Performance Estimation Techniques With MPSoC Transaction-Accurate Models

De Ma, Rongjie Yan, Kai Huang, Min Yu, Siwen Xiu, Haitong Ge, Xiaolang Yan, and Ahmed Amine Jerraya

***Abstract*—Efficient design of multiprocessor system-on-chip (MPSoC) requires early, fast, and accurate performance estimation techniques. In this paper, we present new techniques based on fine-grained code analysis to estimate accurate performance during simulation of MPSoC transaction accurate models. First, a GCC profiling tool is applied in the native simulation process. Based on the profiling result, an instruction analyzer of the target CPU architecture is proposed to analyze the cycle cost of C code under estimation. In addition, a memory analyzer is used to further estimate memory access latency including both instruction/data cache time cost and global memory access cycles. Both data and instruction cache models are proposed to estimate cache miss penalty, and a segment-based strategy is adopted to update the cache models more efficiently. Furthermore, an equalized access model is presented to imitate the memory access behavior of processors for estimating global memory access latency caused by bus contention and memory bandwidth. We have applied these techniques on an H.264 decoder application with different hardware architectures. The experimental results show that applying these techniques can obviously improve estimation accuracy of transaction accurate models close to that of the virtual prototype models, with a tolerable overhead on simulation speed.**

***Index Terms*—Instruction, memory, multiprocessor system-on-chip (MPSoC), performance estimation, profiling, transaction-accurate model.**

I. Introduction

THE FAST increase of embedded applications makes heterogeneous multithread multiprocessor system-on-chip (MPSoC) more attractive to embedded system designers. The integration of more processor components brings high performance with concurrency capability and long-market period with flexible programmability [1], [2]. Because MPSoC design is naturally processor-centric, and thus software-centric, the most difficult design challenge in the programming model is to map application software into efficient hardware implementations [3]. The work in [4] introduces a feasible solution of a programming model with multiple levels of abstraction ranging from very abstract, specification-oriented models to very concrete, cycle-accurate models. As an important abstraction model, the transaction accurate (TA) level of modeling is thought to be a solution to achieve a good tradeoff between result accuracy and time cost, which also helps to find out the best matches between hardware and software to improve the whole system performance [5], [6].

A TA model details the local architecture of each subsystem in MPSoC and makes the communication protocol explicit [7]. It allows us to estimate the performance of the whole system through hardware and software cosimulation. As shown in Fig. 1, a software stack executable binary is built on a host machine by linking the thread codes and main code with an hardware dependent software (HdS) library. For hardware, except for CPUs, all other components are implemented with cycle-accurate models in SystemC, making use of a bus functional model (BFM) and Linux shared memory (IPC Linux shm) for the interaction, data and synchronization exchange between hardware and software elements. Execution time between two read/write operations is back annotated to the BFM, and finally calculated into the total clock cycle costs with communication overhead.

For a timed simulation of a TA model, the execution time (e.g., time1, time2 shown in Fig. 1) is obtained in advance from low-level simulation on a cycle-accurate simulator and statically inserted into the corresponding read/write function during software code generation. However, this static time annotation technique for performance estimation does not consider any variation of execution time when a stimulus is changed [7], [8]. Moreover, the accuracy of the performance result depends on the given architectures, i.e., memory architecture, bus protocol, processor architecture, thread mapping strategy, and so on. Thus, lacking the flexibility to efficiently estimate different MPSoC architectures is also a more serious disadvantage in this static technique, which extremely limits the design space. For example, the memory architecture is a key factor to decide the data access latency to calculate

Manuscript received October 23, 2012; revised July 2, 2013; accepted July 5, 2013. Date of current version November 18, 2013. This work was supported in part by National Science Foundation of China under Grant 61100074 and Fundamental Research Funds for the Central Universities. This paper was recommended by Associate Editor Y. Xie. (*Corresponding author: Kai Huang*).

D. Ma is with the Key Laboratory of RF Circuits and Systems, Ministry of Education, Hangzhou Dianzi University, Institute of VLSI Design, Zhejiang University, Hangzhou 310037, China (e-mail: madehd@163.com).

R. J. Yan is with the State Key Laboratory of Computer Science, Institute of Software, Beijing 100090, China (e-mail: yrj@ios.ac.cn).

K. Huang is with the Institute of VLSI Design, Zhejiang University, Hangzhou 310027, China (e-mail: huangk@vlsi.zju.edu.cn).

X. L. Yan, M. Yu, and S. W. Xiu are with the Institute of VLSI Design, Zhejiang University, Hangzhou 310027, China (e-mail: yan@vlsi.zju.edu.cn; yumin@vlsi.zju.edu.cn; xiusw@vlsi.zju.edu.cn).

H. T. Ge is with Hangzhou C-Sky Micro-system Company, Hangzhou 310012, China (e-mail: haitong_ge@c-sky.com).

A. A. Jerraya is with CEA-LETI, MINATEC, Grenoble Cedex F38054, France (e-mail: ahmed.jerraya@cea.fr).

Digital Object Identifier 10.1109/TCAD.2013.2275252

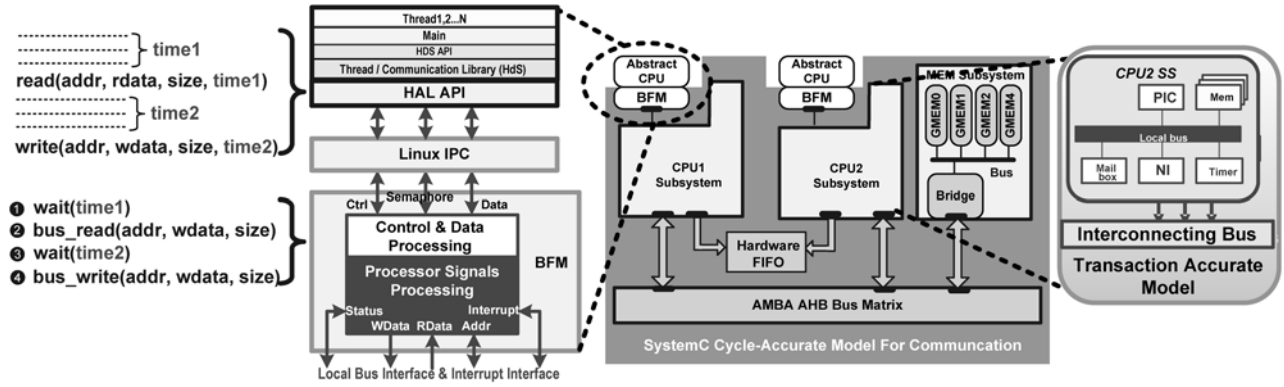


Fig. 1. Hardware and software cosimulation with transaction accurate models.

the execution time. Without considering memory architecture details, the execution time of a given application cannot be evaluated well only with the static time annotation technique. When the wrong execution time is annotated, it further leads to inaccurate communication time even if the communication model is cycle-accurate. Therefore, it is still a challenge to estimate more accurate performance on a TA model while keeping fast simulation speed for more efficient design space exploration.

In this paper, we focus on how to improve the accuracy of TA models with less speed loss. The performance of an application depends on both static and dynamic aspects [9]. Static timing sources, which can be analyzed without simulation, are mainly decided according to the instruction types of the program and memory type of the system. Dynamic aspect relies on various factors, e.g., loops, branch, and cache hit/miss, which can only be measured with simulation. This paper presents new techniques considering both static sources and dynamic factors, to estimate accurate performance based on fine-granularity code analysis during MPSoC TA model simulation. For the static aspect, we use gcov [10], which is a standard utility with GCC, to test code coverage in application software and find out some basic performance statistics. We also take advantage of native simulation to handle dynamic factors. Finally, the analyzing process combines the dynamic factors with static statistics to generate exact performance estimation from TA model simulation, allowing fast and exact hardware and software architecture exploration.

The main contribution of this paper is the introduction of a dynamic simulation and statistic analysis combined method to evaluate the performance of the target MPSoC platform in a TA model. It is used to generate the transaction model with profiling API functions from a Simulink system-level model and measure the performance of the whole system with accurate execution time and communication overhead. The second contribution is to use GCC profiling tool with an instruction analyzer of the target CPU architecture to calculate accurate cycle cost of the given C code during dynamic simulation. The third contribution is to apply a memory analyzer to further estimate memory access latency, including instruction and data cache access time cost. Furthermore, we propose an equalized access model (EAM) to imitate memory access behavior of processors to estimate the global memory access latency caused by bus contention and memory bandwidth. The experimental results with an H.264 decoder application on target MPSoC platforms are adopted to demonstrate the efficiency of the proposed methods.

II. Related Work

The trend of MPSoC architecture is to integrate more heterogeneous processors, which extends the design space greatly. A key step of architecture exploration is to efficiently estimate the performance of an application running on those heterogeneous processors architectures. Current literature offers a large set of references dealing with fast and accurate performance estimation techniques. Most of these techniques can be divided into two categories: static analysis and dynamic simulation. Static analysis is able to provide fast estimation with low-execution effort. There are many analytical techniques based on static analysis of software codes or models, which consider all possible paths in the control flow graph (CFG) and use formal analytical models to represent a system as a network of nodes exchanging streams. They are usually employed to calculate the worst-case execution time (WCET) [11] for real-time systems. The model of Li and Malik [12] computes a tight bound of WCET for the instruction cache for embedded software performance estimation. Even though pure analysis method guarantees system performance, the estimation results obtained by analytical techniques are usually too pessimistic, thus leading to over-provisioning or under-utilization of resource. Simulation-based techniques are widely used for both functional verification and performance estimation. A common approach for execution-driven simulation is to employ a cycle-accurate architecture model with instruction set simulator (ISS) (e.g., ConvergenSC [13], Realview [14], MPARM [15]). The operation of an ISS consists of reading the code compiled for a target platform and executing the instructions by using the target processor model. The ISS model can have several levels of accuracy according to different levels of models, e.g., instruction level model, transaction level model, or register transfer model. Even though execution-driven simulation method is reasonably accurate, it is often too slow to be used for MPSoC design space exploration.

