An Approach to Improve Accuracy of Source-Level TLMs of Embedded Software

Zhonglei Wang*[†], Kun Lu* and Andreas Herkersdorf*
*Technische Universität München, Arcisstraße 21, 80290 München, Germany
{*Zhonglei.Wang, Kun.Lu, Herkersdorf*}*@tum.de*
[†]Karlsruhe Institute of Technology, Chair for Embedded Sytems, Karlsruhe, Germany

Abstract—**Virtual Prototypes (VPs) based on Transaction Level Models (TLMs) have become a de-facto standard for design space exploration and validation of complex software-centric multicore or multiprocessor systems. The most popular method to get timed software TLMs is to annotate timing information at the basic-block level granularity back into application source code, called source code instrumentation (SCI). The existing SCI approaches realize the back-annotation of timing information based on mapping between source code and binary code. However, optimizing compilation has a large impact on the code mapping and will lower the accuracy of the generated source-level TLMs. In this paper, we present an efficient approach to tackle this problem. We propose to use mapping between source-level and binary-level control flows as the basis for timing annotation instead of code mapping. Software TLMs generated by our approach allow for accurate evaluation of multiprocessor systems at a very high speed. This has been proven by our experiments with a set of benchmark programs and a case study.**

I. Introduction

Today, the complexity of embedded systems is ever increasing. Many embedded systems consist of multiple processing elements and complex communication architecture. To support efficient design space exploration of such systems, TLM-based virtual prototypes (VPs) are widely used. TLM (Transaction Level Modeling) is a standard modeling style for system-level design and is often associated with SystemC, a standard System-Level Design Language (SLDL). In this modeling style, communication and computation can be modeled separately, with different communication architectures supporting a common set of abstract interfaces to functional units. In a TLM-based VP of a software-centric system, software TLMs (Transaction Level Models) play a very important role. They represent the functional behavior, capture computation delays and generate workload on the communication architecture. It will reduce the design effort significantly, if we can get an automation tool to generate fast and accurate software TLMs from application code.

A TLM for a software program can be generated from different representation levels of the program, including source level, IR level and binary level [1]. Source-level software TLMs are most widely used, because of their high simulation speed and low complexity. A source-level TLM uses source code as the functional representation. Timing information that represents execution delays of the software on the target processor is inserted into the source code. This process is often called Source Code Instrumentation (SCI). Software simulation using source-level TLMs is called Source Level Simulation (SLS). To the best of our knowledge, in all the existing SCI approaches back-annotation of timing information is based on mapping between source code and binary code. These approaches cannot generate accurate software TLMs in the case of optimizing compilation, because after optimizing compilation it is hard to find an accurate mapping between source code and binary code. Even when a code mapping can be found, due to the difference between source-level and binary-level control flows the back-annotated timing information cannot be aggregated correctly during the simulation. This problem will be presented with an example in Section III.

In this paper, we present a solution to the mapping problem raised by compiler optimizations. In our approach, we annotate the timing information estimated from binary code back to source code according to mapping between binary-level and source-level control flows instead of code mapping. The source-level TLMs generated by our approach allow for much more accurate simulation than those generated by previous approaches.

The rest of this paper is organized as follows: related work is discussed in Section II. Following this, Section III gives an overview of source code instrumentation and states the mapping problem. Then, the proposed approach is presented in detail in Section IV. Some experimental results and a case study are shown in Section V. Finally, the paper is concluded in Section VI.

II. Related Work

Software is usually simulated using instruction set simulators (ISSs). To simulate a multiprocessor system, a popular method is to integrate multiple ISSs into a SystemC based simulation backbone. Many simulators are built following this solution, e.g., the commercial simulators from Synopsys [2] and academic simulation platforms like MPARM [3]. Such simulators are able to execute target binary, boot real RTOSs, and provide cycle-accurate (or cycle-count-accurate) performance data. However, they have the major disadvantage of extremely low simulation speed and very high complexity.

Recently, most research activities focused on modeling processors at a higher level to achieve a much higher simulation speed but without compromising accuracy. Trace-based simulation methods proposed in [4], [5] are in this category. However, trace-driven simulations have a common drawback that traces are not able to capture a system's functionality. Furthermore, the execution of most tasks is data-dependent, but traces can represent only the workload of one execution path of a program.

