association cache, there are two lines in one set), as shown in Fig. 2. The requested address is divided into three fields: *line offset, set field* and *tag*. The line offset is used to select data from the target line. Its width depends on the line size, e.g., for a cache with 32-byte line size, its width is 5 bits. The set field indicates at which line the set may be allocated. Within the target set, all the lines are equivalent. Thus, to determine if a required line is in a set, the most significant bits of an address are used to compare with all tags. With this mechanism, we can determine whether the access is a hit or a miss. When a memory access causes cache miss, the whole line will be fetched from the next level of memory. The updating scheme of the traditional tag-search based cache model depends on the number of cache lines. Using more lines leads to slower simulation speed, and it becomes worse with the increasing number of processors.

Our generic cache model contains two arrays, as illustrated in Fig. 2. One is declared as *bool* type with the size of the total number of instruction/data lines for the given application software, and each bit indicates whether the corresponding line is in cache. For example, if the code section of an embedded software is 1M and the cache line of the target processor is 32 bytes, the size of the first array is $2^{20}/2^5 = 32$K bits. The

other is an array indexed by the line number of the cache. The array consists of the tag of the instructions in the cache and the replacement information (time) for the corresponding line. The second element could be adjusted according to the replacement strategy of the cache. For example, it is ignored if random or round-robin replacement strategy is applied. However, we have to check the variable *time* when LRU (Least Recently Used) replacement strategy is in use. We do not introduce set index in our cache model, for it is indicated by the line index. Taking a four-way associative cache as an example, the line with index *line_num* belongs to set *line_num*/4.

Although the proposed model is memory consuming, the consumption can be ignored in modern workstations, which are equipped with several gigabytes of RAM. For processors with 32-bit data width, the maximum address range is 2^{32}. Considering a 16KB instruction or data cache of 32-byte line size [14], the maximum size of the first array will be $2^{32}/2^5 = 2^{27}$ bits while the second array is $5 * (16/32) = 2.5$ KB. As the second array also exists in traditional tag-search based cache models, the memory overhead of our model is no more than 16MB. Therefore, even for the extreme condition that all the address space is covered, the amount of memory used for modeling this scenario is still quite far from the maximum capacities of modern workstations. The key advantages of the proposed cache model are as follows. For cache with random or round-robin replacement strategies, the time-consuming tag-search process could be avoided absolutely. And the cache hit is detected just by indexing the *bool* array, which always consumes constant time. However, when LRU is the selected strategy, we still have to search across the lines of the set to update the correct LRU information. If the status of a cache model is updated frequently, it will still be time-consuming for LRU cache replacement strategy. To reduce the frequency of cache updating, we adopt a segment-based cache updating strategy.

B. Segment-based cache updating strategy

We have proposed a segment-based cache updating method to reduce the updating frequency of cache without accuracy loss [12]. In this model, an application is divided into segments. Then the execution flow of statements contained in a segment could be analyzed statically according to the profiling result. The strategy allows cache effects of multiple statements to be applied in bulk.

The identification of a segment depends on the control flow of the corresponding C codes, which is either explicit or implicit. An explicit/implicit control block is composed of a set of basic blocks, whose size is constrained by the control structures in the C code. For example, the *for* loop in the code of Fig. 3 creates an explicit control block. And the two basic blocks involved in the *if-else* branch build up an implicit control block. Roughly speaking, the size of a segment is not bigger than the size of cache. A group of explicit control blocks can be in one segment if the sum of their sizes is smaller than the size of cache. For an implicit control block, it can be added to an existing segment if the size of the resulted segment is smaller than the size of cache. Otherwise, a new segment for this control block or a set of new segments for every basic block in this implicit control block will be created.