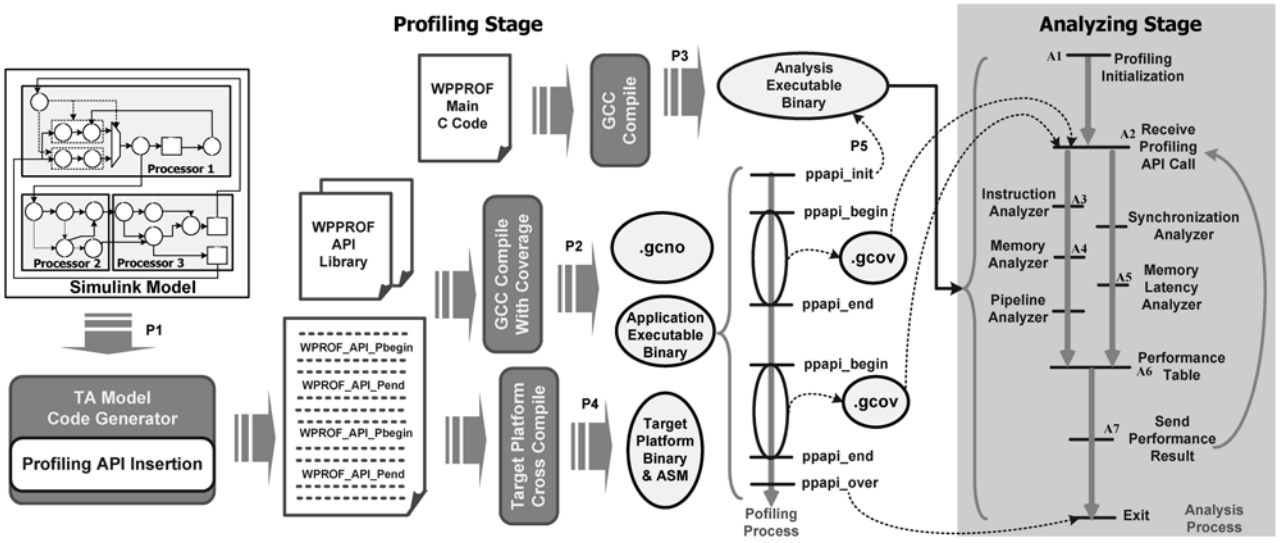


Fig. 2. Overview of the estimation method.

To speed up simulation, some hardware-assisted simulation approaches take the advantage of field-programmable gate array (FPGA) techniques by splitting the tasks for simulation into FPGA and collaborating software modules [16]–[18]. With fast simulation speed, these approaches are able to run an unmodified OS to obtain more accurate performance results of the whole system. However, compared with software simulation approach, a hardware-assisted simulator suffers from hardware limitation (FPGA resources and frequency), and is hard to debug and validate.

Recently, native simulation approaches have been proposed to achieve a good tradeoff among simulation speed, accuracy, and retargeting ability. In native simulation, software runs on the host machine natively and its execution time is calculated through annotation or analysis techniques. Some static annotation-based methods [7] first collect execution samples that contain architectural events (e.g., execution cycle, memory accesses) of each processor under a given architecture configuration. Then, the samples are used as inputs and annotated to native simulation. However, estimation results of these approaches are not accurate in the case of different stimulus inputs. Some improved approaches [19]–[21] insert delay functions into software codes and calculate the total delay during native simulation. These approaches use the assembly code from a target cross-compiler and its processor datasheet to generate delay information for each statement or basic block. Then, the delay information is annotated to the raw software model to generate a delay-annotated software model. They could provide more accurate performance than the static annotation and run faster than ISS. However, the cache and pipeline model is coarse and hardware behaviors are not handled well. And software codes have to be modified largely with fine-grained delay annotation functions.

Our work takes an advantage of dynamic simulation and statistic analysis combined method for efficient performance estimation with high accuracy at low cost of simulation speed. The proposed method is similar to native simulation approaches mentioned above. Compared with previous work, there are only few extra codes added into the original software model for dynamically collecting delay information during native simulation. Meanwhile, two techniques for instruction and memory analysis are used to correct delay result from dynamic information, which improves the accuracy of final estimation on software execution cycles. Furthermore, we feed the code execution cycles to the TA model to make its performance closer to its cycle-accurate virtual prototype (VP).

III. Proposed Estimation Method

To achieve faster estimation speed, we adopt a new native simulation strategy for a TA model to calculate its accurate performance result dynamically based on fine-granularity code analysis. The basic idea is to use gcov [10] to obtain the profiling result of C statements during simulation, and then analyze the generated execution times of each statement with its targeted platform model to calculate the execution time. The counted total cycle of a given code is fed to the TA model in SystemC to estimate the performance of the whole system. Throughout this paper, we use the TA model generated in [5] and [8] to explain how the proposed estimation method works.

The process of execution time estimation consists of two stages (shown in Fig. 2): 1) profiling stage to generate C code with profiling API and collect profiled results of the code during simulation on a host machine, as shown in steps P1–P4 of Fig. 2, and 2) analyzing stage to analyze profiled results according to the architecture of the target platform and calculate the execution time of the code under estimation, as shown in steps A1–A7 of Fig. 2. In the first stage, the code generator of the Simulink-based MPSoC platform [5] generates multithread codes from Simulink model for TA model (Step P1). During code generation, four profiling API functions are inserted into the generated multithread codes to support run-time performance estimation. Function ppapi_init is used to fork a child process for performance profiling.

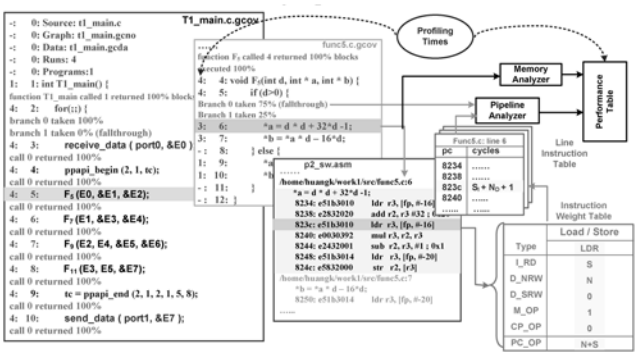


Fig. 3. Work mechanism of an instruction analyzer.

Functions ppapi_begin and function ppapi_end are used as starting and ending points for performance profiling task, respectively. Function ppapi_over is to complete the analyzing process before terminating the whole simulation. To be executed in the host machine, these multithread codes are compiled into an application executable binary with profiling API library by using GCC coverage parameters (Step P2). Meanwhile, extra files suffixed by .gcno are generated by GCC with information to reconstruct the basic block graphs and assign source line numbers to blocks. Besides the application executable binary, an analysis executable binary, which is generated from weight performance profiling code by the GCC compiler, is also necessary (Step P3). It is general for a given hardware platform. The analysis executable binary runs in a new process forked from the parent process that runs the application executable binary when function ppapi_init is called (Step P5). We call the parent process profiling process and the child process analyzing process.

At the beginning of the analyzing stage, some necessary environment variables are configured and initialized (Step A1). Then, the analyzing process keeps waiting for profiling API call from a profiling process. When a profiling API function ppapi_begin is called, the subsequent code executed in the profiling process should be profiled and analyzed to obtain their performance on the target platform. When a profiling API function ppapi_end is called, the statements to be profiled and analyzed are over. As soon as it receives a profiling requirement from a profiling process, the analyzing process starts to analyze the performance on the given portion of the code (Step A2), with the consideration of instruction execution cost (A3) I/D cache miss and hit (A4), pipeline, interthread synchronization, and memory access latency (A5), respectively. The effect of pipeline and synchronization is not taken into account in this paper. There are mainly five steps in the analyzing process.

1) Receive the message of profiling API call and extract the information about which lines to be profiled (Step A2).
2) Analyze the code under estimation (CUE) with its coverage file (.gcov) provided by GCC and calculate its total instruction cost according to the assembly code generated by the target platform cross compiler tool chain (Step A3).
3) Further analyze the latency of memory access, which affects the final performance result greatly. It is necessary to consider the features of cache, local memory and global memory in the target platform (Steps A4 and A5).
4) Obtain the number of total cycles for each statement in the CUE to construct the performance table (Step A6). In this step, the number of total execution cycles is analyzed by considering two factors: instruction weight table and memory latency. More details about these two factors are explained in Sections IV and V.
5) Send the number of total cycles back to the profiling process, then, the analyzing process keeps waiting for new profiling API calls from the profiling process (Step A7).