To get high-level simulation models that capture both the function and data-dependent workload of software tasks, Source-level TLMs are being increasingly used. In recent years many papers on source level simulation are published, such as [6], [7], [8]. All the proposed approaches generate software simulation models by annotating application source code with timing information at the basic-block level granularity. The timing information of each basic block can be obtained by means of static analysis [7], [8] or measurement using cycle-accurate simulators [6]. However, in the scope of a basic block, it lacks the execution context of global timing effects, such as the cache effect and the branch prediction effect, so these timing effects cannot be resolved before the simulation run-time. A solution proposed in [7], [8] is to annotate some code to trigger dynamic simulation of the global timing effects at simulation run-time. Nevertheless, all these approaches use code-mapping as the basis for source code instrumentation and have the mapping problem raised by optimizing compilation, as discussed before.

An efficient solution to the mapping problem is to transform source programs to IR-level code which comprises machine-independent optimizations and has a structure close to that of binary code, so that there is accurate mapping between this IR-level code and binary code and software TLMs can be generated by annotating timing information into the IR-level code. This idea is proposed in [9], [10]. This method has been proven to be efficient but it will increase the complexity of TLM generation. Furthermore, the obtained IR-level TLMs lose readability. It is hard to debug a system's functionality during the simulation.

III. Overview of Source Code Instrumentation and Problem Statements

In Fig. 1 we use an example to give an overview of code-mapping-based source code instrumentation (SCI). The inputs are the source code and binary code of a program and the output is the same source code annotated with extra timing information at the basic-block level granularity. The timing information is obtained by timing analysis on the binary code. Delay values are expressed by *wait()* statements to advance SystemC simulation time.

The back-annotation of the timing information relies on mapping between source code and binary code. However, in the case of optimizing compilation, binary-level control flows are quite different from source-level control flows. The back-annotated timing information according to the code mapping cannot be aggregated correctly along the source-level control flows during the simulation. This is especially true for control-dominated programs, which contain complex structures.

Fig. 2 shows an example to depict this mapping problem. The code is part of the program *fibcall*. It is found that the instructions at 0x1800074, 0x180007c–0x1800080, and