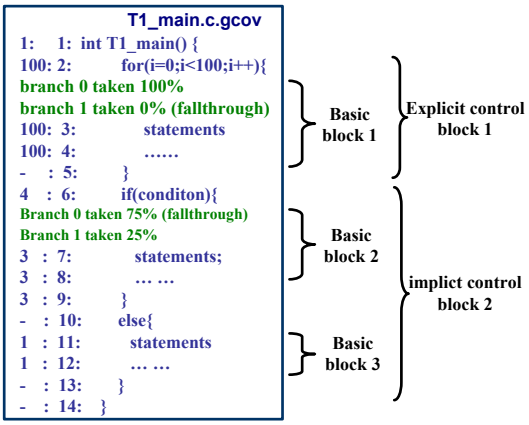

Fig. 3. An example of basic and control blocks

C. Tracing instruction and data addresses

In native-simulation, the source code is compiled with a host compiler. Different compiler optimization strategies and processor instruction sets may make the generated assembly code quite different from the real one running on the target platform. Therefore, to observe the exact instruction or data fetching behavior during native simulation, we compile the source code with two compilers: one is the host compiler to generate an application executable binary file for the native simulation and profiling; the other is the target cross-complier to generate a target assembly code. For instructions, the program counter is amenable to be analyzed statically based on the assembly code from the target platform. Using the target assembly code, we can easily find the relationship between a statement and its corresponding instructions, including the real addresses of the instructions in the target platform. In the example of Fig. 4, file p2_sw.asm shows the correspondence between C statements and their target assembly code. With this relationship, we can model the instruction memory access behavior on the target platform according to the profiling result.

Different from instruction cache modeling, data cache is actually more difficult to model. Tracing the address of a data variable is not easy, because it heavily depends on the dynamic behaviors with less regularity. The key idea to model data cache for native simulation is to derive the addresses of data variables to trace the data memory access behavior in the target platform. Source code annotation techniques are widely adopted to trace variables' addresses during native simulation, to indicate the type of data access and transfer the address to the cache model [7]. Whenever an annotation functions is called, the data cache model will be updated. However, this technique is not fit to segment-based cache updating model, for cache is only updated after the execution of a segment completes. Apart from that, the updating will be time-consuming for the following two reasons: 1) increased code size, where the code size after annotating will be several times bigger than the original; 2) frequent cache updating, where the cache model should be updated every time when data memory access happens.

To trace the addresses of data variables for our cache model, we further introduce a static analysis and dynamic annotation mixed method. According to the ELF standard [4], data allocation in the memory is easy to derive. Global variables are

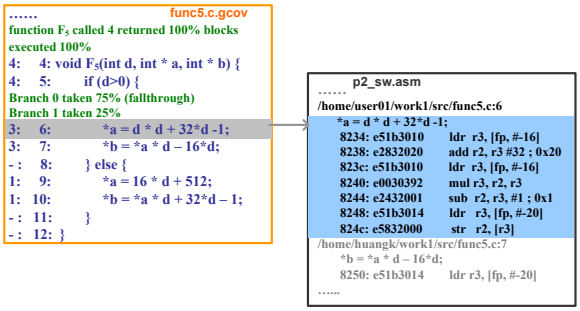

Fig. 4. Analyzing instruction addresses

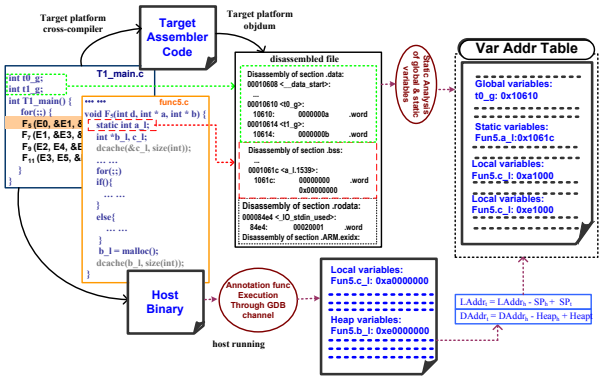

Fig. 5. Address tracing process of data variables

mainly allocated in *data*, *rodata* and *bss* sections. Local variables are allocated in the *stack*, and dynamic data are stored in the *heap*. Global or static variables are allocated statically during program code compilation. We can obtain their addresses by analyzing the disassembled file generated by target processor objdump tool, as shown in Fig. 5.

A local variable is declared in a function or a block and given in a local scope. The address of a local variable is allocated dynamically from the stack according to program running status. It is not possible to derive the absolute addresses of these variables directly from static analysis. Even worse, dynamic data is allocated from and released to the heap by *malloc* and *free* functions respectively during program running, and thus no regularity can be followed. Due to the complicity of heap, few performance estimation related works have considered it.