During native simulation, the cycle counted from an analyzing process is regarded as the execution time for a CUE. The profiling process in native simulation will send the result to the BFM in hardware simulation of the TA model through Linux IPC, as shown in Fig. 1. In this way, the native simulation of application software cooperates with the hardware simulation of a communication model, which combines execution time with communication latency for the exact performance estimation of the whole system.

IV. Instruction Analyzer

An instruction analyzer is used in an analyzing process to calculate instruction cost for a given portion of C code, based on the GCC profiling coverage result. The idea of the instruction analyzer is to calculate the cycle cost of each instruction by considering the number of execution times from profiling results and the cost for a single execution according to the different target platforms. In this section, we first show how to obtain profiling information from a profiling process. Then, we explain how a weight table can be constructed according to different types of instructions and processors. Finally, we propose a formula for instruction cycle cost computation and explain its limitation.

During a profiling process, some coverage files suffixed by .gcov are generated by GCC profiling tool gcov. As shown in Fig. 3, the files contain the basic performance statistics, e.g., execution times of each statement, taken percentage of branches. A gcov file parser is designed to obtain the execution times for each line of the code within the given range between profiling API functions ppapi_begin and ppapi_end. If the current statement calls another function, the parser also parses the gcov file of this function to collect the instruction results for it. In the example of Fig. 3, function F5 is called at the fifth line in the file named T1_main.c. In its gcov file, we can find that this function is already called four times during the simulation. From the gcov file of func5.c, we can know the execution times of each line in function F5. For example, the sixth line of func5.c has already run thrice.

Before running simulation, an assembly code analyzer is used to generate an instruction table for each line according to the instruction weight table of the target processor. An example for ARM7TDMI processor is shown in Fig. 3. Both T1_main.c and func5.c are compiled into a binary code named p2_sw.bin

TABLE I
EXAMPLE OF THE INSTRUCTION WEIGHT TABLE FOR ARM7TDMI PROCESSOR

Type	Data Processing					MSR MRS	Load/Store						B/BL	MCR/ MRC
	ALU	MUL	MLA	MULL	MLAL		LDR	STR	LDM	STM	SWP	STRC/LDC		
I_RD	S	S	S	S	S	S	S	N	S	N	N	N	a(N+S)+S	N/S
D_NRW							N	N	N	N	N	N		
D_SRW									(n-1)S	(n-1)S	S	(n-1)S		
M_OP		4	5	5	6		1		1					b/b+1
CP_OP												C		C
PC_OP	N+S						N+S		N+S					

running on the processor 2. File p2_sw.asm, the disassembled code from p2_sw.bin, shows all instructions and the program counter (PC) for each statement of the code. Therefore, we can establish the relation between each statement of the source code and its corresponding instructions, and further analyze the performance of each statement on the target processor.

To calculate the exact cycle cost of each statement, we have used the instruction weight table of a given processor and considered different kind of performance factors caused by instruction, pipeline, and memory latency. Table I is an example of instruction weight table for ARM7TDMI processor (without I/D cache). It lists cycle cost of those frequently used instructions in ARM instruction set. We have defined six operations, which impact the cycle cost of one instruction, respectively. Slash "/" separates two possible choices.

Type I_RD is used to describe the cost of prefetching an instruction. In modern processors, each instruction is usually fetched at the first pipeline stage, and meanwhile the previous fetched instructions are decoded or executed. If an instruction cannot be fetched in one cycle, pipeline stalls may happen and extra cycle cost should be taken into account. I_RD is regarded as the basic element of cycle cost of one instruction. For a single memory access, we define N as the cycles for a nonsequential data access of a single memory operation, and S as the cycles for a sequential data access of continuous memory operations. The values of N are different from those of S for special bus and memory access. For example, DRAM may take more cycles to access an address not related to the previous address. The conditional branch instruction, like B or BL of ARM7 processor, may take more cycles if branch prediction fails. In Table I, parameter a is equal to 1 if branch prediction fails. Otherwise, it is equal to 0.

Types D_NRW and D_SRW are defined as nonsequential and sequential access to data memory, respectively. The cycles of multiple-load/store instruction, such as LDM or STM of ARM7 processor, involve one nonsequential cycle cost for the first data access and $n-1$ (n is the total load/store times in this instruction) sequential cycle cost for the subsequent data access. Type M_OP represents those instructions with multiple-cycle execution. For multiply instruction of ARM7 processor, its execution cycle depends on the input data, and it requires four cycles to get the final results in the worst case. Type CP_OP is used to describe coprocessor instructions. Its cycle cost is recorded as C for different coprocessor operations.

Besides the above operation types, PC_OP is optional to those instructions that use PC as a destination. In these instructions, PC will be changed and a new instruction will also be prefetched. We should add cycles N for the changed PC load and the following cycles S for the next PC prefetching.

With the disassembled codes of a target application, we can generate a line instruction table based on an instruction weight table. The cycle cost of each instruction in this table should include both the essential cycle cost of the instruction that is obtained from Table I and the register hazard that can be known by analyzing source/destination registers of neighboring instructions. Therefore, the cycle cost of one instruction is computed according to (1). In modern processor architecture, many innovative techniques, e.g., reservation station, are adapted to avoid register hazard. So the cycles caused by register hazard can be ignored to simplify the estimation method. The values of other elements (such as S_I, S_D, N_I, and N_D) for T_{instr} will be further analyzed by the memory analyzer that will be introduced in the next section.

$$T_{instr} = T_{I\_RD} + T_{D\_NRW} + T_{SRW} + T_{M\_OP} + T_{CP\_OP} + T_{PC\_OP} + T_{REG\_HAZARD} \quad (1)$$

There is a restriction on this instruction analyzer: To get the exact mapping between C statements and their assembly codes, GCC compiler optimization is not suggested if high accuracy of performance estimation is required. With our experience gained from the experiments, the estimation error caused by GCC optimization is about 3–54% according to different optimization levels (-O1, -O2, -O3).

V. MEMORY ANALYZER

Even equipped with powerful execution units and deep pipeline architecture, memory latency still impacts processor performance greatly. We have to consider this issue to give an accurate performance analysis. Memory access from one processor consists of instruction fetch and data access. Therefore, memory latency depends on the implementation of instruction and data memory. In multicore systems, local memory is usually designed to tightly work with one processor while global memory is shared by all processor subsystems. Moreover, instruction/data cache is widely used in modern processors, taking advantage of data reusability to improve its memory access performance with less memory latency.

To obtain accurate computation cycle cost, we use a memory analyzer to calculate total delay caused by memory latency resulting from different memory implementations. For instruction fetch, a memory analyzer can preanalyze link file

of C compilation to acquire the information about memory allocation for all instructions. For data access, the memory address of load/store instruction is related to different sources: stack/frame pointer, general-purpose register, global variable, local variable, and pointer variable. The details about how to obtain the address of instruction and data memory access will be explained in Section B and C, respectively. We assume that data located in thread-local memories is not shared among threads except through explicit handshakes, and the interprocess communication data are only stored in global memory.

Nowadays, almost all advanced processors take an advantage of cache to overcome the gap between processor cycles and instruction/data memory access time. It is obvious that cache has a strong impact on the whole system performance. The following subsections first present segment-based cache updating strategy for dynamic factors of a cache model, and then describe how to estimate the performance of instruction and data memory access while taking cache into account. Finally, an equalized access model is introduced to consider global memory access delay caused by bus contention and memory bandwidth.

A. Segment-Based Cache Updating Strategy

Cache behavior should be modeled to obtain more accurate performance estimation. Therefore, the execution flow of target codes must be traced to analyze their cache behavior. However, the code profiling method only shows total execution times of each statement without execution flow information. Therefore, cache updating strategy is a key technique to estimate cache performance exactly and efficiently. We propose a segment-based cache updating model to reduce the updating times without accuracy loss. In this model, the application is divided into segments. It allows cache effects of multiple statements to be applied in bulk.

To identify a segment, we first recap the concept on basic block. In structured programming, C programs are organized by three elemental and hierarchical flow structures: sequence, selection, and iteration. A basic block is internally composed of sequential statements, with iteration or selection statements being its boundaries. As Fig. 4 shows, basic blocks 1, 2, and 3 are bounded by the control flow statements "for" and "if ... else ...", respectively.

The control flow statements in C/C++ language can be divided into two types: implicit control and explicit control. A control flow is explicit if its behavior can be analyzed statically with the code coverage result. Otherwise it is implicit. Taking the "for" loop in Fig. 4 as an example, file T1_main.c.gcov shows that the loop is executed 100 times, and the percentage of taking the branch is 100%. So all of the statements contained in the "for" iteration are executed 100 times in sequence. On the other side, though the execution times of the "if ... else..." control flow is also given, the execution flow of its statements is still unknown. With these observations, the control flow statements in C/C++ language are further divided, as shown in Table II.

A control block is either a set of basic blocks belonging to an implicit control flow, or a set of basic blocks bounded

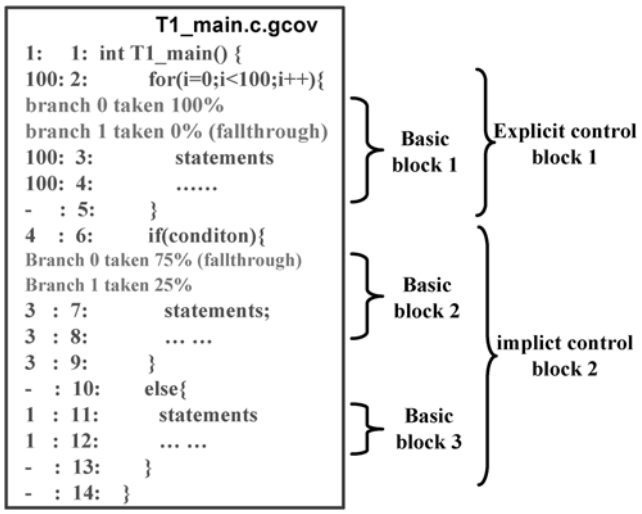


Fig. 4. Example of basic and control blocks.