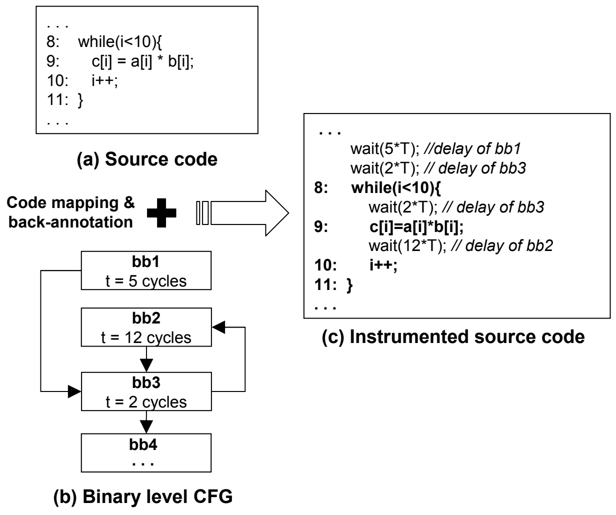


Fig. 1. An Example of Source Code Instrumentation

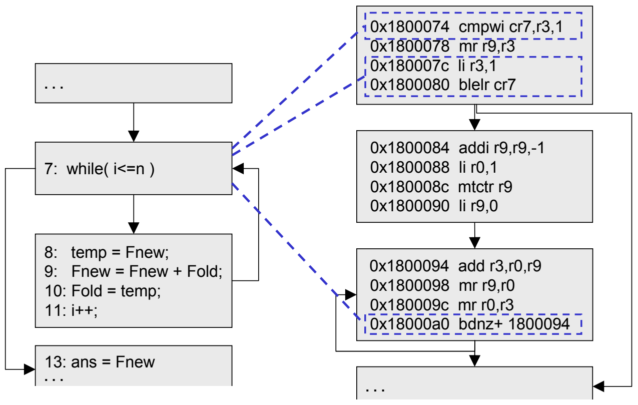
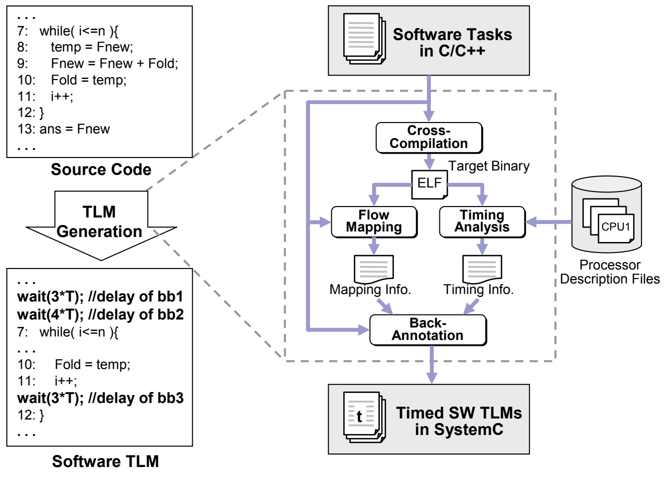


Fig. 2. Code Mapping

0x18000a0 are generated from source line 7. This code mapping is illustrated in the figure with dashed lines. According to the code mapping, the timing information of these instructions will be back-annotated into the *while* loop. All the timing information will be aggregated the same number of times as the loop iterations, but, along the binary-level control flows, the instructions at 0x1800074 and 0x180007c–0x1800080 are executed only once. Therefore, the software TLM generated by the code-mapping-based SCI has a large error. Another serious problem is that, in order to get a code mapping, basic blocks have to be broken into smaller sequences of instructions, most consisting of only 2-3 instructions. Timing information obtained from timing analysis on such small sequences of instructions is hard to take some important timing effects into account.

IV. The Proposed Approach

The proposed TLM generation approach is illustrated in Fig. 3 along with an example. The source code is the same as the one in Fig. 2, but, in the generated software TLM, the timing information has been correctly annotated. As shown at the right hand side of the figure, the whole approach consists of three working steps: (1) mapping information generation, (2) timing information generation and (3) back-annotation of



```
. . .
7:   while( i<=n ){
8:       temp = Fnew;
9:       Fnew = Fnew + Fold;
10:     Fold = temp;
11:     i++;
12: }
13: ans = Fnew
. . .
```

```
. . .
wait(3*T); //delay of bb1
wait(4*T); //delay of bb2
7:   while( i<=n ){
. . .
10:     Fold = temp;
11:     i++;
wait(3*T); //delay of bb3
12: }
. . .
```

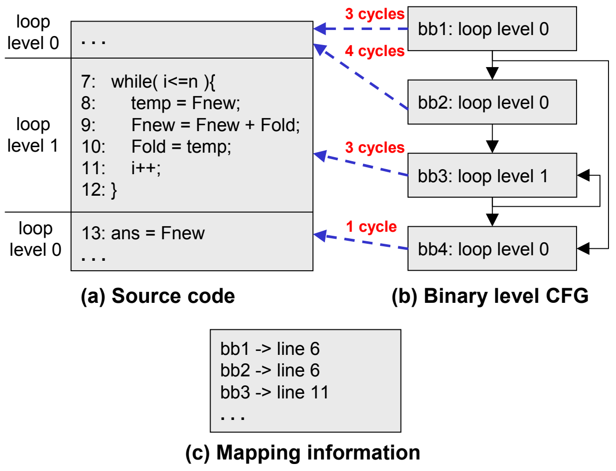
Fig. 3. The Proposed Software TLM Generation Approach

the timing information according to the mapping information. We use a hybrid timing analysis approach, similar to the one introduced in [8], to get timing information at the basic block level granularity. Different from the previous SCI approaches, we use a more efficient way to get mapping information for accurate back-annotation of timing information. This is the main contribution of this paper. Hence, the step of mapping information generation is first and foremost introduced in detail. The other two steps are described briefly. After that, the whole approach is demonstrated using an example *insertsort*.