The idea of our dynamic annotation and static analysis combined address tracing method is as follows. For global or static variables, we establish a variable address table of the target platform by analyzing the cross-compiler ELF file statically. However, for dynamic variables allocated from stack or heap, we insert annotation functions into GCC debugger script to trace the offset address of these variables. In GCC debugger script, we set breakpoints where local variables are defined or memory is allocated for dynamic data. Then the debugger can print the addresses for these variables. The whole process is illustrated in Fig. 5. We use annotation functions to establish the address table of dynamic variables by printing the variable hierarchy and their addresses in the host machine during native simulation. For example, for a local variable *var* defined in *Fun5* with address 0x8a0000, the printing format is $Fun5.var : 0x8a0000$. Though the absolute addresses of the

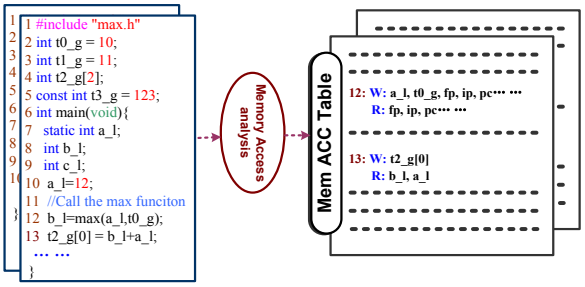

Fig. 6. An example of basic and control blocks

host machine do not match those in the target platform, we can maintain the offset of the variables within each section. To keep the address offset in the host machine and its target platform the same, it is necessary to ensure that the types of variables in the host machine have the same size as those in the target. Otherwise, a slightly modification of the source code is necessary to covert the variable type. For example, if the type *long* in host machine is 64 bits while in the target processor it is 32 bits, the variable with type *long* should be converted to *int* type. With the printed result of annotation functions through native simulation, we can compute the addresses of the target platform by the following two formulas:

$$LAddr_t = LAddr_h - SP_h + SP_t \quad (1)$$

$$DAddr_t = DAddr_h - Heap_h + Heap_t \quad (2)$$

where $LAddr_t$, SP_t, $DAddr_t$ and $Heap_t$ represent the addresses of local variables, the stack pointer, dynamic data and heap base in the target platform respectively, while $LAddr_h$, SP_h, $DAddr_h$ and $Heap_h$ represent the corresponding variables in the host machine.

Besides the variable address table, we have also established a memory access table according to memory visits of each statement, as shown in Fig. 6. Before native simulation, each statement is analyzed statically to find all variable operations, each of which indicates a cacheable load or a store operation. A writing access is performed when a variable is on the left side of an equal expression (=, +=), or when there is an operator such as $++$ or $--$, or a function calling where stack push happens. A read access happens when a variable is on the right side of an equal expression, or comparing expression ($>, <, \geq$), or exiting a function where stack pop happens. Meanwhile, fp, ip, lr, and pc are system variables and should be reserved and recovered when calling and exiting functions respectively. The types of system variables and their corresponding operations are directly related to the type of a processor.

D. Cache updating

When a cache_up function is called in the profiling process, we need to update the cache model, and count the hit times of each instruction. The cache updating process consists of three stages: 1) Collect execution times of each statement by comparing current .gcov files with the one when the previous cache_up function was called. 2) Analyze the execution flow of statements in the segment according to the .gcov files. 3) Extract the target-platform assembly code of each statement for instruction cache, and the memory access information of each

TABLE I
DATA CACHE MISS COMPARISON BETWEEN DIFFERENT MODELS

cases	Data cache misses				
	ISS-based	Search-based	Annotation -based	Proposed method	Error(%)
Bubble 1000	127	127	126	119	6.2%
Bubble 10000	5199310	5207383	5207383	4980341	4.2%
Hanoi	38	44	44	40	5.2%
Factorial	375	500	500	410	9.3%

statement for data cache. And then update the instruction and data cache model respectively according to the execution flow.

To model instruction cache, both *i_state* and *icache_state* arrays are declared globally as static variables to remark the statuses of each instruction line in an application software and the instruction cache model respectively in cache update main code. In the initialization of a profiling task, all bits of the *i_state* array are initialized to be false while the time field of *icache_block* is initialized to zero. During simulation, when cache_up is invoked, the corresponding assembly code is analyzed to determine whether the cache is hit. This task is performed by checking the array *i_state* with the line address of the instruction. If a cache miss happens, both *i_state* and *icache_state* arrays are updated. If some cache line is empty in the set mapped by the instruction, we store the tag of the new instruction in the *icache_state* array directly, and change the bits of the corresponding *i_state*. Otherwise, one line in the cache will be replaced by the new with the replacement strategy. As most of the cache accesses result in a hit in normal operations such as loops and sequential statements, this solution minimizes the cache modeling overhead while keeps the estimation accuracy.

