Timed Compiled-Code Functional Simulation of Embedded Software for Performance Analysis of SOC Designs

Jong-Yeol Lee

Division of EE, Department of EECS, KAIST

Abstract

In this paper, a new timing generation method is proposed for performance analysis of embedded software. The time stamp generation of I/O accesses is crucial to performance estimation and architecture exploration in the timed functional simulation, which simulates the whole design at a functional level with timing. A portable compiler is modified to generate time-deltas which are the estimated cycle counts between two adjacent I/O accesses, by counting the cycles of the intermediate representation (IR) operations and using a machine description that contains information on a target processor. Since the proposed method is based on the machine-independent IR of a compiler, the method can be applied to various processors by changing the machine description. The experimental results show that the proposed method is effective in that the average estimation error is about 2% and the maximum speed-up over the corresponding instruction-set simulators is about 300 times. The proposed method is also verified in a timed functional simulation environment.

I. Introduction

According to advances in design methodology and semiconductor technology, many of today's complex embedded systems are implemented as system-on-a-chips (SOC's). In such a SOC, processor cores and other system components available as intellectual properties (IP's) are integrated on a single chip. In addition to the design complexity, the lifetime of a design becomes shorter than ever before. It is very important, therefore, to reduce the design time as minimal as possible in order to fit the design in a narrow time-to-market window. Since software design is more flexible than hardware design and design errors can be corrected more easily even in the later design stage, an increasing amount of system functionality is being implemented in software running on the processor cores. An example of an SOC is shown in Fig. 1, where a processor, memories and functional blocks are connected with a system bus.

In SOC design, the fast and accurate performance analysis of a target system at a higher level of abstraction is greatly required to reduce the exploring time of the design space of the target system and to obtain guidelines on proceeding to the lower level design. As the processor core and other system components are connected with one or more buses in a SOC, the bus contention is one of the major factors that determine the target system performance. To identify how and when the bus contention occurs, the analysis of the timing at which the system components including the software component access the bus is required.

Various techniques have been proposed for the estimation of software timing in the mixed hardware/software systems [1][2][3][4][5][6][7]. Some works mandate long execution times since they use instruction-set simulators for cycle-accurate results. The authors of [1] addressed the problem of annotating the original C code with timing estimates guessing the compiler behavior. However, their method shows good results only on codes generated by POLIS system since the codes always have the same structure. In [2], GNU C compiler is utilized to generate "assembler-level" C code, which can be annotated with timing information and used as a simulation model. This approach needs to run the simulation model and different simulation models for other processors. In [3], an annotated C code is generated from a functional C code or the object code generated by a target compiler and the annotated C code is simulated on a separate simulator. A method supported by mathematical models of C statements in terms of elementary operations was proposed in [4]. The method in [4] is based on a hierarchical analysis of the code structure and gathers a mathematical model of the timing issues with profiling information. The deterministic contribution is combined with a statistical term accounting for all those aspects that cannot be quantified exactly. The authors of [5] proposed a method based on the analysis of the compiler-generated target binary code for each basic block in order to incorporate information

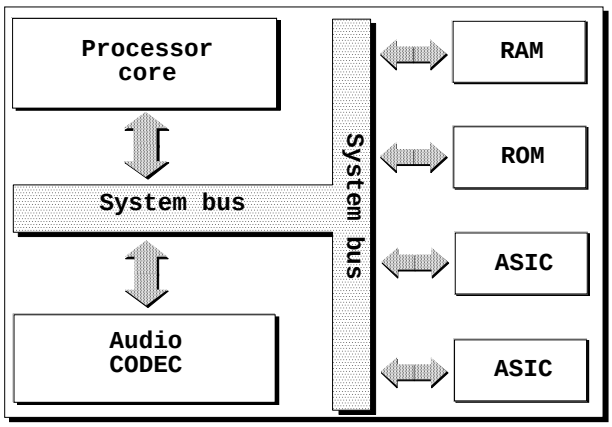

Fig. 1. Example of an SOC.

about the compiler optimization. Since the target binary code is translated into host binary code in the method proposed in [5], it needs a target-specific binary code translator for each target. In Seamless [6], a method called "Compiled Code Execution" is supported. In the method, the embedded software is compiled for a host machine instead of a target CPU. As the software executes on the host, the I/O transactions are trapped and selectively translated into bus cycles that excite the surrounding hardware. The results of the method may be inaccurate since there is no way to make up for the discrepancy between the host-machine and target CPU. Innoveda's Virtual-CPU [7] also provides compiled-code execution capability. This mechanism allows embedded system software compiled for the host platform to access the hardware without requiring source code modification. A bus functional model simulates the input/output pin behavior for the hardware portion of the simulation. However, no timing information is available from the compiled-code execution of the embedded software.

A simulation technology called timed functional simulation can make it possible to simulate the whole design at a functional level with timing by estimating the timing of the software components. In timed functional simulation, only the accesses in the software component to other hardware components (peripherals) are considered and the memory-mapped I/O method is assumed. Furthermore, since the memory location mapped to peripherals must remain unchanged throughout the program execution, they must be declared in the source code as specially qualified global variables which are called memory/IO variables, and should not participate in optimizations that would increase, decrease, or delay any reference to the variables since the variables can have their values altered in ways not under the control of user code [10].