TABLE II
EXPLICIT AND IMPLICIT CONTROL DIVISIONS

Type	Sub type	C/C++ statements
Explicit	Iteration	For, while, do…while
	Branch	Function call ,goto
Implicit	Branch	Break, continue, return,
	Selection	If…[else], switch

by the same explicit control flow. The type of a control block is the type of its control flow statements. For example, basic blocks 2 and 3 form an implicit control block, for they are bounded by the same "if ... else ..." statement; while the "for" loop in Fig. 4 is an explicit control block.

With the concepts explanation above, Algorithm 1 presents the process of identifying a segment shown in Fig. 5. If a segment contains implicit control blocks, its type is also implicit (marked by 1). Otherwise it is explicit (marked by 0). In Algorithm 1, we assume that the code of application software has been divided into a sequence of control blocks, each of which is composed of a set of basic blocks. First, an empty explicit segment s is initialized. Then, the control blocks in C are parsed in sequence. For an explicit control block c_i, if the type of the current segment s is explicit, c_i is merged to s directly. Otherwise, we have to consider the size of the cache, and create a new segment for c_i if the addition of c_i makes the size of s greater than the cache size (line 4). If c_i is implicit, and the size of s with c_i is less than the size of cache, c_i is merged to s and the type of s is modified to one for containing c_i. Otherwise, we also need to create a new segment. If the size of c_i is less than that of cache, c_i is added to the new segment (line 7). Otherwise, we have to create a new segment for each basic block of c_i (line 8).

From the above analysis, segments are formed by following either of the rules.

1) The execution flow of the statements contained in a segment could be analyzed statically according to the profiling result.
2) The size of the instructions (for instruction cache) or data (for data cache) in a segment should be small

Algorithm 1: Generating segments

```
input : a piece of code C = {c_i = {b_j}_{1≤j≤K_i}}_{1≤i≤n}, with n control
        blocks, where K_i is the number of basic blocks in c_i
output: a list of segments S = {s_i}_{1≤i≤M}
begin
1   S = null;
2   new(s); s.type = 0;
    for (i = 1; i ≤ n; i++) do
        // for an explicit control block
        if c_i.type == 0 then
            if (s.type == 0) ||
            (s.type == 1 && s.size + c_i.size < cachesize) then
3               s.add(c_i);
            else
4               S.add(s); new(s); s.type = 0; s.add(c_i);
        // for an implicit control block
        else
            if s.size+ c_i.size < cachesize then
5               s.type = 1; s.add(c_i);
            else
6               S.add(s); new(s); s.type = 1;
                if c_i.size < cachesize then
7                   s.add(c_i);
                else
                    for (j = 1; j ≤ K_i; j++) do
8                       S.add(s); new(s); s.type = 1; s.add(b_i);
                    if i < n then
9                       S.add(s); new(s); s.type = 0;
10  S.add(s);
end
```

Fig. 5. Pseudo-code for generating segments.

enough to fit into the cache. For the first rule, both the instruction and cache models are treated as the same. For example, given a loop with only sequential statements, no matter how large its body is, we consider the effects of both its data and instruction cache after the termination of the loop. The reason is that the behavior of the loop could be analyzed statically.

For the second rule, size calculation for a control block is quite important. According to the unique memory addressing characteristics of instruction and data, they are considered in different ways. The effect of the second rule for the instruction cache is obvious. For example, in a loop with a small enough body, the loop body will be in the cache when the loop terminates. And each statement in the loop body will have experienced one instruction cache miss, no matter what kinds of statements the segment contains. Thus, it is not necessary to calculate the effect of instruction cache miss for each instruction. For data cache, the address of variables is considered for size calculation for the second rule (see Section V-C for detail). To make sure that no data cache replacing activity happens when a segment is executed, we introduce two constraints for a segment: 1) none of the two global variables within a segment have the same cache index to guarantee that at least one position is reserved in each cache set for local variables (see the fifth paragraph of Section V-B for detail), and 2) total size of all variables in a segment is no more than ((way number - 1)/way number) of the cache size. Since local variables are always allocated in sequence, this constraint also ensures that there is enough space for global variables in each set. For example, suppose that the size of local variables in each control block is 1 KB in Fig. 4, and global variables

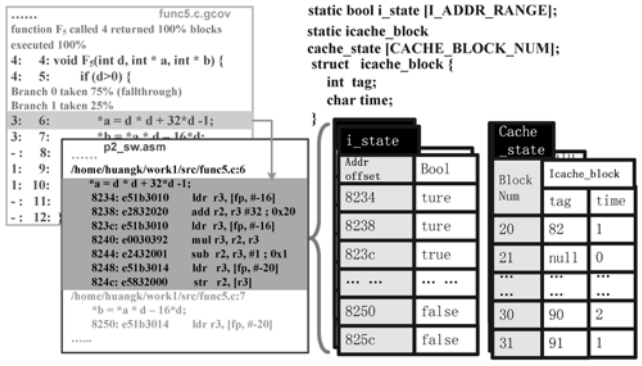


Fig. 6. Example of an instruction cache model.

of each block have unique cache indexes, if the cache size is 2*(1 KB + 1 KB) with the two-way association, the two control blocks 1 and 2 could be merged to form a segment.

With the identification of a segment, cache updating is performed at segment level and cache updating API functions are inserted to instrument the cache model updating during native simulation. For the same codes, data and instruction cache may have different segment sizes. Therefore, to start up a cache updating task in an analyzing process, there are two kinds of cache_up APIs for data and instruction segments, respectively. There are also two objectives for the cache updating task: one is to update the cache status in the cache model; the other is to calculate the hit times of the instruction or data cache. Thus, taking advantage of segment-based cache updating strategy, we can simulate cache behavior exactly with less speed loss than those previous strategy based on basic block or even statement [22], [23]. Our experimental results show that the simulation speed using segment-based cache updating strategy is on average 2.3× faster than block-based strategy while keeping the same accuracy.

B. Instruction Cache Analyzer

For the performance of instruction cache, three elements are taken into account: instruction cache model, the address of each instruction and L2 memory access latency caused by L1 cache miss. In order to observe the exact instruction fetch behavior during native simulation, the source code is compiled by two compilers: one is the host compiler to generate an application executable binary file for the profiling process; the other is the target cross-complier to generate a target assembly code to be analyzed. 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. 6, file p2_sw.asm shows the correspondence between C statements and their target-platform assembly codes. With this relationship, we can model the instruction memory access behavior on the target platform during native simulation.

The instruction fetching cycle is calculated according to (2). t_{memory} is the memory access cycles of fetching L2 instruction which depends on the type of L2 memory. And t_{cache} is the cycle cost for cache access. When a cache miss happens, besides loading the missed instruction, the processor also

prefetches the neighboring instructions into one cache line according to spatial locality. refill_len$_I$ points to the number of loading instructions. These three parameters depend on hardware memory architecture and are fixed before native simulation. Parameter R_{hit} describes one-bit cache hit information. In the case of cache hit, the value of R_{hit} is set to 1, otherwise 0. The value of this parameter is dynamic during native simulation.

$$T_{cycle} = (1 - R_{hit}) * t_{memory} * (refill\_len_I) + R_{hit} * t_{cache} \quad (2)$$

With (2), we can easily calculate the instruction fetch time used in an instruction analyzer. For instance of processor CK510, a seven-stage pipeline is adopted such that for an instruction, its fetching time is overlapped with other stages when cache is hit; while for a miss, the delay is N + (refill_len$_I$ − 1)*S.

When a cache_up function is called in the profiling process, the cache model will be updated and the hit times of each instruction will be counted. The cache updating process consists of three stages.

1) Collecting execution time of each statement by comparing current .gcov files with the one when the previous cache_up function was called.
2) Analyzing the statement execution flow.
3) Getting the target-platform assembly code for each statement and updating the cache model according to the execution flow.

Cache memory is composed of lines. Each cache line consists of a certain number of bytes stored sequentially in memory, with a unique identity named tag and replacement information. Cache lines are arranged in sets, depending on the association degree (i.e., for the two-way association cache, there are two lines in one set). The typical cache model is based on tag-search strategy [24] with a high-simulation overhead. For example, the cache of ARM920T consists of 64 lines in each set, and comparing with 64 tags in the cache model may cause obvious cost overhead on simulation time. The proposed instruction cache model defines one-bit Boolean variable for each instruction line to indicate whether this line is in instruction cache or not. The size of instruction line is the same as that of cache line, i.e., for CKCore the size of cache line is 32 bytes. As Fig. 6 shows, the model consists of two arrays. The first one is declared as bool type with the size of total number of instruction lines for the application software and each bit indicates whether the line is in cache or not. The other is an array indexed by the line number of the cache, and its structure consists of the tag of the instructions in the cache and the replacement information for the corresponding line. The second element could be adjusted according to the replacement strategy of the cache. For example, the element is ignored if random or round-robin replacement strategy is applied and the variable times is used when least recently used (LRU) replacement strategy is in use.