A. Mapping Information Generation

During optimizing compilation, a compiler will often move code from within a loop to outside the loop as much as possible, to achieve a better execution time. However, using a code-mapping-based SCI approach, the binary code that has been moved out of the binary-level loop is still mapped to source code in the source-level loop and its timing information will be annotated into the source-level loop. This is the main error in software TLMs generated by code-mapping-based approaches. This problem has been illustrated in Fig. 2 in the last section. Therefore, a correct instrumentation should put the timing information of the binary code outside a binary-level loop out of the corresponding source-level loop and put the timing information of the binary code within the binary-level loop into the source-level loop, without caring from which source code the binary code is generated. To establish such a mapping between source-level loops and binary-level loops, we introduce the notion of *loop level*. A loop level indicates the depth of a loop. All the code that does not belong to any loop is assigned loop level 0. Then, the most outer loops are assigned loop level 1. If a loop with loop level n contains another loop, the loop level of the inner loop is $n+1$. In this way, all the code blocks of a program are assigned a loop level. This applies to both source code and binary code.

After identifying the loop levels, the timing information of binary code is then back-annotated into source code at the same loop level. The exact mapping information is generated according to the mapping between source-level and binary-level control flows at the same loop level. In other words, loops set up scopes for flow mapping. Therefore, in our approach, mapping information generation is achieved by two steps: (1) identifying loop levels and (2) mapping control flows at each loop level. The whole approach is flow-mapping-based, different from the code-mapping-based approaches.



```
bb1 -> line 6
bb2 -> line 6
bb3 -> line 11
. . .
```

Fig. 4. An Example of Mapping Information Generation

1) Identifying Loop Levels: It is relatively easy to identify loop levels for source code. We can simply parse the source code to find the key words of loop structures such as *while* and *for* and then find the scope of the loops. Fig. 4 shows the same example as in Fig. 2. In the source code, there is only one *while* loop. The code in the loop is assigned loop level 1, while the other code belongs to loop level 0.

It is a little more complicated to identify loops at the binary level than at the source level. We first need to create a control flow graph (CFG) for each function, which is essential to control flow analysis. In a CFG, a back edge is a special edge that points to an ancestor in a depth-first traversal of the CFG. Each loop in a binary-level CFG contains at least one back edge. We define this kind of back edges as *loop-forming back edges*. Finding a back edge is considered as a necessary condition but not the sufficient condition of identifying a loop. To make sure that a back edge is really a loop-forming back edge, it must be checked whether there is a control flow leads the destination block of the back edge to its source block, i.e., to find a closed graph. In the binary-level CFG of the running example, there is only one back edge, which jumps from the last instruction of bb3 to the first instruction of the same block. Therefore, bb3 alone forms a loop and is assigned loop level 1. The other basic blocks are at loop level 0.

2) Mapping Control Flows: After identifying loop levels, mapping information can be generated based on a mapping between source-level and binary-level control flows at respective loop levels. Besides loops, the control flows at each loop level are constructed by simple branches. Normally, each path of a binary-level branch can be simply mapped to a source-level structure like "*if ... else ...*" or "*switch ... case ...*". An entry of mapping information is written in form of "*basic block* *number* ⟶ *source line number*”, indicating that the timing information of the specified basic block should be annotated after the specified source line.

In the running example, the timing information of bb3 at loop level 1 should be back-annotated into the *while* loop, the code of which is also at loop level 1. The binary-level loop contains only one basic block and the loop body of the source-level loop has also only one basic block. Therefore, the mapping is quite straightforward. On principle, the timing information of bb3 can be put anywhere in the loop body. We insert it after the last line of the source-level basic block, so we get the mapping entry “bb3 ⟶ line 11”. In the binary code at loop level 0, we find a branch that leads control flows from bb0 to bb2 and bb4. However, we cannot find a source-level branch that corresponds to this branch. There is only one basic block of source code before the *while* loop. Therefore, it fails to get an exact mapping between the control flows in the scope of loop level 0. In this case, we simply annotate the timing information of both bb1 and bb2 into the source code before the *while* loop. This does not cause any error in most cases, since most input data does not lead the control flow from bb1 to bb4 at all.

During the flow mapping basic blocks are the smallest units. We do not need to break them into smaller sub-blocks as in code-mapping-based approaches. Hence, the source-level TLMs generated by our approach are annotated with more accurate timing information.

