Accurate Source-Level Simulation of Embedded Software with Respect to Compiler Optimizations

Zhonglei Wang, Jörg Henkel
Karlsruhe Institute of Technology, Chair for Embedded Sytems, Karlsruhe, Germany
{*zhonglei.wang, henkel*}*@kit.edu*

***Abstract*—Source code instrumentation is a widely used method to generate fast software simulation models by annotating timing information into application source code. Source-level simulation models can be easily integrated into SystemC based simulation environment for fast simulation of complex multiprocessor systems. The accurate back-annotation of the timing information relies on the mapping between source code and binary code. The compiler optimizations might make it hard to get accurate mapping information. This paper addresses the mapping problems caused by complex compiler optimizations, which are the main source of simulation errors. To obtain accurate mapping information, we propose a method called *fine-grained flow mapping* that establishes a mapping between sequences of control flow of source code and binary code. In case that the code structure of a program is heavily altered by compiler optimizations, we propose to replace the altered part of the source code with functionally-equivalent IR-level code which has an optimized structure, leading to *Partly Optimized Source Code* (POSC). Then the flow mapping can be established between the POSC and the binary code and the timing information is back-annotated to the POSC. Our experiments demonstrate the accuracy and speed of simulation models generated by our approach.**

I. Introduction

Simulators based on Transaction Level Models (TLM)[1] are in widespread use for design space exploration of complex multiprocessor embedded systems at system level. To date, most embedded systems are software-centric, and therefore, predicting software performance is a key issue to make performance-oriented design decisions.

In recent years, Source Code Instrumentation (SCI) has become a de-facto standard for automatic generation of fast and accurate source-level software TLMs from application source code. A source-level TLM is actually the source code annotated with low-level timing information. The timing information is usually at the basic-block level granularity, obtained by timing analysis using a performance model taking important microarchitectural effects into account, such as pipeline effects, cache effects and branch prediction effects. The back-annotation of the timing information is based on a mapping between source code and binary code. However, the optimizing compilation might heavily alter the code structure of a program and makes it hard to obtain an accurate mapping between source code and binary code, leading to inaccurate simulation models. Currently, there are few previous approaches addressing such mapping problems [1], [2], [3], [4], which will be discussed in more detail in Section II.

[1]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).

This paper is aimed to solve the mapping problems and to achieve a more general SCI approach. We use a control flow analysis method that we call *fine-grained flow mapping* to obtain accurate mapping information for source code instrumentation. During control flow analysis if it fails to establish a flow mapping due to complex alteration of the code structure made by the compiler optimizations, **our novel idea** is to identify this altered part of source code and to transform it to optimized IR-level code, which is functionally equivalent to the original source code but has an optimized structure that is close to the binary code structure. The resulting code is a mixture of the original source code and the optimized IR-level code, and therefore, is called *Partly Optimized Source Code* (POSC). Then, we can obtain an accurate flow mapping between the POSC and the binary code. Given the accurate mapping information, the low-level timing information obtained from the binary code can be back-annotated into the POSC to generate an accurate source-level TLM.

The rest of this paper is organized as follows: Section II gives an overview of related work. Then, Section III briefly introduces SCI and summarizes the mapping problems. After that, the proposed approach is described in detail in Section IV. Some experimental results are shown in Section V. Finally, the paper is concluded in Section VI.

II. Related Work

The conventional way of software simulation is using Instruction Set Simulators (ISS). The common problem of ISSs is their low simulation speed and high complexity. Trace-driven simulation [5], [6] is another choice. However, it rules out functional simulation and a set of traces can simulate the workload of only one execution path of a program.

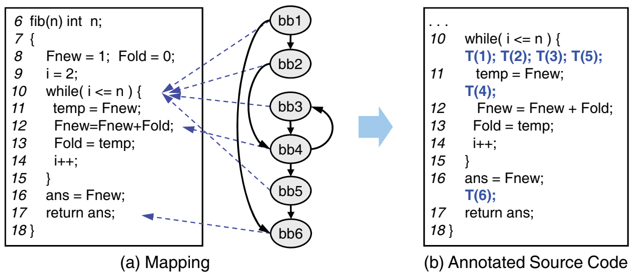
Source-Level Simulation (SLS) achieves a significant speedup without compromising much accuracy compared to cycle-accurate simulations. It is more suitable for system-level design. Early work on SLS, such as [7], [8], [9], [10], assume that there is a unique mapping between source code and binary code and does not consider compiler optimizations.