Data cache modeling in native simulation is similar to the instruction cache described above. Two arrays named *d_state* and *dcache_state* are declared globally as the static variables. *d_state*, indexed by the line addresses of variables, is used to indicate whether the variable is hit in the cache, while *dcache_state* is used to indicate the status of each cache block.

III. CASE STUDIES

To demonstrate the advantages of the proposed cache model, we take both small benchmarks (Bubble, Hanoi and Factorial) and a real application (H.264 decoder) as case studies. The simulation results have been compared with those from CK610 ISS, and results with ARM920T reported in [6, 7] and [14], to illustrate the efficiency of our techniques. The target processor CK610 includes 16KB instruction and data caches respectively with 2-way association and 16-Byte line size.

As small benchmarks are especially fit to analyze the corner cases, we first show the experimental results on data cache comparison between our techniques and others in Table I. The column for ISS (ARM920T Instruction Simulator) lists the accurate results, and the two subsequent columns present the results listed in [7] and [14], where the search-based method is traditional, and the annotation-based method is proposed in [7] and [14] (The two papers present same results). The code sizes of the benchmarks are so small that all the instructions are in the cache after the first execution.

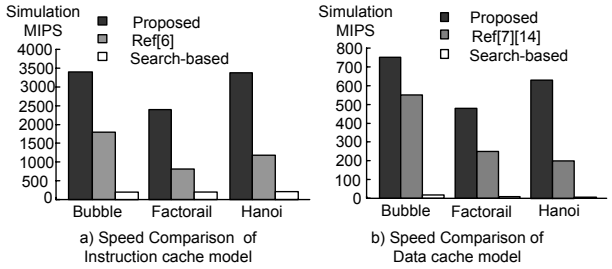

Fig. 7. Speed comparison between different techniques

In Table I, we can observe that for the bubble benchmarks, the annotation-based method has higher accuracy than ours. The reason is that our cache model is updated only after the execution of a segment, and the branch execution flow is analyzed statically, which will be different from the real one. For the annotation-based method, its cache model communicates with the annotation function whenever memory access happens during the native simulation. However, for the last two benchmarks, the accuracies of the annotation-based method reduce greatly with 25% and 33.3% error rates respectively, while the error rates of our model are only 5.2% and 9.3% respectively. The reason is that Hanoi and Factorial have a large percentage of function calls that cause a large amount of stack push/pop operations. We have considered both system variables (i.e., fp, ip, lr and pc) and formal parameters in our model, which are ignored in [7] and [14].

To compare simulation speed, we have executed the benchmarks with three techniques: search-based, annotation-based [6, 7, 14] modeling and our method. In Fig. 7, we show the speed comparison between the three techniques. The speed is described in simulation MIPS (Million Instructions Per Second, Millions of Instructions of the target platform could be simulated per second). In Fig. 7, we find that our cache model can achieve 2.5 times average speedup compared to the instruction cache model proposed in [6], and 2 times average speedup compared to the data cache model proposed in [7] and [14] respectively. All the models can avoid tag-search process for cache hit detection, which accelerates the speed greatly, compared to the search-based model. However, the instruction cache update in [6] is based on basic block, while the data cache proposed in [7] and [14] is updated whenever the memory access happens, both of which cause lots of simulation overhead for communication between cache models and the application. In our cache model, as the code sizes and address ranges for variables in all benchmarks are smaller than the sizes of instruction and data caches respectively, the whole routine is treated as a segment. Therefore, both instruction and data cache models are updated after the execution of the whole routine, and there is almost no overhead on simulation time.

To further analyze the efficiency of our cache model for different cache architectures, we choose a 30-frame Foreman QCIF H.264 decoder on a 4-processor hardware platform as the case study with different cache configurations, i.e., CK610 ISS, the model proposed in [12] and our model respectively. As shown in Fig. 8, the hit rate of instruction cache obtained from our model is almost the same as that from ISS, which means

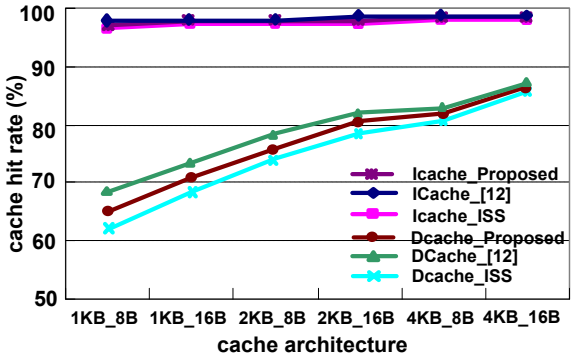

Fig. 8. Instruction and data cache hit rate comparison for different cache architectures