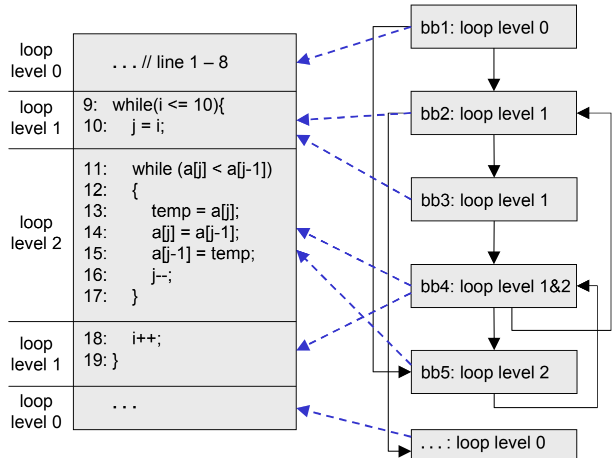
B. Timing Information Generation and Back-annotation

We use a hybrid timing analysis method to take into account the important timing effects of processor microarchitecture in source-level TLMs. Hybrid timing analysis means that timing analysis is performed both statically and dynamically. Some timing effects like pipeline effects are analyzed at compile-time using an offline performance model and are represented as delay values. Other timing effects like the branch prediction effect and the cache effect that cannot be resolved statically are analyzed at simulation run-time using an online performance model. Therefore, there are two categories of timing information: (1) delay values obtained by static timing analysis, and (2) some code for triggering dynamic analysis. In all the illustrated examples in this paper, only delay values are shown for clarity. For more information about dynamic simulation of global timing effects, please refer to [8]. Given the mapping information as shown in Fig. 4(c), the basic-block level timing information can be simply back-annotated.

The SystemC *wait()* statements will cause a large overhead on simulation performance. To improve the simulation performance, we can reduce the number of *wait()* statements by aggregating timing information using variables and calling a *wait()* only before an access to an external device.

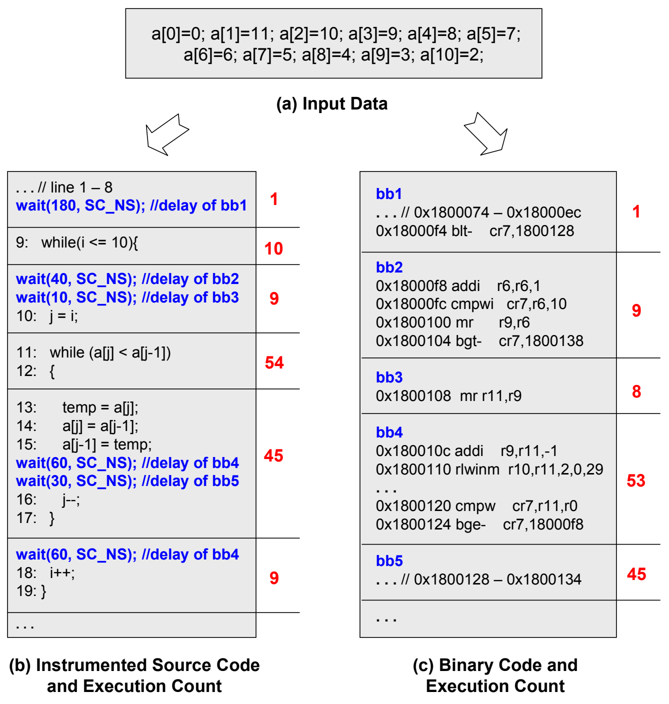
C. Another Example: insertsort

Now, we use a more complicated example *insertsort*, which contains two nested loops, to demonstrate the whole approach. With this example, we will also discuss about the possible



```
. . . // line 1 – 8
9:   while(i <= 10){
10:     j = i;
11:     while (a[j] < a[j-1])
12:     {
13:         temp = a[j];
14:         a[j] = a[j-1];
15:         a[j-1] = temp;
16:         j--;
17:     }
18:     i++;
19: }
. . .
```