In this paper, we introduce a new method to generate the timings for memory/IO accesses by using a compiler. A compiler has an intermediate representation (IR) that is a kind of abstract machine language that can express the target-machine operations without committing to a full of machine-specific details. The IR code is generated after the front end of a compiler analyzes the source code. The IR codes generated for various processors are virtually all the same regardless of processors for the same source code. The proposed method exploits compiled-code execution and shows decreased execution time compared to instruction-set simulators. A compiler is modified to generate executables that run on a host machine. The executable estimates and reports the times at which memory/IO accesses occur by counting the execution cycles of the IR operations between adjacent accesses to memory/IO variables. For accurate estimation, the proposed method needs the machine description of a target machine. One section of the machine description is a table called *an operation table*, which specifies the size and execution cycle of each IR operation when

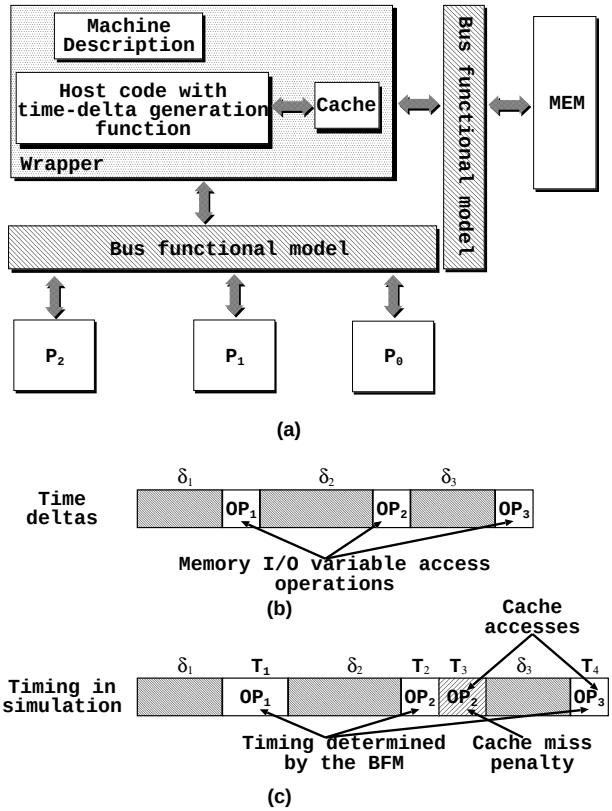

Fig. 2. (a) Conceptual block diagram of timed functional simulation, (b) Time-deltas, and (c) Timing in simulation.

the IR operation is translated into machine instructions. The machine description also contains other information such as the type of a processor, the number of registers, the number and kinds of functional units, and the latency of memory.

The rest of the paper is organized as follows. The proposed timed functional simulation is described in Section II. The proposed time-delta generation method and its two inputs are explained in Section III and Section IV, respectively. In Section V, two augmentations made to increase the accuracy the time-delta generation is described. After showing the experimental results in Section VI, conclusions are addressed in Section VII.

II. Timed Functional Simulation

Fig. 2(a) shows a conceptual block diagram of timed functional simulation. For timed functional simulation, an algorithm described in a high-level language is compiled into an executable, called a wrapper, that runs on a host machine and has a time-delta generation function. A time-delta is an estimated cycle count between two adjacent memory/IO variable access operations. An example of time-deltas, δ_1, δ_2 and δ_3, is shown in Fig. 2(b), where OP_1, OP_2, and OP_3

are memory/IO operations. Let us assume that OP_1 is a transfer to P_1, OP_2 is an access to MEM that is missed in CACHE, and OP_3 is an access to MEM that is hit in CACHE. Only the timing between memory/IO operations is estimated in the time-delta generation since it is determined in software execution. To estimate the time-delta, the wrapper counts the number of IR operations between two memory/IO operations. Since the IR code is usually generated for scalar processors, the IR operations are scheduled to increase the accuracy of time-deltas for superscalar or VLIW processors. The wrapper also contains routines necessary for interrupt processing. The bus functional model (BFM) translates the memory/IO accesses into real chip signals according to the target CPU specification. The access time of memory/IO operations is determined by cache simulation, memory latency and the BFM that models the interaction between a target CPU and peripherals. For example, the memory latency determines the timing of the operations that load or store data from/to memory. Fig. 2(c) shows the timing in a timed functional simulation in which time-deltas shown in Fig. 2(b) are simulated. The access time of OP_1, T_1 is determined in the BFM by simulating the bus signals of an embedded processor. For OP_2 and OP_3, the accesses are checked in CACHE to decide whether the data is in CACHE. These checking times are denoted as T_2 and T_4. As the OP_2 causes a cache miss, a memory access is invoked through the BFM, which is denoted as T_3. Therefore, the access time of OP_2 is the sum of T_2 and T_3.

The memory block (MEM) and peripherals (P_0, P_1, and P_2) in Fig. 2(a) can be software models in high-level languages such as C, hardware description language (HDL) models or a mixture of software models and HDL models. By changing the machine description and Verilog tasks called in the BFM, other target CPU's can be easily simulated. One of the advantages of timed functional simulation is that the behavior and timing of the memory/IO signals of a target CPU can be simulated although the simulation is performed at functional level.

III. Time-Delta Generation

To generate time-deltas, a method based on the portable compiler concept is proposed. A compiler is generally divided into two parts, the front and the back end. The front end parses a source code and generates an IR code. The back end performs various optimizations on the generated IR code. As a final step of compilation, the IR operations are translated into machine instructions. Since the front and the back end of a compiler are linked with an IR that is independent of target machines, porting a compiler to a new machine can be achieved by only changing the back end of the compiler. This is the main concept of a portable compiler [8].