that our instruction cache model provides accurate estimation on instruction cache behavior. However, compared with ISS, the hit rate error of data cache estimated by our model is almost 4% in the case that CK610 processor is equipped with 1 KB cache and 8-Byte line size (1K_8B). This error is reduced to less than 1% if the cache size becomes 4 KB and its line size is 16 Bytes (4K_16B). The reduction comes from the increased size of cache. The average error rate of our model is reduced by half, compared with that of the model in [12]. Overall, the data cache hit rate curve of our model is similar to that of ISS, which indicates that our model is able to present the exact changing trends of cache performance through native simulation.

IV. Conclusion

We have presented a fast and accurate cache modeling method, which combines dynamic annotation and static analysis on the profiling results from native simulation. Meanwhile, we have considered more details in data cache model to obtain accurate cache hit information. In the proposed cache model, for global or static variables, we establish a variable address table of the target platform by analyzing the cross-compiler ELF file statically. For local variables and dynamic data, annotation functions are inserted to GCC debugger script to trace the offset address of local variables. Experimental results show the advantage of our model in simulation speed and estimation accuracy. For the future work, we will further investigate the techniques to improve the accuracy of cache model, and cache updating strategies to accelerate simulate speed with less loss of accuracy.

Acknowledgements

This work is supported in part by National Science Foundation of China under Grant No. 61100074, National Science and Technology Major Project of China under Grant No. 2012ZX01039-004 and Fundamental Research Funds for the Central Universities.

References

[1] ARM, Inc. RealView MaxSim. http://www.arm.com/products/DevTools/MaxSim.html.

[2] Coware, Inc. ConvergenSC. http://www.coware.com/S.

[3] GNU, GCOV-a test coverage program. http://gcc.gnu.org/onlinedocs/gcc/Gcov.html.

[4] TIS committee, Executable and Linkable Format, version 1.2. http://www.parolamia.eu/ELF.pdf.

[5] L. Benini, D. Bertozzi, A. Bogliolo, F. Menichelli, and M. Olivieri. Mparm: Exploring the multi-processor soc design space with systemc. *J. VLSI Signal Process. Syst.*, 41(2):169–182, Sept. 2005.

[6] J. Castillo, H. Posadas, E. Villar, and M. Martinez. Fast instruction cache modeling for approximate timed HW/SW co-simulation. In *Proceedings of the 20th symposium on Great lakes symposium on VLSI*, pages 191–196, 2010.

[7] L. Diaz, H. Posadas, and E. Villar. Obtaining memory address traces from native co-simulation for data cache modeling in systemc. In *Conference on Design of Circuit and Integrated Systems*, 2010.

[8] P. Gerin, M. M. Hamayun, and F. Pétrot. Native mpsoc co-simulation environment for software performance estimation. In *Proceedings of the 7th IEEE/ACM international conference on Hardware/software codesign and system synthesis*, pages 403–412, 2009.

[9] A. Jerraya, H. Tenhunen, and W. Wolf. Guest editors' introduction: Multiprocessor systems-on-chips. *Computer*, 38(7):36–40, 2005.

[10] A. A. Jerraya and W. Wolf. Hardware/software interface codesign for embedded systems. *Computer*, 38(2):63–69, Feb. 2005.

[11] C. Kirchsteiger, H. Schweitzer, C. Trummer, C. Steger, R. Weiss, and M. Pistauer. A software performance simulation methodology for rapid system architecture exploration. In *Electronics, Circuits and Systems*, pages 494–497, 2008.

[12] D. Ma, R. Yan, K. Huang, M. Yu, H. Ge, X. Yan, and A. Jerraya. Performance estimation techniques with mpsoc transaction-accurate model. *Transaction on Computer-Aided Design of Integrated Circuits and Systems*, to appear.

[13] A. Pedram, D. Craven, and A. Gerstlauer. Modeling cache effects at the transaction level. In *Analysis, Architectures and Modelling of Embedded Systems*, volume 310, pages 89–101. 2009.

[14] H. Posadas, L. Daz, and E. Villar. Fast data-cache modeling for native co-simulation. In *16th Asia and South Pacific Design Automation Conference*, pages 425–430, 2011.

[15] J. Schnerr, O. Bringmann, A. Viehl, and W. Rosenstiel. High-performance timing simulation of embedded software. In *Proceedings of the 45th annual Design Automation Conference*, pages 290–295, 2008.