Fig. 5. Flow Mapping of insertsort


(a) Input Data
(b) Instrumented Source Code and Execution Count
(c) Binary Code and Execution Count

```
a[0]=0; a[1]=11; a[2]=10; a[3]=9; a[4]=8; a[5]=7;
a[6]=6; a[7]=5; a[8]=4; a[9]=3; a[10]=2;
```

```
. . . // line 1 – 8
wait(180, SC_NS); //delay of bb1                1
9:  while(i <= 10){                             10
wait(40, SC_NS); //delay of bb2
wait(10, SC_NS); //delay of bb3                 9
10:  j = i;
11:  while (a[j] < a[j-1])                      54
12:  {
13:     temp = a[j];
14:     a[j] = a[j-1];
15:     a[j-1] = temp;
wait(60, SC_NS); //delay of bb4                 45
wait(30, SC_NS); //delay of bb5
16:     j--;
17:  }
wait(60, SC_NS); //delay of bb4
18:  i++;                                       9
19: }
. . .
```

```
bb1
. . . // 0x1800074 – 0x18000ec                  1
0x18000f4 blt-    cr7,1800128
bb2
0x18000f8 addi   r6,r6,1
0x18000fc cmpwi   cr7,r6,10                     9
0x1800100 mr      r9,r6
0x1800104 bgt-    cr7,1800138
bb3
0x1800108  mr r11,r9                            8
bb4
0x180010c addi    r9,r11,-1
0x1800110 rlwinm  r10,r11,2,0,29                53
. . .
0x1800120 cmpw    cr7,r11,r0
0x1800124 bge-    cr7,18000f8
bb5
. . . // 0x1800128 – 0x1800134                  45
. . .
```

Fig. 6. Instrumented Source Code and Binary Code of insertsort

sources of errors. First, we identify the loop levels for both source code and binary code, as shown in Fig. 5. The outer loop has loop level 1, while the inner loop has loop level 2. In the binary code, bb4 is a common part of the inner loop and the outer loop, so it is assigned both loop level 1 and 2. After identifying loop levels, the binary-level basic blocks can be mapped to the source code. The mapping is shown in the figure. With this mapping, the timing information of each basic block is back-annotated and we get instrumented source code as shown in Fig. 6(b). As bb4 belongs to both loop level 1 and 2, its timing information is back-annotated into both the inner loop and outer loop. In Fig. 6, we also show the execution count of each block of source code and binary code at its right hand side, given a set of input data shown in Fig. 6(a). As shown, the delay values of most basic

TABLE I
SIMULATION ACCURACY

	Estimated Execution Time in Cycles				Estimation Error of SLS+		
	ISS	BLS	SLS	SLS+	Timing Error	Mapping Error	Overall Error
fibcall	3307	3281	12996	3281	-0.79%	0.00%	-0.79%
insertsort	516	518	672	525	0.39%	1.35%	1.74%
bsearch	46057	47008	51008	47008	2.06%	0.00%	2.06%
crc	10288	10284	15430	10540	-0.04%	2.49%	2.45%
blowfish	178610	175919	187184	175919	-1.51%	0.00%	-1.51%
AES	2292275341	2296272902	2336484895	2296272902	0.17%	0.00%	0.17%

blocks are aggregated the same number of times as these basic blocks are executed, except bb3 and bb4. The delay values of bb3 and bb4 are aggregated 9 and 54 times, respectively, along the source-level control flows, but bb3 and bb4 are actually executed 8 and 53 times in the real execution. This results in an error of 7 cycles. This error is very small, only 1.35% in percentage, and is fully acceptable in system level design space exploration. This example shows that in the case of some complicated control flows our approach may fail to find an exact mapping between binary-level and source-level control flows, resulting in a small margin of error.

D. Limitations

The method is based on the assumption that the loop structures are not heavily changed by optimizations. Some loop transformations such as complete loop unrolling or loop fission can still be tackled. However, if a loop is partially unrolled or is broken into multiple loops which iterate over different contiguous portions of the index range, then it is hard to perform back-annotation of timing information.

V. EXPERIMENTAL RESULTS

The experiments have been carried out mainly to show the accuracy and performance of software TLMs generated by the proposed approach, compared with representative approaches of ISS, binary level simulation (BLS) and source level simulation (SLS). We chose the approach introduced in [8] to represent SLS. To be differentiated from the previous SLS approaches, the proposed one is referred to as SLS+ in the following discussion. We selected 6 benchmark programs and chose a PowerPC processor as the target processor. All the programs were compiled using a cross-compiler ported from a GCC compiler, with the optimization level -O2. After showing the efficiency of our approach, we used a case study to demonstrate how the approach facilitates design space exploration of MPSoCs.