Since general compilers focus on the functionality and optimization of the compiled code and

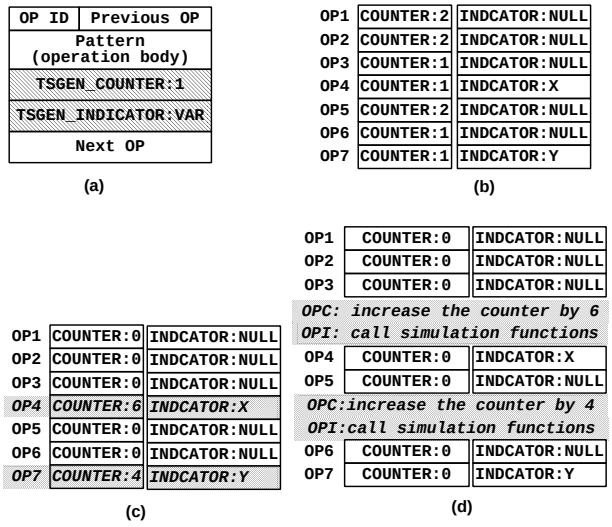

Fig. 3. (a) Data structure of IR operation with counter and indicator fields. The shaded fields represent optional fields. (b) Sequence of IR operations. The indicator fields of the non-memory/IO operations are set to null. (c) Sequence of IR operations after merging optional fields in (b). (d) Sequence of IR operations obtained by inserting IR operations, OPC and OPI, for counters and indicators.

have no functionalities related to the timing generation of the compiled code, it is necessary to modify compilers to generate the time-deltas. To generate the time-deltas, methods to indicate the memory/IO operations and to estimate the cycles elapsed between the indicated memory/IO operations must be provided. The proposed method is based on the IR of a compiler in order to exploit the portability of compilers. An optional field, called *an indicator field*, is added in the IR data structure to indicate the locations of memory/IO operations in the IR code. The indicator fields are set to the name of the variables that are accessed in the memory/IO operations identified by analyzing and searching the IR code for the memory/IO variables. The cycle counts between memory/IO operations are estimated by accumulating the cycle counts of IR operations. Since the cycle count of an IR operation is determined by a target machine, the cycle count can be obtained from the machine description. To record the obtained cycle counts in the IR operations, another newly defined optional field, called *a counter field*, is added in the IR data structure. An example of the data structure of an IR operation is shown in Fig. 3(a), where counters for IR operations and indicators for memory/IO accesses are represented as two optional fields, `TSGEN_COUNTER` and `TSGEN_INDICATOR`. A sequence of IR operations with counters and indicators is shown in Fig. 3(b), where the indicator field of a non-memory/IO operation is null.

After the optional fields are set for each IR operation, the optional fields are merged to save

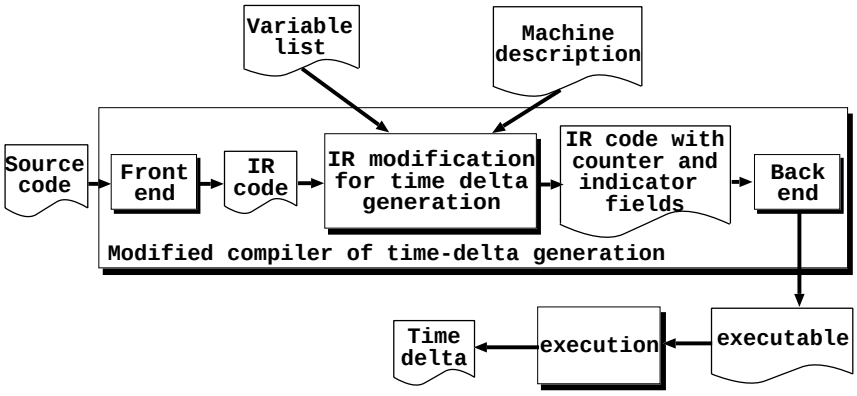

Fig. 4. Overview of time-delta generation.

execution time if there is no memory/IO access in a straight-line code such as basic blocks. A sequence of IR operations after merging optional fields is shown in Fig. 3(c), where the values of the counter fields of OP4 and OP7 are replaced by the sum of the counter fields of the preceding operations and the counter fields of the preceding operations are set to zero. The merged optional fields are translated into IR operations in the final compilation step so that the added IR operations should not affect the optimization results. The counter fields are translated into IR operations that increase the total cycle counts by the cycle count in the counter field. The total cycle counts is stored in a global variable, `tsgen_total_cycles`, defined in the library used in time-delta generation. The indicator fields are translated into IR operations that calculate the time-deltas using the value of `tsgen_total_cycles` at the previous indicator and the current value of `tsgen_total_cycles`. The IR operations translated from the indicator field also contain calls to necessary functions at each memory/IO access to simulate the interaction with the outside of the target CPU. For example, when the target CPU has a data cache, a cache simulation function is called. In Fig. 3(d), a sequence of IR operations obtained by inserting IR operations, *OPC* and *OPI*, for counters and indicators is shown. The first OPC increases the total cycle counts by six that is the value of the counter field of OP4. The operations, OPI's, are inserted for the indicator fields whose values are not null. After inserting appropriate IR operations for the counter and indicator fields, the IR code is translated into a host executable.

The proposed approach has two advantages. First, since the proposed method is based on the IR code that is independent of target machines, it can be easily applied to different processors with a small modification. By changing only the machine description, the proposed method can be retargeted to other processors. The second advantage is that the estimated cycle counts are more accurate than other methods because the counters and indicators needed for estimating

cycle counts are inserted as optional fields that have no effects on the code optimization and the modification of IR operations has the same effect on counters and indicators. For example, when an IR operation is deleted, the corresponding counter and indicator are also deleted since they are in the optional fields of the deleted IR operation.