To address the mapping problems, in [1], [11], [12] the compiler is modified to add timing information into Intermediate Representation (IR). However, modifying the compiler takes much effort. Furthermore, not all compilers can be modified. Kempf et al. [13] and Wang et al. [2], [14] propose a more efficient approach. They let the compiler generate IR and change it back to C code, so no compiler modification is needed. Nevertheless, the IR-level C code might have different timing behavior from the original source code. Furthermore, IR-level code is hardly read-

978-3-9810801-8-6/DATE12/©2012 EDAA

able, and therefore rules out manual optimizations and source-level debugging.

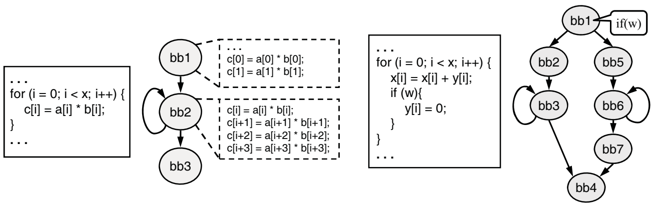
Two of the most recent works, [3] and [4], try to overcome the mapping problems while still using the original source code as the functional model. The work in [3] divides both source code and binary code of a program into segments in terms of so-called *loop levels* and annotates the timing information of binary code in a loop to the source position at the same loop level. However, this method does not work for the case that the loop structure is heavily altered by the optimizer. The work in [4] mainly addresses function inlining and loop unrolling. So-called *path simulation code* is used to simulate binary-level control flows. It works based on accurate annotations but in this paper the authors make no statements of how accurate annotations are achieved. They use what they call *loop simulation code* to map source-level loop iterations to binary-level loop iterations by a pattern with manually obtained parameters like the unroll factor and the loop bound.



```
fib(n) int  n;
{
   Fnew = 1;  Fold = 0;
   i = 2;
   while( i <= n ) {
     temp = Fnew;
     Fnew=Fnew+Fold;
     Fold = temp;
     i++;
   }
   ans = Fnew;
   return ans;
}
```

```
. . .
   while( i <= n ) {
   T(1); T(2); T(3); T(5);
     temp = Fnew;
   T(4);
     Fnew = Fnew + Fold;
     Fold = temp;
     i++;
   }
   ans = Fnew;
   T(6);
   return ans;
}
```

Fig. 1. Source Code Instrumentation (SCI)



```
. . .
for (i = 0; i < x; i++) {
   c[i] = a[i] * b[i];
}
. . .
```

```
. . .
c[0] = a[0] * b[0];
c[1] = a[1] * b[1];
```

```
c[i] = a[i] * b[i];
c[i+1] = a[i+1] * b[i+1];
c[i+2] = a[i+2] * b[i+2];
c[i+3] = a[i+3] * b[i+3];
```

```
. . .
for (i = 0; i < x; i++) {
   x[i] = x[i] + y[i];
   if (w){
      y[i] = 0;
   }
}
. . .
```

Fig. 2. Examples of Advanced Loop Optimizations

III. Problem Statement

Fig. 1 presents SCI using the *fibcall* program. In the annotated source code, each *T(block_id)* is used to simulate the timing behavior of a basic block. In SystemC, *T(block_id)* may contain a *wait()* to advance the simulation time or other code for dynamic simulation of some timing effects like cache effects and branch prediction effects. An automated SCI tool relies on computer-manipulatable information for instrumentation. To date, all existing tools use debugging information to obtain a mapping between source code and binary code. However, for optimized code, the compiler may generate wrong debugging information. Another cause of wrong mapping is due to code motion. The compiler often moves some code from inside to outside of a loop to improve performance. However, according to code mapping, their timing information is annotated back into the source-level loop, where the code was generated. The dashed lines in Fig. 1(a) depict the basic-block level mapping established through debugging information. According to this mapping, *bb1*, *bb2*, *bb3*, *bb4* and *bb5* are all simulated in the loop, but actually only *bb3* and *bb4* are part of the loop. Therefore, sophisticated analysis is needed to handle wrong debugging information and detect code motion effects.