Although the proposed model is memory consuming, the consumption can be ignored in modern workstations, which are equipped with several gigabytes of RAM. The key advantage presented by this model is that the time-consuming tag-search process is avoided and the time consumption of hit detection is always constant. When LRU is adopted as the replacement strategy, we still have to search across the lines of the set in order to update the correct LRU information. But we only need to update the LRU information after completing one segment's execution. What is more, for other cache replacement strategies, i.e., random, robin, the tag search process could be avoided absolutely. The techniques achieve a speed up of one order of magnitude compared to tag-search based techniques.

Both i_state and cache_state arrays are declared globally as static variables in a profiling code. When function ppapi_init is called, all bits of the i_state array are initialized to be false. 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 i_state array with the address offset of the instruction as the index. If cache miss happens, both the i_state array and cache_state array are updated. If some cache block is empty in the set mapped by the instruction, the tag of the new instruction is stored into the cache_state array directly, while the bits of the corresponding i_state are changed. Otherwise, one line in the cache will be replaced by the new with the replacement strategy. As in normal operations such as loops and sequential statements, most of the cache accesses result in a hit, this solution minimizes the cache modeling overhead while keeps the estimation accuracy.

When a cache miss happens, the instructions are fetched from external memory and the delay is determined by three elements: memory type, instruction refilling length, and communication bus state. The memory type and instruction refilling length are easy to determine. But it is more complicated for the bus state. If the fetched instructions are placed in the global shared memory, more than one processor may fetch instructions at the same time when a cache miss happens. In this case, only the processor with the highest priority can fetch the instructions successfully while the others are blocked. In order to reduce the estimation complexity, we assume that no bus contention exists during instruction fetching. The assumption is reasonable because the cache hit rate of modern processors is more than 95% and most instructions are located in local memory owned by local processors.

C. Data Cache Analyzer

To estimate the memory access latency of data cache, we also take advantage of the segment-based cache updating strategy. Different form instruction address that is amenable to be analyzed statically based on the target-platform assembly code, the address of a data variable is more difficult to trace since 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. According to the ELF standard [25], data allocation in the memory is easy to derive for each type of data. Global variables are mainly allocated in data, rodata, and bss sections, local variables are allocated in the stack and dynamic data are stored in the heap. The heap is managed dynamically,

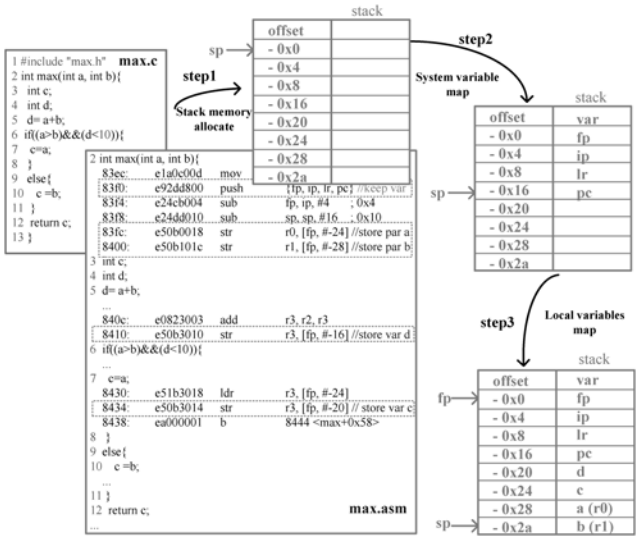


Fig. 7. Example of local variables address offset derivation.

and it is quite difficult to trace its address. So, we did not consider it in the proposed cache analyzer technique. Global or static variables are statically allocated during compilation of program code and it is easier to obtain their addresses from the disassembled file generated by objdump tool.

A local variable is declared in a function or a block and is given in a local scope. The address of a local variable is related to both a stack point (SP) and the offset from the SP. Since the value of the SP is related to running status, the absolute addresses of these variables are impossible to derive directly from static analysis. But it is easy to establish address offset tables for the local variables in functions by analyzing the assembly code statically. In the example of Fig. 7, a function is first parsed statically to calculate the size of total variables, then the size of system variables are added to obtain the total size needed for the function. For example, in ARM9, four words are used to store system variables (i.e., fp, ip, lr, and pc). Second, from the bottom of the stack (SP position), the required memory size is allocated for the function. As Fig. 7 shows, 32 bytes are allocated for function max. Third, the system variables are directly mapped to the stack from the bottom of the stack (low offset), shown in step 2 of the figure. Fourth, the local variables are mapped to the stack from high offset to low offset according to the order of their definitions, shown in step 3. Though different compilers will adopt different local variable allocation strategies, the only difference in the results is the variables allocation order. It is easy to regulate the local variables mapping order in our proposed method according to the selected compiler. The absolute address of a local variable can be inferred by subtracting the offset from the stack point. Moreover, the initialized address of an SP is assigned in the link file during compilation. And it is only changed when a function is called or a block is entered, and it is recovered when the process leaves the block or the function. Therefore, the analyzing process in native simulation is improved to trace the address of stack pointer and derive the stack pointer address when calling the function or entering the segment. For example, in the example of Fig. 7, the address of SP will be decreased by 32 when the function max is entered.

For data cache modeling in native simulation, the method is similar to the instruction cache described in the previous section. Two arrays named d_state and dcache_block_cache_state are declared globally as the static variables. Before native simulation, each statement is analyzed statically to find all variable operations, each of which indicates a cacheable load or a store operation. After that, a memory access table is established corresponding to those memory visits of each statement. When function cache_up is called in a profiling process, the data cache model is updated according to the execution flow of C code.

D. Equalized Access Model of Global memory

Different from local memory or cache, global memory is shared by all the processors. The limited bandwidth of global shared memory leads to the contention of global resource in the case of multiple accesses to the same memory bank from different processors at the same time. The contention is aggravated with the increasing number of processors in parallel and the delay should be calculated into the total memory latency of those failed accesses in the arbitration. However, it is difficult for a TA model to obtain accurate contention results, because the BFM of an abstract processor cannot imitate the memory access behavior of the processor exactly. The key problem to calculate the latency of global memory is how to imitate memory access behavior of each processor as exact as possible. There are two kinds of memory access: instruction fetching and data load/store. For instruction fetching, most modern processors make use of instruction cache (ICache) with more than 95% hit rate. Moreover, each CPU subsystem is equipped with local memory to store those frequently accessed instructions, which may reduce the possibility of instruction fetching from global shared memory. However, the cache hit rate of data cache is extremely low than that of instruction cache. Thus, many data accesses are caused by cache miss, which aggravate the contention of global memory. In addition, for most applications, data exchanges are quite frequent among the parallel executing threads through global memory. Therefore, in our paper, we assume that there are few instructions fetched from global memory, and only focus on the contention of data access for its memory latency estimation.

Data access to global memory can be divided into two categories: explicit access and implicit access. The explicit access is launched by the write_data/read_data functions, while the implicit is caused by noncacheable memory access, cacheable line fill, and cacheable write back. For the explicit access, the read/write address and size are specified by the communication functions. A pair of profiling APIs is inserted before and after the communication functions to profile the execution time between two explicit memory accesses. Therefore, a write/read operation can be launched at the exact time by the BFM for explicit memory access.

However, for implicit memory access, it is hard to know when to launch a write/read operation based only on the analysis result from a memory analyzer. To solve this problem, we propose an EAM to imitate implicit global memory access

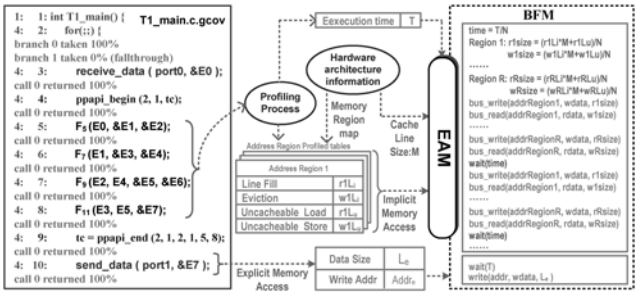


Fig. 8. EAM work mechanism.

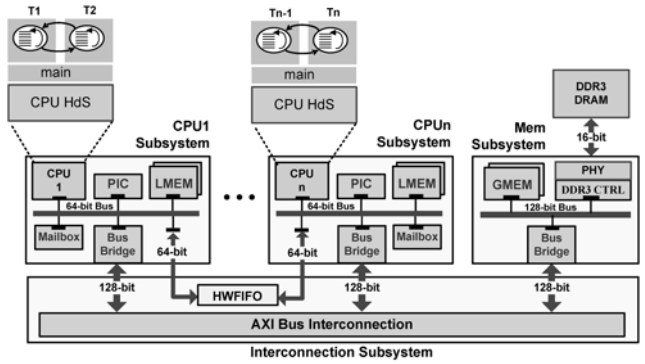


Fig. 9. Template for MPSoC software and hardware platform.

behavior. The EAM generates a uniform distribution for all the behaviors of global memory access within a given period to imitate the contention of global memory. As shown in Fig. 8, according to hardware architecture information, we can obtain the memory region map of global shared memory and cache line size of individual processors. Based on statistic results from the profiling process and the memory region map, we can generate an address region profiled table that contains total time cost of four kinds of memory access, i.e., cache line fill, cache eviction, noncacheable load, and noncacheable store, for each memory region. With the data from the address region profiled table, the EAM can distribute all implicit memory access behaviors into multiple explicit ones with uniform time interval period.