A. Accuracy and Performance of Software Simulation

To evaluate software simulation alone, the selected programs can execute to the end without interactions to other devices. The estimation error of SLS+ can be analyzed by comparing estimates from ISS, BLS and SLS+.

BLS generates a software simulation model of a program by first translating the binary code to functionally equivalent C code, with one basic block corresponding to a C function, and then annotating the timing information of each basic block into the corresponding C function. The timing information at basic-block level granularity used in BLS has a slight error compared with the cycle level timing information of the ISS. We can measure this error by comparing the estimates from BLS and the ISS. Our SLS+ uses the same basic-block-level timing information as BLS but has an extra error caused by back-annotation of the timing information from the binary level to the source level. As BLS gets both functional representation and timing information from the binary level, it has no such a mapping error. By comparing the estimates from BLS and SLS+, we can measure the mapping error of SLS+. And by comparing the estimates from the ISS and SLS+, we can get the overall estimation error.

The estimated execution times of all the programs from the ISS, BLS, SLS and SLS+ as well as the estimation errors of SLS+ are shown in Table I. We can see that for four programs SLS+ has no mapping error and allows for software simulation as accurate as BLS. For the other two programs the mapping error is also within 2.5%. The reason for the mapping error in the simulation of *insertsort* has been discussed in detail in Section IV-C. The error of the basic-block level timing information used in BLS is also very small, only 0.83% on average for the selected programs. SLS+ uses the same timing information as BLS and therefore has the same timing error. Taking both mapping error and timing error into account, SLS+ has an overall error of 1.45% on average, much better than SLS, which applies code-mapping-based source code instrumentation. As shown in Table I, SLS allows for accurate simulation of computation-intensive programs like AES and blowfish. In the simulation of the other programs, large errors are seen. The average error is 65.1%

We measured the simulation performance on a PC equipped with a 3 GHz Intel CPU and 4 GB memory. As all the programs are simulated without interactions to other devices, the timing information is aggregated simply using a variable to get the execution time of a program and thus the slowdown due to SystemC *wait()* statements and events has not been taken into account. These effects are taken into account in the following case study, where a whole system is simulated. SLS+ had an average simulation speed of 4675.6 MIPS, close to native execution (5198.2 MIPS on average) and much faster than the ISS (11.9 MIPS on average) and BLS (349.4 MIPS on average).

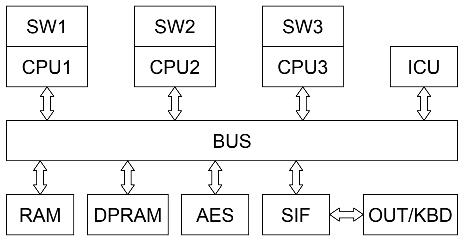


Fig. 7. System Configuration

HW/SW timing comparison			
Case	1	2	3
ISS[ms]	9.74	80.94	174.8
SW TLM[ms]	10.05	81.14	181.78
Error[%]	3.23	0.24	4.02

Runtime comparison			
Case	1	2	3
ISS [s]	0.82	4.77	14.15
SW TLM[s]	4.36e-3	0.011	0.027
Gain	187	428	523

TABLE II
SIMULATION RESULTS FOR BOTH ISS-BASED AND TLM-BASED VPS

B. Case Study of MPSoC Design

The proposed software TLM generation method is applied in a hypothetic MPSoC system design to justify its efficiency. Besides the software TLMs, a virtual prototype (VP) of the whole system also includes the hardware TLMs.

The configuration of the system is shown in Fig. 7. The system is similar to the one shown in [11]. Three CPUs with the MIPS instruction set are instantiated. Each CPU runs an algorithm of prime number calculation. It fetches instructions and data from the RAM and DPRAM respectively, stores the calculated numbers in a buffer, and sends them to the SIF (serial interface) for display. In addition, each CPU encrypts/decrypts the data using an AES accelerator module.

We built two VPs for the same system. One uses instruction set simulators (ISSs) for software simulation, while the other uses the software TLMs generated by the proposed approach. In each ISS, a simplified OS is also modeled to provide library or device driver functions to the application software. The simulation models of the other system components are the same in the both VPs. The ISS-based VP is taken as reference to evaluate the other one.