Fig. 4 shows an overview of the proposed time-delta generation method. The memory/IO variables to be observed are registered in the variable list. The machine description contains the information on the target processor needed to generate time-deltas. Using a compiler modified for time-delta generation, a source code is compiled into a host-machine executable. The executable contains code segments and function calls that are to be inserted at the locations indicated by indicators. The executable can run in a stand-alone mode to dump time-deltas to a designated file or run in a timed functional simulation environment to invoke hardware peripherals.

IV. Variable List and Machine Description

As shown in Fig. 4, time-delta generation needs two inputs, the variable list and machine description. In this section, the inputs are described.

1. Variable List

The variable list contains the list of memory/IO variable names to be observed. The variables must be specially qualified global or static variables in functions or block boundary. The IR code is searched for the IR operations that access the variables in the variable list. A variable in the IR code is accessed either by directly referring the variable name or by loading the address of the variable into a register. Therefore, the IR accessing the designated variables can be identified by searching the IR operation related to the variable names in the variable list or by analyzing the IR code to find registers used for storing the addresses of the designated variables.

2. Machine Description

The machine description contains the information of a target processor, such as the execution cycle of each IR operation, processor architecture, memory parameter, and cache configuration. Since the IR of a compiler is independent of target machines, the IR modification routine in Fig. 4 can be used independent of target machines and only the machine description needs to be changed for other target machines.

- Operation table

The machine description contains a section called an operation table that specifies the size and execution cycle of an IR operation when the operation is translated into machine instructions.

For example, in the machine description of M-CORE, the IR operation that loads a datum from memory to a register takes two cycles since the load instruction in M-CORE is executed in two cycles. The default value is one for all the IR operations, but the value can be specified by the user. To enhance the accuracy of time-delta generation, the granularity difference between an IR operation and the corresponding machine instruction code is considered in determining the execution cycle of the IR operation. A load operation is an example of granularity difference. A single load operation in an IR code is translated into multiple instructions since an additional instruction that loads an address into a register and hence additional cycles are needed. In case of ARM7, the additional instruction that loads an address into a register takes three cycles, and thus LOAD_{ARM7} which represents the cycles consumed for a load in ARM7 is six cycles. STORE_{ARM7} which represents the cycles consumed for a store in ARM7 is five cycles.

In order to consider the effects of compiler optimization and enhance the accuracy of time-delta generation, the time-deltas generated from counters are weighted. This weighting is performed to reduce the error in total execution cycles. The weighting factor is determined by calculating a harmonic mean of the difference between the execution cycles estimated by the counters and those obtained from the ISS. In case of ARM7, the weighting factor is 1.2.

- Processor architecture

The machine description contains the type of a target processor. Since the IR code is generated for a scalar processor, for VLIW or superscalar processors such as PowerPC, additional scheduling is needed for generating more accurate time-deltas. In the proposed method, the IR code is scheduled based on the information such as the number of functional units, IR operations executed on each functional unit and the number of registers in order to estimate the real schedule on the target machine. When the average number of instructions executed in a single cycle is available, time-deltas can be generated by scaling the time-deltas obtained from a scalar machine. The number of registers in the target machine is described in the processor architecture section of the machine description. When the number of general-purpose registers of a target machine is smaller than that of a host machine, restricting the number of general-purpose registers used in simulation enhances the accuracy of time-delta generation. For example, M-CORE has only 16 general-purpose registers and restricting the number of register used in the proposed time-delta generation tool yields a good estimation result. This is because the number of general-purpose registers affects the result of compiler optimizations. Especially, the number of instructions that save and restore registers into and from memory is heavily affected by the number of registers.

A compiler must be modified to use only the registers specified in the machine description

so that the time-delta generation tool can be used for simulating various processors. This can be achieved by making the register allocation routine take parameters specifying the number of registers used for register allocation. Since the register allocation routine of a compiler generally deals with only general-purpose registers, restricting the registers is more effective for a target processor with few general-purpose registers.

- Memory parameter

The machine description also contains the latency of memory that determines that latency of load instructions. In the timed functional simulation, a load operation must wait until data is ready from the memory, whereas a store operation can go on to the next operation by writing data into a write buffer.

- Cache configuration

To generate time-deltas for a target processor with caches, the parameters for cache configuration such as the size of cache, the number of ways, associativity, replacement policy, and write policy are described in the machine description. A cache simulator [11] is provided in the time-delta generation library in order to include the effects of data and instruction caches. Data cache is simulated by calling the cache simulator at every memory access. When the configuration of cache is given in the machine description, an IR operation representing a call to a cache simulation function is inserted at the location of an indicator. In addition to the function call, IR operations that save the address and data of each memory access as arguments of the cache simulation function are inserted. Instead of assembly instructions, the IR operations are used because direct insertion of assembly instructions is possible only after the assembly code is generated and in that case, the inserted assembly instructions may cause some problems in branch operations and require register saving and restoring not to disturb the data flow. Instruction cache is simulated by assigning a virtual address to each IR operation since the real address is known after code generation and it is very hard to translate the program counter of each machine instruction into the address of IR operation. Since the size of instructions is important in instruction cache simulation, the size of an IR operation is obtained from the operation table. Each IR operation is assigned a virtual address when IR code is generated. Similarly to data cache, IR operations that call a cache simulation function with a virtual address are inserted before each IR operation. Cache miss penalty is modeled as additional clock cycles.