A more serious problem is that the code structure of a program is heavily altered during optimizing compilation. Fig. 2 shows two examples of advanced loop optimizations. In the case of the partial loop unrolling shown in Fig. 2(a), although the loop structure is retained, the loop body and the number of iterations are altered. If the execution time of *bb2* is annotated into the source-level loop, the simulation result will be wrong. In the example of loop unswitching (Fig. 2(b)), the condition inside the loop is moved to outside by duplicating the loop body. In the binary code, there is a condition that selects a path to execute while in the source code there is no condition outside the loop.

Hence, the mapping problems can be categorized into two types:

- **Problem 1:** The optimizing compiler does not alter the code structure but performs some code motions and generates wrong debugging information, which makes it hard to obtain accurate mapping information for source code instrumentation.
- **Problem 2:** The code structure is heavily altered and there is no exact mapping between the source code structure and the binary code structure.

IV. The Proposed Instrumentation Approach

Our work is aimed to achieve a source code instrumentation approach which copes with both mapping problems summarized in the last section. Our solution to the mapping problems is as follows: For programs with only Problem 1 we use *fine-grained flow mapping* to obtain accurate mapping information, while for programs with Problem 2 we transform the program structure to the one that is close to the binary code structure, in order to retrieve the mapping. We therefore propose to replace part of code that causes the problem with IR-level code that accounts for all machine-independent optimizations, leading to *Partly Optimized Source Code* (POSC). Hence, our SCI approach can be divided into three steps: 1) partly optimized source code generation, 2) mapping information generation and 3) timing information back-annotation. More details are presented in the following subsections.

A. Summary of Advantages

First, we give a summary about how our approach is **superior to** state-of-the-art with respect to the two mapping problems. The previous work in [3] addresses Problem 1 by means of control flow mapping. This flow mapping just sets scopes for code mapping with respect to the loop levels. This means that the timing information of binary code in a loop should be annotated in the source code at the same loop level. However, the exact information about where in the loop the timing information should be annotated is still obtained by means of coding mapping established through debugging information. This flow mapping can well handle the code motions between the inside and outside of a loop but cannot cope with code motions within the loop, for example, between complex conditional structures. Our approach achieves accurate mapping information by establishing a mapping for each fine-grained sequence of control flow between source code and binary code. That's the reason why our approach is called *fine-grained flow mapping*.

The state-of-the-art solution to Problem 2 is to annotate timing information into the IR code instead of the source code [2], [13]. The structure of the IR code has been optimized, so there is no problem to obtain a mapping between the IR code structure and the binary code structure. One problem of the IR code is that it is hardly readable. If we want to carry out some manual adjustments or analysis on the simulation model, it is difficult to do on the IR code. Furthermore, IR-level code might have slightly different timing behavior from the original source code. Therefore, it's better to use the original source code for the generation of simulation models. Our approach is based on the fact that many programs do not encounter Problem 2 and in a program with Problem 2 usually only fractions of code are heavily optimized leading to the problem. Our novel idea is to identify this part of code by means of control flow analysis and only transform this part of code to the IR level. We also transform some unstructured IR-level code back to high-level structures to increase the readability. Thus, the negative effects of IR code can be minimized.

B. Generation of Partly Optimized Source Code

The key issue of POSC generation is to identify code that causes Problem 2. We mainly check whether loop structures are heavily altered. There are two reasons for this. First of all, the code without loop statements does not cause serious mapping problems. Second, accurate annotation for code within loops is more important, because errors due to wrong annotation within a loop will be aggregated with loop iterations, leading to a large error of the overall simulation. We define mainly three loop optimization effects that lead to Problem 2:

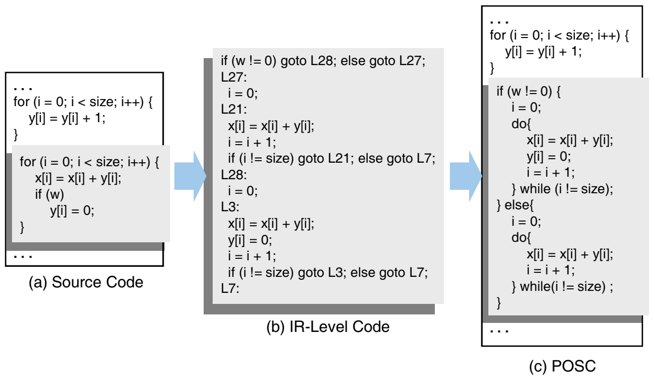
- **Changing number of loops:** some loop optimizations (e.g. loop fusion) merge loops while some other optimizations (e.g. loop fission, splitting and unswitching) duplicate or split loops.
- **Alteration of loop structure:** a loop is optimized along with complex conditional statements leading to a different loop structure. Or, two nested loops are optimized and intertwined so that it is hard to determine which one is the outer loop and which one is the inner loop.
- **Alteration of loop iterations:** after some optimizations like partial loop unrolling and loop interchange, although the loop structure is retained, the number of iterations is changed.