The experimental results are shown in Table II. Three cases of software computation effort are inspected. Case 1 represents low computation effort, where each CPU calculates the prime number up to 1000. Case 2 considers computation effort, where each CPU computes up to 100, 1000, and 5000, respectively. Case 3 shows the highest computation effort with each CPU running the algorithm up to 5000. In all cases, the HW/SW co-simulated timing is compared between the two VPs. Using software TLMs, the estimation error of the overall HW/SW timing remains below 4% in all the examined cases. And this error does not scale with the computation effort. Thus, it justifies the accuracy of the software TLMs generated by the proposed approach. For the simulation performance, the gain of simulation performance using the source-level TLMs is related tightly to the computation effort. It is more than 500 in case 3, where the computation time dominates the whole simulation.

VI. CONCLUSIONS

This paper presented an efficient approach for automatic generation of source-level software TLMs. Especially, we attacked the mapping problem that exists in previous source code instrumentation approaches and limits their usability. The software TLMs generated by our approach are annotated with accurate timing information at the basic-block level granularity and take all the compiler optimizations and the necessary timing effects of the processor microarchitecture into considerations. Combined with simulation models of other system components, the software TLMs allow for fast and accurate software simulation in MPSoC design. The efficiency of the source-level TLMs was proven by our experiments with a set of benchmark programs and a case study of MPSoC design. The experimental results show that the source-level TLMs allow for 393 times faster simulation than an ISS with the simulation accuracy larger than 97.5%.

ACKNOWLEDGMENT

We would like to thank Michael Velten from Infineon Technologies AG for providing the virtual platforms used in the MPSoC case study.

REFERENCES

[1] Z. Wang and A. Herkersdorf, "Software performance simulation strategies for high-level embedded system design," *Performance Evaluation*, vol. 67, no. 8, pp. 717–739, 2010.
[2] Synopsys. [Online]. Available: http://www.synopsys.com
[3] L. Benini, D. Bertozzi, A. Bogliolo, F. Menichelli, and M. Olivieri, "MPARM: Exploring the Multi-Processor SoC Design Space with SystemC," *Springer Journal of VLSI Signal Processing*, vol. 41, no. 2, pp. 169–182, 2005.
[4] A. D. Pimentel, M. Thompson, S. Polstra, and C. Erbas, "Calibration of abstract performance models for system-level design space exploration," *Journal of Signal Processing Systems*, vol. 50, no. 2, pp. 99–114, 2008.
[5] T. Wild, A. Herkersdorf, and G.-Y. Lee, "TAPES–Trace-based architecture performance evaluation with SystemC," *Journal of Design Automation for Embedded Systems*, vol. 10, no. 2-3, pp. 157–179, 2005.
[6] T. Meyerowitz, M. Sauermann, D. Langen, and A. Sangiovanni-Vincentelli, "Source-level timing annotation and simulation for a heterogeneous multiprocessor," in *Proceedings of the conference on Design, automation and test in Europe (DATE'08)*, 2008.
[7] J. Schnerr, O. Bringmann, A. Viehl, and W. Rosenstiel, "High-performance timing simulation of embedded software," in *Proceedings of the Design Automation Conference*, Anaheim, USA, June 2008.
[8] Z. Wang, A. Sanchez, and A. Herkersdorf, "SciSim: A Software Performance Estimation Framework using Source Code Instrumentation," in *Proceedings of the 7th International Workshop on Software and Performance (WOSP'08)*, Princeton, NJ, USA, Jun 2008, pp. 33–42.
[9] A. Bouchhima, P. Gerin, and F. Pétrot, "Automatic instrumentation of embedded software for high level hardware/software co-simulation," in *Proceedings of the Asia and South Pacific Design Automation Conference*, 2009, pp. 546–551.
[10] Z. Wang and A. Herkersdorf, "An Efficient Approach for System-Level Timing Simulation of Compiler-Optimized Embedded Software," in *Proceedings of the 46th Annual Design Automation Conference (DAC'09)*, San Francisco, California, July 2009, pp. 220–225.
[11] W. Ecker, V. Esen, and M. Velten, "TLM+ Modeling of Embedded HW/SW Systems," in *Proceedings of the conference on Design, automation and test in Europe (DATE'10)*, 2010.