To speed up the simulation with a cache, the hit or miss of a cache access can be estimated by comparing a random number and the desired hit ratio, R_{hit}, of the cache. To estimate the

hit or miss of a cache access, a random number between 0.0 and 1.0 is generated and compared to R_{hit}. If the random number is larger than R_{hit}, the cache assess is estimated to cause a miss and a cache function that deals with cache misses is called. Otherwise, the access is estimated as a hit.

V. Enhanced Time-Delta Generation

The accuracy of timed functional simulation can be enhanced if a target compiler is provided and interrupt processing is considered.

1. Target Compiler

When a compiler for a target CPU is available, the time-delta can be made more accurate by using a target compiler. The number of cycles between accesses to variables can be calculated by compiling an application using a target compiler and generating the final IR code that is translated into the assembly code in the final compilation step. In the timed functional simulation, when an indicator is met during the execution of the host executable, the calculated cycles from the IR code generated by the target compiler are passed as arguments to various functions for simulation such as interrupt processing functions. Since the variables to be observed are specially qualified not to be optimized, the accesses to the variables are not deleted after optimization and an access corresponding to an access in target assembly code can be always found in the host-machine executable. To find the accesses to the designated variables, the IR code is searched in the same way as in the case of indicator insertion for the variables in the variable list.

2. Interrupts

To process interrupts in timed functional simulation, a method to process interrupts must be provided in the time- delta generation. In the proposed method, interrupt service routines (ISR's) are assumed to be provided by the user. Unlike ISS's in which interrupt requests are checked after each instruction, interrupt requests cannot be checked in the execution of compiled-code, and thus they are processed using traps on a host machine to service the interrupts as soon as possible after interrupts are requested.

When an interrupt is requested from a peripheral hardware, the wrapper causes a software trap on the host with the trap number indicating the requested interrupt. In the host-machine ISR, a target ISR corresponding to the trap number is identified and the target ISR provided by the user is called.

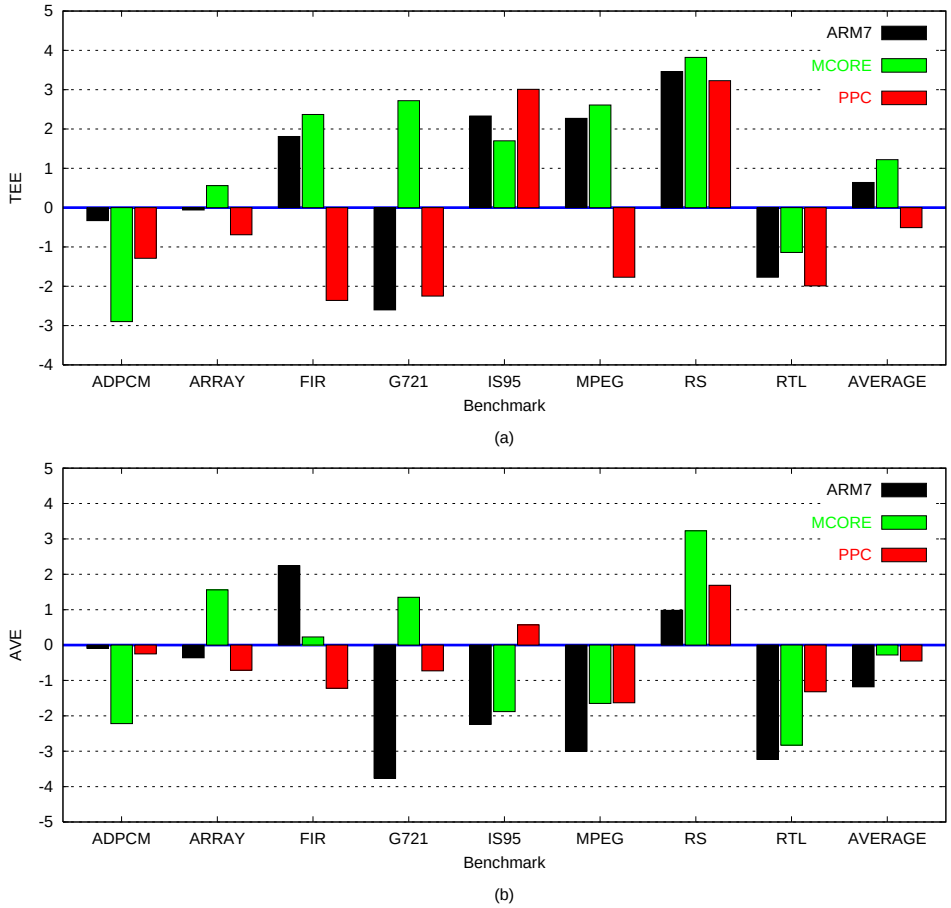

Fig. 5. Estimation errors using unoptimized host executables. (a) TEE. The standard deviation of each processor is $\mathrm{SD}_{ARM7} = 0.0216$, $\mathrm{SD}_{MCORE} = 0.0225$, and $\mathrm{SD}_{PPC} = 0.0216$. (b) AVE. The standard deviation of each processor is $\mathrm{SD}_{ARM7} = 0.0219$, $\mathrm{SD}_{MCORE} = 0.0218$, and $\mathrm{SD}_{PPC} = 0.0103$.

VI. Experimental Results

We developed a time-delta generation tool by modifying GNU C compiler (GCC) [9]. The code quality of GCC is reported to be better than commercial compilers for some machines. The IR of GCC, called RTL, has a Lisp-like form. In the modified GCC, two newly defined optional fields, the counter and indicator fields, are inserted in the RTL code that is generated from the front end. The RTL code with the counter and indicator fields is optimized in the back end. After all optimizations are performed, the counter and indicator fields are translated into appropriate assembly codes. The library needed for time-delta generation, such as cache simulators and host-machine ISRs for interrupt processing, was also developed.