As shown in the BFM in Fig. 8, the time interval between two memory access behaviors is decided by parameter N, which depends on the features of application software. It is chosen as a smaller value for an application with less memory access frequency. Otherwise, a larger one is chosen. We define N_{max} as the maximum value of parameter N. According to (3), the value of N_{max} is calculated from four parameters: L_c (total cacheable memory access times), L_{nc} (total noncacheable memory access times), R_c ($L_c/(L_c + L_{nc})$: the percentage of cacheable access in total memory access) and M (cache line size).

$$N_{max} = R_c * M * L_c + (1 - R_c) * L_{nc} \tag{3}$$

To find an appropriate parameter N, there are four choices: N_{max}, $N_{max}/2$, $N_{max}/4$, and $N_{max}/8$ in terms of the implicit memory access probability during the total execution time between two explicit memory accesses (like send_data or receive_data) from a processor. The choices will be assigned to different EAMs according to the memory access probabilities of their processors. For the example of a six-core architecture with six independent BFMs, if the memory access probability of a given processor is the largest, we choose N_{max} as the value of the parameter for the corresponding EAM. And we choose $N_{max}/2$ for the EAM of a processor with the second largest probability. In the case that the access probability of a processor is smaller than the fourth largest, we still choose $N_{max}/8$ for its EAM. The experimental results show that such a way to decide parameter N based on the probability can reduce the error of contention estimation by 8% on an average, compared with the way using the same value. We do not choose other values for parameter N because the experience gained in the experiments tells us that the error of contention estimation is much aggravated if the value is less than $N_{max}/16$. The coarse granularity of memory access in the EAM may lead to long-time occupancy of the shared memory, which causes unexpected contention.

We also give an example of bus operations generated by the EAM for implicit memory access in the BFM in Fig. 8. By taking advantage of the EAM, a TA model can simulate the communication behavior more close to the real system and provide more accurate performance estimation.

VI. Case Study

We have implemented the proposed estimation method in a Simulink-based MPSoC design platform [5], [8]. From an application Simulink model, this platform can automatically generate hardware and software cosimulation environments at two abstraction levels: transaction accurate model for transaction accurate simulation and VP model for cycle accurate simulation. The proposed techniques are integrated into the toolsets of performance evaluation in the hardware and software cosimulation environment for the TA model. To check the effectiveness of the proposed techniques in the simulation of the TA model, we have applied them to different MPSoC architectures with an H.264 baseline video decoder.

A. Software and Hardware Platform

To efficiently evaluate the proposed techniques, we have adopted a flexible MPSoC software and hardware platform with good scalability and strong configurability. As shown in Fig. 9, the hardware platform consists of CPU subsystems, a memory subsystem and an interconnection subsystem. Each CPU subsystem uses a 64-bit local bus to connect one processor with other local components. It is also connected with an external 128-bit global AMBA AXI [26] bus interconnection through a bus bridge. The memory subsystem uses a 128-bit local bus to connect on-chip global SRAM (GMEM) and off-chip DDR3 SDRAM. This platform has good scalability that is capable of accommodating two to sixteen CPU subsystems. Moreover, the processor type in the CPU subsystem is configurable with two different choices: ARM7 (ARM, Inc.) processor and CKCore processor [27] (C-SKY, Inc.) processor. A global FIFO (GFIFO) mechanism is used for interprocessor communication and those buffers for thread communication

TABLE III
APPLYING TECHNIQUES FOR EACH MODEL

Model	Applied Techniques
TA2_a	InstructionAnalyzer
TA2_b	InstructionAnalyzer + MemoryAnalyzer
TA2_c	InstructionAnalyzer + MemoryAnalyzer + CacheAnalyzer
TA2_d	InstructionAnalyzer + MemoryAnalyzer + CacheAnalyzer + EAM

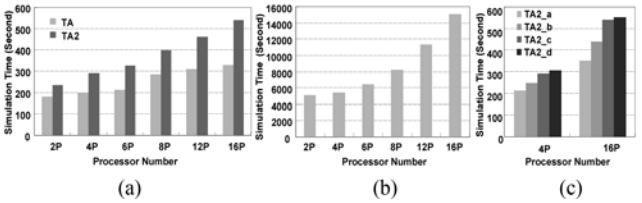


Fig. 10. Simulation time of a ten-frame QCIF Foreman H.264 decoder application. (a) Simulation Time Comparison between TA and TA2. (b) Simulation Time of VP with different processor numbers. (c) Simulation Time Comparison among TA2_a, TA2_b, TA2_c and TA2_d with 4-Processor and 16-Processor architecture.

are not cacheable. The processor in each CPU subsystem has an identical cacheable memory region that is not shareable with other CPU subsystems. The software platform consists of thread codes, CPU main codes, and an HdS library on the target CPU.

B. Simulation Time

Simulation speed is an important factor for high-efficient performance estimation. In our experiments, we partition an H.264 decoder application into 16 threads that are distributed to run in 2- to 16-CKCore MPSoC platforms. To distinguish from the traditional TA model, we name the TA model using our estimation method as TA2. Fig. 10(a) shows the simulation time consumed by different MPSoC architectures on TA and TA2 models. Compared with the traditional TA model, the simulation time of the TA2 model is increased by 20–70% in different architectures. These speeds are much faster than that of the VP model, as shown in Fig. 10(b). We can also find that the simulation time of the TA2 model is almost linear with the increasing number of processors, which indicates that it has good scalability for more complex MPSoC performance estimation.

To fully analyze the simulation time of the techniques used in the TA2 model, we run the experiments with four configurations listed in Table III: TA2_a using the instruction analyzer only, TA2_b using instruction and memory analyzers without Cache, TA2_c adding the instruction/data cache analyzer based on TA2_b, and TA2_d using all techniques including an EAM of global memory. The simulation results are shown in Fig. 10(c).

C. Estimation Accuracy

Accurate performance evaluation for a given MPSoC platform is a key to successful architecture design. However, it is still a challenge because there are many factors to affect the accuracy of the final performance estimation result. Above all, both memory confliction and cache latency are usually the most important factors for the estimation methods based on static analysis. According to the two factors, we use three kinds of MPSoC architectures to demonstrate the feasibility of the TA2 model in our experiment.

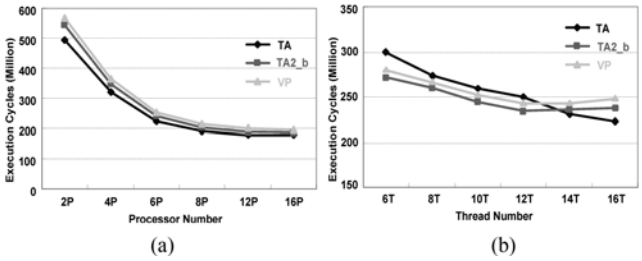


Fig. 11. Execution cycle results on the MP1 platform.(a) Execution Time Comparison for different processor number. (b) Execution Time Comparison for different thread numbers.

1) MP1 architecture platform: It is an ideal platform with less consideration on global memory confliction and cache latency. This platform assumes that local memory is large enough to contain all the instructions and variables for local processor usage. The global memory is only used for interprocessor communication and the TA model in SystemC can explicitly observe its access or contention.
2) MP2 architecture platform: Based on the MP1 platform, it is further equipped with instruction and data cache mechanism. This platform also assumes that local memory is capable of containing all necessary data for a local processor. Local memory belongs to the cacheable memory region, and global memory is noncacheable.
3) MP3 architecture platform: It is a real platform with more consideration on global memory confliction and cache latency. Compared with the MP2 platform, the local memory of the MP3 platform is limited to be small, owing to the constraint of chip area. The global memory is cacheable and used by each processor for both instruction and data storage.

We have applied different thread architectures to the above three platforms with different number of processors. The purpose of our experiments is to prove the feasibility and accuracy of the proposed method for performance estimation.

1) *Experimental Results on the MP1 Platform:* We choose a 30-frame Foreman QCIF H.264 decoder as the input in this experiment. Running 16 threads on the MP1 platform with multiple ARM7 processors is implemented in the TA, TA2, and VP models, respectively. In Fig. 11(a), the curves illustrate the trend of total execution cycles for different MP1 architectures with 2–16 processors. It shows that the curves of TA and TA2 models are almost similar to that of the VP model. But the execution cycles of the TA2 model is more close to that of the VP model, which indicates that the result of the TA2 model is more accurate than that of the TA model.

In another experiment, we have partitioned the H.264 decoder application into 6- to 16-thread architectures and run them on the MP1 platform with six ARM processors. As shown in Fig. 11(b), the curve of the TA2 model is also

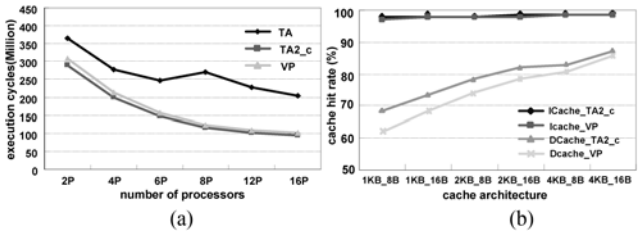


Fig. 12. Experimental results on the MP2 platform. (a) Execution Time Comparison for different processor number. (b) Instruction and data cache hit rate of CPU1 in 4-Processor MP2 Platform.