Algorithm 1 shows the pseudo code of the loop structure analysis for identifying the effects resulting from above mentioned loop optimizations carried out by the compiler. It first identifies loops in both source code and binary code. In the source code it needs to search for keywords like "while", "for" and "do". The loop information is stored in a list. Each node of the list contains the id, the loop level and the scope of a loop. *Loop level* is a value that represents the relation of nested loops. In the binary code a loop is identified by searching for a closed graph in the Control Flow Graph (CFG). A closed graph is identified first by finding a back edge in the CFG and then checking whether there is a control flow leading the destination block of the back edge to its source block. A loop may consist of multiple closed graphs sharing some common basic blocks. That's why a closed graph is called a *potential loop* (*pLoop*). The information of all *pLoop* is stored in a list. It then finds the corresponding source-level loop for each *pLoop* and assigns the *pLoop* to the id of the source-level loop (*sLoop_id*) (line 14).

Algorithm 1 Loop Structure Analysis

```
 1: (a) Creating a source-level loop list
 2: sLoop_list: a list to store information of source-level loops
 3: while ParseLine(source program) do
 4:     if a loop is identified then
 5:         add a node with sLoop_id, loop_level, scope into sLoop_list
 6:     end if
 7: end while
 8:
 9: (b) Creating a binary-level loop list
10: bLoop_list: a list to store information of binary-level loops
11: CFG ← CreateCFG(binary file)
12: IdentifyPotentialLoops(CFG)
13: for each potential loop pLoop do
14:     sLoop_id ← SearchSourceLoop(pLoop, sLoop_list, debugInfo)
15:     add a node with sLoop_id and information of pLoop into bLoop_list
16: end for
17:
18: (c) Identifying loops to be transformed
19: SortBLoopList(bLoop_list) //in the ascending order of sLoop_id
20: while SearchInBLoopList(bLoop_list) do
21:     if multiple pLoop have the same sLoop_id AND do not have common
        basic blocks then
22:         TransformSLoop(sLoop_id) //loop splitting or duplication
23:     else if one pLoop has more than one sLoop_id then
24:         for each sLoop_id do
25:             TransformSLoop(sLoop_id) //loop merging
26:         end for
27:     else if multiple pLoop have the same sLoop_id AND have common
        basic blocks then
28:         mapping_found ← FineGrainedFlowMap(pLoop, sLoop_id)
29:         if !mapping_found then
30:             TransformSLoop(sLoop_id) //alteration of loop structure
31:         end if
32:     else if multiple pLoop have sLoop_id of nested loops then
33:         mapping_found ← FineGrainedFlowMap(pLoop, sLoop_id)
34:         if !mapping_found then
35:             for each sLoop_id do
36:                 TransformSLoop(sLoop_id) //alteration of loop structure
37:             end for
38:         end if
39:     else
40:         p_loop_unrolling ← FineGrainedFlowMap(pLoop, sLoop_id)
41:         if p_loop_unrolling then
42:             TransformSLoop(sLoop_id) //partial loop unrolling
43:         end if
44:     end if
45: end while
```

After collecting all required information in the two lists, it starts to identify the loops to be transformed to the IR level. The *changing number of loops* effect is identified by checking whether there is more than one *pLoop* with the same *sLoop_id* (line 21) or there is one *pLoop* assigned to multiple *sLoop_id* (line 23). If multiple *pLoop* with common basic blocks have the same *sLoop_id* (line 27), they actually belong to the same loop with a complex control flow. In this case and also in the case of nested loops (line 32), it performs *fine-grained flow mapping* to check if there is a mapping for each sequence of control flow between the source-level loop and the binary-level loop (line 28 and 33). As the same analysis is used for the generation of



```
. . .
for (i = 0; i < size; i++) {
   y[i] = y[i] + 1;
}
 for (i = 0; i < size; i++) {
    x[i] = x[i] + y[i];
    if (w)
       y[i] = 0;
 }
. . .
```

```
if (w != 0) goto L28; else goto L27;
L27:
  i = 0;
L21:
  x[i] = x[i] + y[i];
  i = i + 1;
  if (i != size) goto L21; else goto L7;
L28:
  i = 0;
L3:
  x[i] = x[i] + y[i];
  y[i] = 0;
  i = i + 1;
  if (i != size) goto L3; else goto L7;
L7:
```

```
. . .
for (i = 0; i < size; i++) {
   y[i] = y[i] + 1;
}
if (w != 0) {
   i = 0;
   do{
      x[i] = x[i] + y[i];
      y[i] = 0;
      i = i + 1;
   } while (i != size);
} else{
   i = 0;
   do{
      x[i] = x[i] + y[i];
      i = i + 1;
   } while(i != size) ;
}
. . .
```