To show the accuracy of the time-delta generation tool (*tsgcc*), we compared the results of tsgcc to those of instruction set simulators (ISS's) of target processors for a set of test programs. The target processors used in our experiments are ARM7TDMI [12], M-CORE [13] and PowerPC

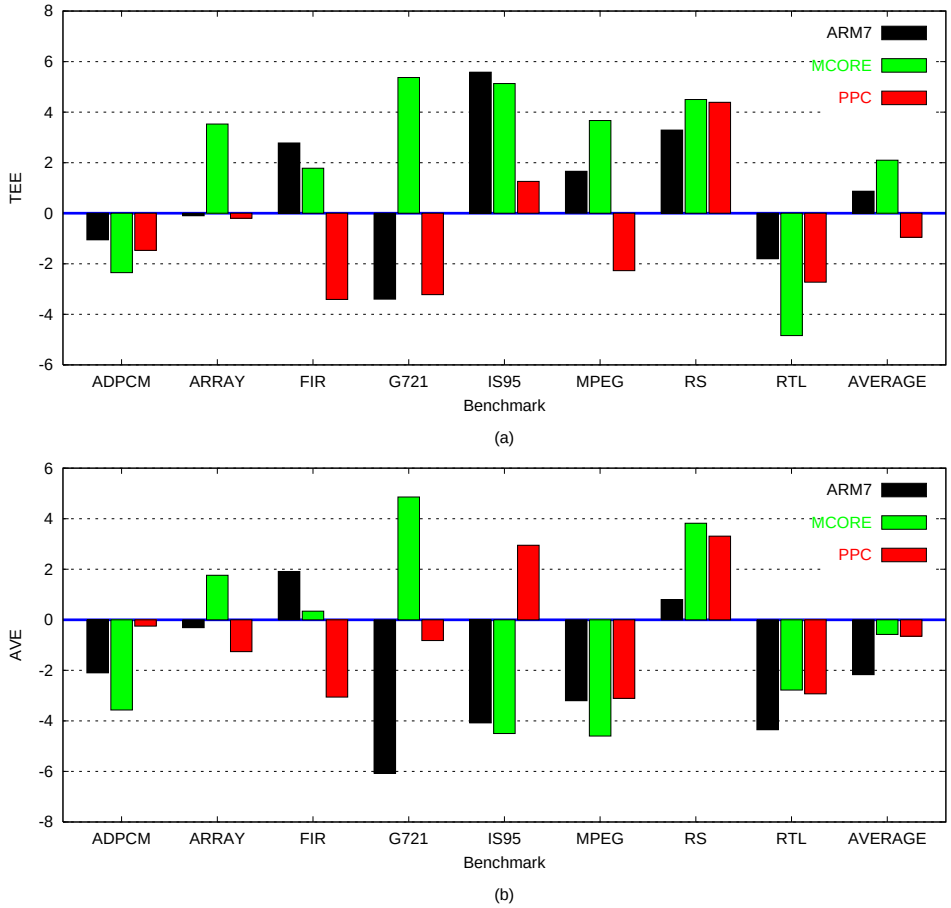

Fig. 6. Estimation errors using optimized host executables. (a) TEE. The standard deviation of each processor is $\mathrm{SD}_{ARM7} = 0.0298$, $\mathrm{SD}_{MCORE} = 0.0375$, and $\mathrm{SD}_{PPC} = 0.0251$. (b) AVE. The standard deviation of each processor is $\mathrm{SD}_{ARM7} = 0.0277$, $\mathrm{SD}_{MCORE} = 0.0379$, and $\mathrm{SD}_{PPC} = 0.0257$.

601 [14], which are most frequently used in embedded systems. As PowerPC 601 is a superscalar processor capable of issuing and retiring three instructions per clock, a list scheduling is performed in the time-delta generation tool. For cache simulation, it is assumed that PowerPC 601 contains a 32-Kbyte, eight-way set associative, unified cache and the cache line size is 64 bytes. The cache uses write-back write policy and a least-recently-used replacement policy.

Fig. 5 shows the total estimation error (TEE) and average error (AVE) of time-deltas when the host executables are generated without machine-dependent optimizations. TEE and AVE are calculated as follows.

$$TEE = \frac{C_{target} - C_{tsgcc}}{C_{target}} \text{ and}$$

$$AVE = \frac{1}{N} \sum_{i=1}^{N} \frac{Target\delta_i - Tsgcc\delta_i}{Target\delta_i}$$

where N is the number of time-deltas, C_{target} and C_{tsgcc} are the total number of execution cycles measured on the ISS and estimated by tsgcc, respectively, and $Target\delta_i$ and $Tsgcc\delta_i$