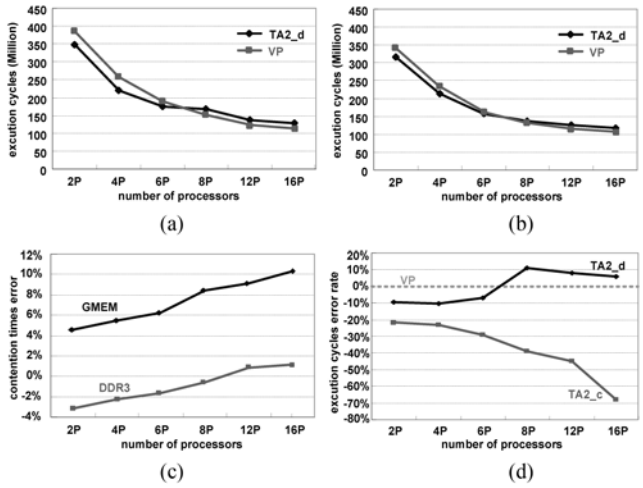


Fig. 13. Experimental results on the MP3 platform. (a) Execution Time Comparison with system clock ration 8:4:1. (b) Execution Time Comparison with system clock ratio 4:2:1. (c) GMEM and DDR3 Memory Access Contention Times Errors between TA2_d and VP. (d) Execution Time Error Rate comparison between TA2_d and TA2_c.

similar to that of the VP model, but the result of the TA model becomes a bit far different from that of the VP model. With the increasing number of threads, the result error of the TA model may be caused by less consideration on the HdS, such as the cycles consumed by its thread scheduler. Another reason leading to the result error of the TA model is that those annotated cycles for the coarse-grained threads are not accurate enough, if different conditions are taken. The curve slope of the TA model is obviously higher than that of the TA2 and VP models. We may choose 16-thread software architecture as the best one among all options after TA model performance estimation. But, in fact, the result is not better than the 12-thread architecture after VP model performance estimation. Therefore, we can find that the TA model is not accurate enough to guarantee the best choice for a given design space. Instead, the TA2 model provides more accurate estimation while keeping fast simulation speed, which greatly improves the efficiency of performance evaluation during design space exploration.

2) *Experimental Results on the MP2 Platform:* In this experiment, we still choose the 30-frame Foreman QCIF H.264 decoder as the input and run 16 threads on the MP2 platform with multiple CK510 CKCore processors for TA, TA2_c, and VP models, respectively. As shown in Fig. 12(a), the performance results are obtained respectively for TA, TA2_c, and VP models by using the MP2 platform, in which each CKCore processor is equipped with a 4 KB instruction cache and a 4 KB data cache separately. We can find that the curve of the TA2_c model is still quite similar to that of the VP model. With the cache analyzer, the average error rate between TA2_c and VP models increases from 4% to 7%, compared to the TA2_b model on the MP1 platform. Moreover, it also shows that this error rate is reducing with the increasing number of processors. The reason for this reduction comes from higher cache hit rate and less memory visit with more fine-granularity threading architecture on each processor. However, we also find that the curve of the TA model is quite different from that of the VP model. The results estimated by the TA model look much worse than that of the VP model because of the pessimistic cache hit rate annotated from the single-thread application simulation on the VP model. Therefore, the estimation result of the TA model on the MP2 platform is not accurate enough to guide us to make the best decision in architecture exploration.

To further prove the accuracy of the TA2_c model, we have also implemented two-way set-associative cache architectures for each processor in the MP2 platform with different cache sizes and different cache block sizes. The instruction and data cache hit rates of one processor (CPU1) in the four-processor MP2 platform are shown in Fig. 12(b). The instruction cache rate obtained from the TA2_c model is almost the same as that from the VP model, which means that the instruction cache analyzer provides accurate estimation on instruction cache behavior. However, compared with the VP model, the error of data cache hit rate estimated by the TA2_c model is almost 8% in the case that each processor is equipped with 1 KB cache and 8-byte block (1KB_8B). This error is reduced to less than 2% if the cache size becomes 4 KB and its line size is 16 bytes (4KB_16B). On the whole, the curve for data cache hit rate of the TA2_c model is similar to that of the VP model, which indicates that the TA2_c model is able to present the exact changing trends of performance during MP2 platform architecture exploration.

3) *Experimental Results on the MP3 Platform:* We also use the 30-frame Foreman QCIF H.264 decoder as the input in the experiment on the MP3 platform. The MPSoC hardware platform is configured as a feasible prototype with limited local memory size from 2 to 8 KB. It integrates a 512 KB global on-chip SRAM (GMEM) for shareable data among different processors. We have allocated other big buffers, such as decoded picture buffer (DPB), to external DDR3 SDRAM, which is cacheable to each local processor. The processor type is configured as CK610 CKCore processor. The clock ratio among processors, local bus subsystems, and AXI bus system in the MP3 platform is defined as 8:4:1, which indicates that the clock frequency of the processor is eight times as that of AXI bus and GMEM.

From the experimental results shown in Fig. 13(a), we can find that the curve of the TA2_d model also keeps the same changing trend as that of the VP model. The difference between TA2_d and VP models is varied from 7% to 16%, which increases a lot compared with those shown in the

MP2 platform experiment. More global memory access from local processors obviously brings more estimation error to the TA2_d model. To find the reason to the increased error, we also present the results of another experiment running on the MP3 platform with clock ratio defined as 4:2:1, which brings less memory contention on global memory access. As shown in Fig. 13(b), the difference of execution cycles between TA2_d and VP models is from 6% to 11%, which is extremely reduced for less memory contention. In order to show the effectiveness of the EAM technique, we have compared the accuracy of the TA2_d model against that of the TA2_c model in another experiment. As shown in Fig. 13(d), we use the error rate of execution time to indicate the accuracy of estimated result. If the error rate is much closer to 0, the accuracy is higher. We find that the error rate of the TA2_d model is from -10% to 11%, while the result of the TA2_c model is too optimistic with error rate from -21% to -69%. Therefore, without the EAM technique, the accuracy of performance estimation is decreased extremely and becomes worse with the increasing number of processors.

In the case that the EAM is used, the errors of contention times for accessing GMEM and DDR3 SDRAM are given with two curves, respectively, for different MPSoC architectures in Fig. 13(c). For all architectures, we can find that the total contention times of GMEM estimated by the TA2_d model is more than that obtained from the VP model. For both GMEM and DDR3 SDRAM, the error between these two models is increasing with more processors integrated into the MP3 platform. However, the contention result of DDR3 SDRAM seems more complicated and has to be analyzed with two different cases. When the number of processors is 12 or 16, the total contention times of DDR3 SDRAM estimated by the TA2_d model is more than that obtained from the VP model. In other cases, the result of contention times estimated by the TA2_d model is less than that of the VP model. This also explains why two curves intersect as shown in Fig. 13(a) and (b). The most accurate result is obtained from the six-processor TA2_d model with only 6–7% difference from the VP model and the result estimated from the TA2_d model is less than that of the VP model when the number of processors is not more than six.

VII. Conclusion

We have presented dynamic simulation and static analysis combined techniques for performance estimation of multi-thread applications running on MPSoC. To obtain the code coverage information, the multithread code of application software are complied and executed natively, concurrently with a child process named analyzing process. In the analyzing process, the instruction and memory analyzers are applied to calculate the computation cycles and memory access latency based on the results of native simulation. For the instruction analyzer, the instruction weight table is imported to look up executing cycles of each instruction. And for the memory analyzer, both instruction and data cache models are introduced to improve the accuracy of memory access latency. Furthermore, we have also presented an EAM to imitate implicit global memory access behaviors with uniform distribution to generate more accurate contention result. A series of experiments using an H.264 decoder as application input demonstrate both simulation efficiency and estimation accuracy of the proposed techniques.

In the future work, the branch prediction and pipeline stall models will be investigated and developed based on our current work, which are expected to improve the accuracy of processor performance estimation and approach to that of ISS. Furthermore, the EAM will be optimized with more advanced models to improve the imitation accuracy of global memory access. Finally, we will also integrate the cache models and the EAM to support estimating the performance of cache coherence in symmetric multi processing (SMP) system.

References

[1] A. A. Jerraya and W. Wolf, "Hardware/software interface co-design for embedded systems," *IEEE Comput.*, vol. 38, no. 2, pp. 63–69, Feb. 2005.

[2] A. A. Jerraya, H. Tenhunen, and W. Wolf, Guest editors "Introduction: Multiprocessor systems-on-chips," *IEEE Comput.*, vol. 38, no. 7, pp. 36–40, Jul. 2005.

[3] M. Grant, "Overview of the MPSoC design challenge," in *Proc. Conf. DAC*, 2006, pp. 274–279.

[4] A. A. Jerraya, A. Bouchhima, and F. Petrot, "Programming models and HW-SW interfaces abstraction for multi-processor SoC," in *Proc. Conf. DAC*, 2006, pp. 280–285.

[5] K. Huang, S. I. Han, K. Popovici, L. Brisolara, X. Guerin, L. Li, X. L. Yan, S. I. Chae, L. Carro, and A. A. Jerraya, "Simulink-based MPSoC design flow: Case study of motion-JPEG and H.264," in *Proc. Conf. DAC*, 2007, pp. 39–42.