Fig. 3. An Example of POSC Generation

mapping information, more details about it will be provided in the next sub-section. If a flow mapping is not found, the *alteration of loop structure* effect is identified and the loop is transformed. The *partial loop unrolling* effect is identified by checking whether there are multiple identical control flows in the binary-level loop mapped to the whole source-level loop body.

In the example in Fig. 3, the change of loop number is identified for the second loop, so it is replaced by IR-level code, while the source code of the first loop can be kept. We obtain the IR-level code by modifying the compiler-generated IR that accounts for all machine-independent optimizations back to C code. As shown in Fig. 3(b), the IR-level C code is in three-address code form. We transform a regular IR-level unstructured loop with one loop entry and one exit back to a "do...while..." loop to increase the readability. The final generated POSC is shown in Fig. 3(c).

Algorithm 2 Fine-Grained Flow Mapping

```
 1: $\mathbf{S} \leftarrow$ DivideCFGIntoSegments($CFG$)
 2: for each segment $s \in \mathbf{S}$ do
 3:     $\mathbf{F} \leftarrow$ DivideIntoFlows($s$)
 4:     for each sequence of flow $f \in \mathbf{F}$ do
 5:         $\mathbf{L} \leftarrow$ GetInitialLineSet($f, debugInfo$)
 6:         for two parallel sequences of control flow $f_i \in \mathbf{F}$ and $f_j \in \mathbf{F}$ do
 7:             if $(\mathbf{L}_i \backslash \mathbf{L}_j) \bigcup (\mathbf{L}_j \backslash \mathbf{L}_i) = \Phi$ then
 8:                 $(\mathbf{L}_i, \mathbf{L}_j) \leftarrow$ RefineLineSet($\mathbf{L}_i, \mathbf{L}_j, debugInfo$)
 9:                 $(\mathbf{L}_i, \mathbf{L}_j) \leftarrow$ SetDifference($\mathbf{L}_i, \mathbf{L}_j$)
10:             end if
11:         end for
12:         $\mathbf{M} \leftarrow \mathbf{M} \bigcup \{(f, \mathbf{L})\}$
13:     end for
14: end for
```

C. Generation of Mapping Information

The mapping information is generated by means of fine-grained flow mapping. The pseudo code of the algorithm for fine-grained flow mapping is shown in Algorithm 2. The whole approach is also illustrated with an example in Fig. 4.

First, according to the loop levels the binary-level CFG is divided into segments, which are further divided into *sequences* of control flow. In a loop each sequence of control flow f starts with a loop entry or a conditional branch and ends with a loop exit or a conditional branch. In the example in Fig. 4, *bb2* is the loop entry and also contains a conditional branch, so *bb2* alone builds a sequence of control flow, while $\rightarrow bb3$ starts with a branch and ends with a loop exit. With flow mapping, f is mapped to a set of source lines $\mathbf{L}$. Therefore, each piece of mapping information m is denoted as $f \rightarrow \mathbf{L}$, which means that the timing information of basic blocks in f can be annotated before any source line $\ell \in \mathbf{L}$. The initial line set of each f is obtained by mapping the two ends of f into source lines. Each loop entry and each loop exit are mapped to the first line and the last line of the corresponding source-level loop, respectively. Then, the mapping between source-level conditional statements and binary-level conditional instructions can be accurately established through debugging information. For example, the line set $\mathbf{L}_2$ of *bb2* starts from the first line of the source-level loop (line 11), ends at the source line corresponding to the conditional branch in *bb2* (line 12). As *bb2* is within the loop, its timing information cannot be annotated before line 11. Therefore, we get $\mathbf{L}_2 = \{12\}$.