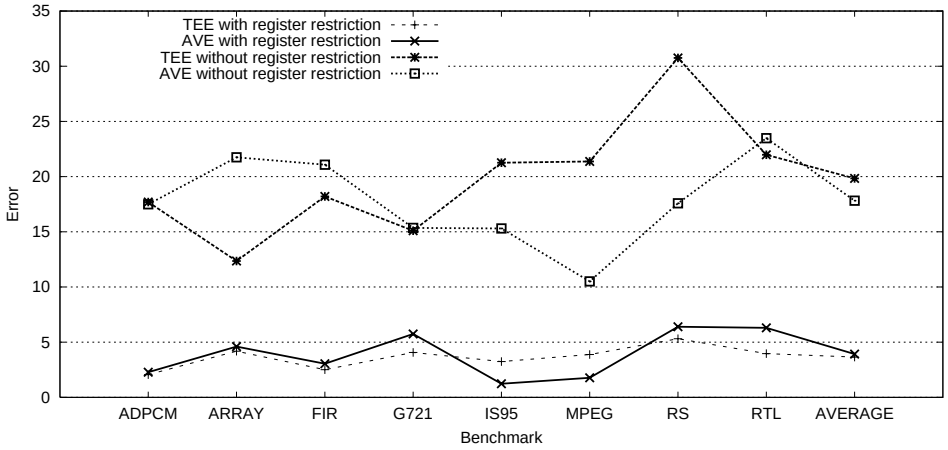

Fig. 7. Estimation errors with and without register restriction in M-CORE. The standard deviations of TEE and AVE with register restriction are 0.0143 and 0.0195 and those without register restriction are 0.0318 and 0.0298.

are time-deltas measured on the ISS and estimated by tsgcc, respectively. The AVE represents the average errors between two time-deltas generated from the ISS and tsgcc. The average TEE over benchmark programs is about 0.64% for ARM7, 1.22% for M-CORE and -0.51% for PowerPC. The TEE of M-CORE is larger than those of ARM7 and PowerPC, because of the large granularity difference between M-CORE instruction and RTL operations, whereas the difference is small in cases of ARM7 and PowerPC.

The errors for optimized host executables are shown in Fig. 6. The average TEE increases to 0.87% in ARM7, 2.10% in M-CORE, and -0.95% in PowerPC. The average AVE also increases to -2.17% in ARM7, -0.58% in M-CORE and -0.65% in PowerPC.

As shown in Fig. 5 and Fig. 6, the proposed method shows good accuracy in both unoptimized and optimized cases because it is based on the IR of a compiler. When an IR operation is deleted, inserted or moved in the back-end optimization, the counter of the deleted IR operation is also deleted, inserted or moved since the counter is associated with the optional field of the IR data structure. This means that the effect of optimization is directly reflected in the proposed time-delta generation.

Fig. 7 shows the estimation errors when the number of registers used in tsgcc is not restricted to the number of registers in M-CORE. The TEE and AVE without register restriction are experimented to show the effect of the number of registers. In this case, the TEE and AVE are increased to 19.84% and 17.82%, respectively, while the TEE and AVE with register restriction are 3.65% and 3.92%, respectively. Comparing the errors in Fig. 7, we can know that tsgcc can underestimate the execution time if more registers are provided than the real CPU has.

TABLE I

DISTRIBUTION OF DIFFERENCE IN TIME-DELTAS ($|Target\delta_i - Tsgcc\delta_i|$).

ARM7			MCORE			PowerPC		
Diff.	Count	%	Diff.	Count	%	Diff.	Count	%
2	104571	79%	3	105685	79 %	1	108932	82%
21	2729	2%	53	3255	2%	4	4471	3%
22	2684	2%	9	2790	2%	10	2551	2%
25	2513	2%	8	2699	2%	12	1921	1%
78	2479	2%	48	1909	1%	2	1885	1%
86	1721	1%	2	1746	1%	6	1479	1%

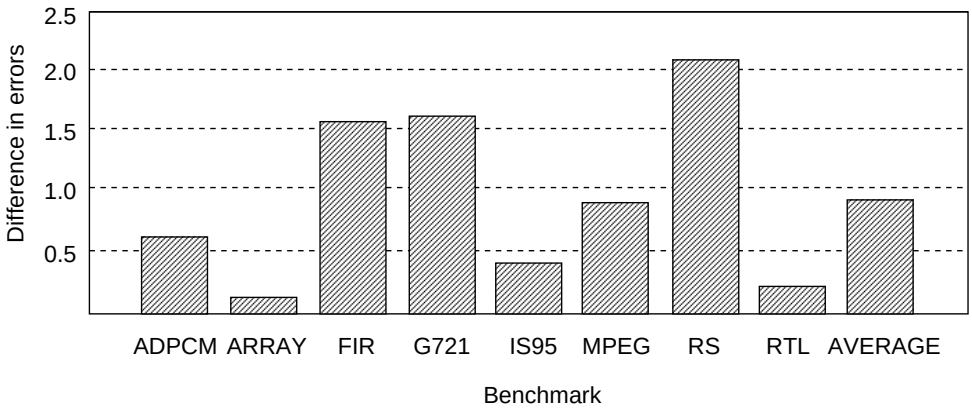

Fig. 8. Differences in TEE's obtained in the cache simulation and the estimation using random numbers and $R_{hit} = 0.95$.

However, with restricting the number of registers to that of M-CORE, the error is reduced since the smaller number of registers leads to more load and store operations. This result shows that restricting the number of registers is very effective in enhancing the accuracy of the time-delta generation.

The distribution of $|Target\delta_i - Tsgcc\delta_i|$ in the MPEG2 encoder is shown in Table I, where we can see that only one difference value is dominant in all processors. This is true for all other test programs. This characteristics can be used to enhance the accuracy of time-delta generation by treating the cause of the dominant difference.

The differences in TEE's obtained in the cache simulation and the estimation using random numbers are shown in Fig. 8, where the average difference in TEE is 0.93%. The simulation is speeded up by six times when the estimation is used instead of the cache simulation. From the results in Fig. 8, we can know that the estimation of cache misses using random numbers is effective in reducing simulation times.