[6] P. Gerin, M. Hamayun, and F. Petrot, "Native MPSoC co-simulation environment for software performance estimation," in *Proc. CODES + ISSS*, 2009, pp. 403–412.

[7] K. Popovici, A. A. Jerraya, F. Rousseau, and W. Wolf, "Embedded software design and programming of multiprocessor system on chip," Berlin, Germany: Springer, 2010.

[8] S. I. Han, S. I. Chae, L. Brisolara, L. Carro, K. Popovici, X. Guerin, A. A. Jerraya, K. Huang, L. Li, and X. L. Yan, "Simulink-based heterogeneous multiprocessor SoC design flow for mixed hardware/software refinement and simulation," *Integration*, vol. 42, no. 2, pp. 227–245, 2009.

[9] K. Albers, F. Bodmann, and F. Slomka, "Hierarchical event streams and event dependency graphs: A new computational model for embedded real-time systems," in *Proc. Conf Real-Time Syst.*, 2006, pp. 97–106.

[10] GNU. (2013, May). *GCOV-a Test Coverage Program* [Online]. Available: http://gcc.gnu.org/onlinedocs/gcc/Gcov.html

[11] R. Wilhelm, J. Engblom, A. Ermedahl, N. Holsti, S. Thesing, D. Whalley, G. Bernat, C. Ferdinand, R. Heckmann, T. Mitra, F. Mueller, I. Puaut, P. Puschner, J. Staschulat, and P. Stenstr, "The worst-case execution-time problem-overview of methods and survey of tools, "*ACM Trans. Embedded Comput. Syst.*, vol. 7, no. 3, pp. 36–51, 2008.

[12] E. Wandeler, L. Thiele, M. Verhoef, and P. Lieverse, "System architecture evaluation using modular performance analysis: A case study," *Int. J. Softw. Tools Technol. Transfer (STTT)*, vol. 8, no. 6, pp. 649–667, Nov. 2006.

[13] Coware, Inc. Cadence, Inc. (2004). *ConvergenSC/Incisive Design Flow* [Online]. Available: http://w2.cadence.com/whitepapers/CDNCoWare_WPfnl.pdf

[14] ARM, Inc. (2008). *RealView MaxSim* [Online]. Available: http://www.arm.com/community/partners/display_product/rw/ProductId/2250/

[15] L. Benini, D. Bertozi, A. Brogiolo, F. Menichelli, and M. Olivieri, "MPARM: Exploring the multi-processor SoC design space with systemC," *J. Very Large Scale Integr. Signal Process.*, vol. 41, pp. 169–182, Sep. 2005.

[16] D. Chiou, D. Sunwoo, J. Kim, N. A. Patil, W. Reinhart, D. E. Johnson, J. H. Keefe, and H. Angepat "FPGA-accelerated simulation technologies (fast): Fast, full-system, cycle-accurate simulators," in *Proc. Conf. Micro Architecture*, 2007, pp. 249–261.

[17] E. S. Chung, E. Nurvitadhi, J. C. Hoe, B. Falsafi, and K. Mai, "A complexity-effective architecture for accelerating full-system multiprocessor simulations using FPGAs," in *Proc. FPGA*, 2008, pp. 77–86.

[18] D. A. Penry, D. Fay, D. Hodgdon, R. Wells, G. Schelle, D. I. August, and D. Connors, "Exploiting parallelism and structure to accelerate the simulation of chip multi-processors," in *Proc. High Performance Comput. Architecture*, 2006, pp. 27–38.
[19] C. M. Kirchsteiger, H. Schweitzer, C. Trummer, C. Steger, R. Weiss, and M. Pistauer, "A software performance simulation methodology for rapid system architecture exploration," in *Proc. Conf. ICECS*, Aug. 2008, pp. 494–497.
[20] P. Gerin, M. M. Hamayun, and F. Pétrot, "Native MPSoC co-simulation environment for software performance estimation," in *Proc. Conf. Hardw./Softw. Co-Design Syst. Synth.*, Oct. 2009, pp. 403–412.
[21] E. Cheung, H. Hsieh, and F. Balarin, "Fast and accurate performance simulation of embedded software for MPSoC," in *Proc. Conf. ASP-DAC*, 2009, pp. 552–557.
[22] J. Castillo, H. Posadas, E. Villar, and M. Martinez, "Fast instruction cache modeling for approximate timed HW/SW co-simulation," in *Proc. Great Lakes Symp. Very Large Scale Integr.*, 2010, pp. 191–196.
[23] H. Posadas, L. Diaz, and E. Villar, "Fast data-cache modeling for native co-simulation," in *Proc. Conf. ASP-DAC*, 2011, pp. 425–430.
[24] J. Schnerr, O. Bringmann, A. Viehl, and W. Rosenstiel, "High-performance timing simulation of embedded software," in *Proc. Conf. DAC*, 2008, pp. 90–295.
[25] TIS committee, *Executable and Linkable Format, version 1.2* [Online]. Available: http://www.parolamia.eu/ELF.pdf
[26] ARM, Inc. (2003). AMBA AXI Protocol v1.0
[27] C-SKY, Inc. (2012). *CKcore Processor* [Online]. Available: http://www.c-sky.com/

De Ma was born in June 20, 1985. He received the B.S. degree in electrical and information engineering from Zhejiang University, Hangzhou, China, in 2004, and the Ph.D. degree from the Institute of VLSI Design, Zhejiang University.

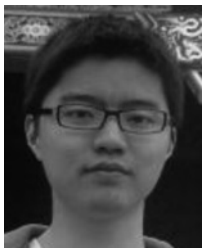
From 2010 to 2011, he was an Intern at the VERIMAG Laboratory, Grenoble, France. He is currently a Faculty Member with the Department of Electronic and Information, Hangzhou Dianzi University, Hangzhou, China. His current research interests include H.264/AVC video codec, SoC architecture exploration and implementation, and MPSoC performance estimation.

Rongjie Yan received the Ph.D. degree from the Institute of Software, Chinese Academy of Sciences, Beijing, China, in 2007.

She is an Assistant Researcher with the Institute of Software, Chinese Academy of Sciences. She has spent two years at VERIMAG, Grenoble, France, where she focused on compositional and incremental verification methodology, and correctness-by-construction of component-based systems. Her current research interests include modeling and formal verification of embedded systems. Recently, she worked on extra-function analysis of embedded systems.

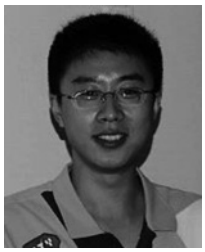
Kai Huang received the B.S.E.E. degree from Nanchang University, Nanchang, China, in 2002, and the Ph.D. degree in engineering circuit and system from Zhejiang University, Hangzhou, China, in 2008.

From 2006 to 2006, he was a short-term Visitor with the TIMA Laboratory, Grenoble, France. From 2009 to 2011, he was also a Post-Doctoral Research Assistant with the Institute of VLSI Design, Zhejiang University. In 2010, he also worked as a Collaborative Expert in VERIMAG Laboratory, Grenoble, France. Since 2012, he has been an Associate Professor with the Department of Information Science and Electronic Engineering, Zhejiang University. His current research interests include embedded processors and SoC system-level design methodology and platforms.



Min Yu received the bachelor's degree from the Department of Information Science and Electronic Enginerring, Zhejiang University, Hangzhou, China, in 2009. He is currently pursuing the Ph.D. degree at the Institute of VLSI Design, Zhejiang University.

His current research interests include performance estimation, high-performance software exploration on multiprocessors, and performance-oriented automatic code generation on MPSoC.



Siwen XIU received the bachelor's degree in electronic science and technology from Zhejiang University, Hangzhou, China, in 2009. He is currently pursuing the Ph.D. degree at the Institute of VLSI Design, Zhejiang University.

His current research interests include MPSoC performance estimation and architecture exploration, multiprocessor architecture design, and digital system design.

Haitong Ge received the bachelor's degree from Chongqing University, Chongqing, China, in 1994, and the Ph.D. degree in engineering circuit and system from Zhejiang University, Hangzhou, China, 2009.

Since 2000, he has been with Hangzhou C-SKY Microsystems, Hangzhou, China. His current research interests include embedded CPU and SoC design.

Xiaolang Yan received the B.S.E.E. and M.S.E.E. degrees from Zhejiang University, Hangzhou, China, in 1968 and 1981, respectively.

From 1993 to 1994, he was a Visiting Scholar at Stanford University, Palo Alto, CA, USA. From 1994 to 1999, he was a Professor and the Dean of the Hangzhou Institute of Electronic Engineering, Hangzhou, China. Since 1999, he has been a Professor, the Dean of the Information Science and Engineering College, and the Director of the Institute of VLSI Design, Zhejiang University. His current research interests include embedded CPU design, SoC design methodology, and design for manufacturability.

Ahmed Amine Jerraya received the Engineering degree from the University of Tunis, Tunis, Tunisia, and two doctoral degrees in computer science from the University of Grenoble, Grenoble, France.

He joined Leti, Grenoble, France, in 2007 as the Research Director and the Head of strategic design programs. Prior to joining Leti, he cofounded two startups and was the Research Director of the French National Center for Scientific Research (CNRS), Rennes, France. At CNRS, he also managed the TIMA laboratory's research dealing with multiprocessor system-on-chips. He has co-authored eight books and published more than 200 papers in international conferences and journals.