TABLE II

SPEED-UP OF HOST-CODE EXECUTION OVER ISS.

program	ARM7	MCORE	PPC
ADPCM	4.02	3.86	4.04
G721	18.91	26.44	25.85
MPEG2	342.05	298.25	280.75

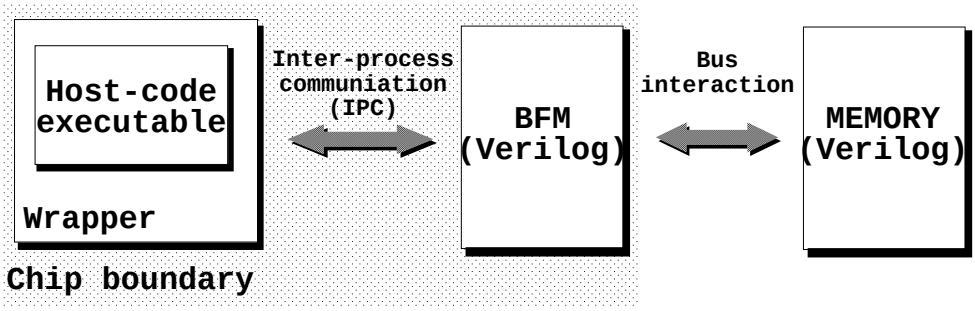

Fig. 9. Simulation with a BFM and memory model in HDL.

The speed-up of compiled-code execution over ISS is shown in Table II. The maximum speed-up occurs with the MPEG2 encoder program. The time-delta generation tool runs about 300 times faster than ISS. The speed-up becomes larger in programs with longer execution time.

Fig. 9 shows the configuration of a simulation, where a host-machine executable, a wrapper, a BFM, and a memory model are used. In the simulation, the wrapper contains codes that compensate the time-deltas generated from the host-code executable and communicates with the BFM using inter-process communication. The wrapper also contains the ISR of the host machine. The BFM and memory is described in Verilog HDL. The BFM generates bus signals to drive the memory model. The shaded area in Fig. 9 represents the boundary of a target CPU, which is ARM7TDMI. The simulation is performed with the selected global variables stored in the memory model. The waveform of signals driving the memory model in the simulation is shown in Fig. 10. The simulation was successful even for random interrupt requests and we could observe the interaction between the host executable and memory model through the BFM.

VII. CONCLUSIONS

In this paper, we have presented a new time-delta generation method. The generation of accurate time-delta is crucial to performance estimation and architecture exploration. To generate accurate time-deltas for various machines, we proposed a method based on the portable compiler concept. In the proposed method, a compiler is modified to generate time-deltas by counting the

Fig. 10. Waveform of simulation in Fig. 9.

cycles of the IR operations and using the machine description that contains information on a target processor such as the number of the cycles of each IR operation and the number of registers. Since the proposed method is based on the IR of a compiler, the method can be easily retargeted by changing the machine description. The experimental results show that the proposed method is effective in that the average error is about 2% and the maximum speed-up over ISS is about 300 times. The proposed method is also verified in a timed functional simulation environment.

Acknowledgments

The author would like to thank Prof. In-Cheol Park for valuable technical insight and advice. This work was supported in part by the Korea Science and Engineering Foundation through the MICROS center, and the Ministry of Science and Technology and the Ministry of Commerce, Industry and Energy through the project System IC 2010.

References

[1] K. Suzuki and A. Sangiovanni-Vincentelli, "Efficient Software Performance Estimation Methods for Hardware/Software Codesign," In Proc. 33rd Design Automation Conf., 1996, pp. 605-610.

[2] M. Lajolo, M. Lazarescu, and A. Sangiovanni-Vincentelli, "A Compilation-based Software Estimation Scheme for Hardware/Software Codesign," In Proc. Int. Workshop on Hardware/Software Codesign, 1999.

[3] J. R. Bammi, W. Kruijzer, L. Lavagno, and M. Lazarescu, "Software Performance Estimation Strategies in a System-Level Design Tool," In Proc. Int. Workshop on Hardware/Software Codesign, 2000.

[4] C. Brandolese, W. Fornaciari, F. Salice, and D. Scuito, "Source-level Execution Time Estimation of C Programs," In Proc. Int. Workshop on Hardware/Software Codesign, 2001.

[5] V. Zivojnovic and H. Meyr, "Compiled HW/SW Co-Simulation," In Proc. 33rd Design Automation Conf., 1996.

[6] B. Bailey, R. Klein and S. Leef. Hardware/Software Co-Simulation. [Online]. Available:http://www.mentor.org/seamless/papers.

[7] B. V. Bank. Innoveda's Virtucal-CPUTM Co-Verification Environment.

[Online]. Available:http://www.innoveda.com/products/datasheets_HTML/vcpu.asp.

[8] A. Aho, R. Sethi, and J. Ullman, Compilers - Principles, Techniques, and Tools. Reading, MA: Addison-Wesley, 1986.

[9] R. M. Stallman. Using and Porting GNU CC.
[Online]. Available: http://gcc.gnu.org/onlinedocs/gcc-2.95.3/gcc.html.

[10] S. P. Harbison and G. L. Steele Jr., C A REFERNECE MANUAL. Englewood Cliffs, New Jersey:Prentice Hall, 1995.

[11] Mark D. Hill and Alan Jay Smith, "Evaluating Associativity in CPU Caches," IEEE Trans. on Computers, C-38, 12, December 1989, p.1612-1630.

[12] ARM Ltd. ARM7TDMI Data Sheet, August 1995.

[13] Motorola. M-CORE-MM2001 Reference Manual, 1998.

[14] IBM. PowerPC 601 RISC Microprocessor User's Manual, 1993.