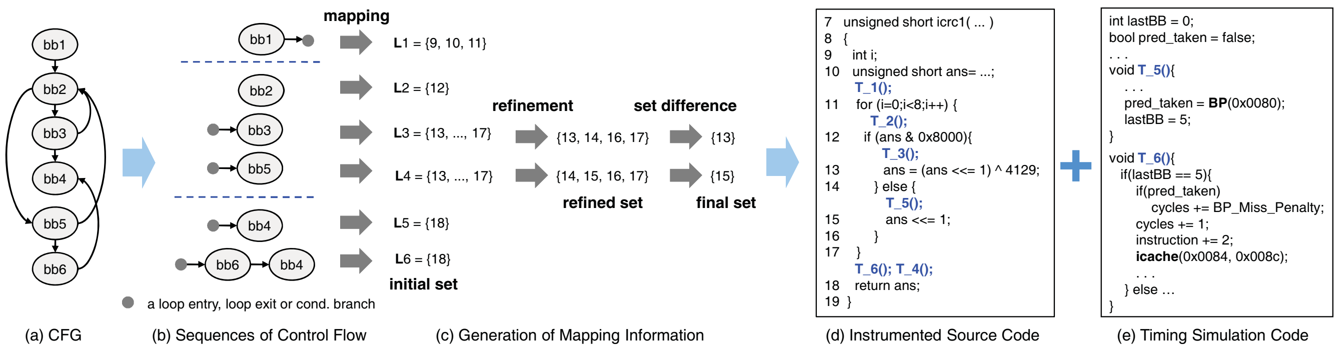
For two parallel sequences of control flow f_i and f_j that are mutually exclusive, the timing information of f_i must be annotated before any line in the *set difference* of $\mathbf{L}_i$ and $\mathbf{L}_j$, which is the set of elements in $\mathbf{L}_i$ but not in $\mathbf{L}_j$. Formally, the set difference is denoted as $\mathbf{L}_i \backslash \mathbf{L}_j = \{\ell \in \mathbf{L}_i | \ell \notin \mathbf{L}_i\}$. In the same way, the timing information of f_j must be annotated before $\ell \in \mathbf{L}_j \backslash \mathbf{L}_i$. However, in the example the line sets $\mathbf{L}_3$ and $\mathbf{L}_4$ of the two parallel sequences of control flow, $\rightarrow bb3$ and $\rightarrow bb5$, do not have set difference, i.e. $(\mathbf{L}_3 \backslash \mathbf{L}_4) \bigcup (\mathbf{L}_4 \backslash \mathbf{L}_3) = \Phi$. In this case we need to refine the initial line sets with the information from the source-level control flow. For two parallel sequences of control flow f_i and f_j there must be two parallel paths in source lines in $\mathbf{L}_i \bigcap \mathbf{L}_j$. Otherwise, the source-level control flow and the binary-level control flow do not match. This control flow mismatch would have been detected in the loop structure analysis and the loop would have been transformed to the IR level. In the example, line 13 and line 15 are in two parallel paths. Through the debugging information, it is found that no instruction in $\rightarrow bb3$ is generated from line 15, so line 15 is removed from $\mathbf{L}_3$. In the same way, line 13 is removed from $\mathbf{L}_4$. Thus, we can obtain the set difference between $\mathbf{L}_3$ and $\mathbf{L}_4$.

The two sequences of control flow $\rightarrow bb4$ and $\rightarrow bb6 \rightarrow bb4$ are also mutually exclusive, but there is no parallel path in the source code. Because they are not part of any loop, the algorithm leaves the source code unaltered. As the two sequences of control flow have different predecessor blocks, we can set the simulation of their predecessor blocks as the condition for their simulations. As shown in Fig. 4(e), only when the last simulated basic block is *bb5*, *bb6* will be simulated.

D. Back-Annotation of Timing Information

The low-level timing effects of a processor can be categorized into local timing effects and global timing effects. Pipeline effects are typical local timing effects. When loading instructions into a pipeline, adjacent instructions affect each other but remote instructions do not affect their execution. Local timing effects can be accurately estimated in the scope of a basic block using static timing analysis. Whereas, global timing effects are highly context-related and different execution paths set different context scenarios for a basic block. A cache access or a branch instruction will affect future cache accesses or branch predictions, so the cache and branch prediction effects are global timing effects. They need to be estimated at run time.

Correspondingly, the timing information to be back-annotated consists of two parts: 1) basic block latency that accounts for



```
7   unsigned short icrc1( ... )
8   {
9     int i;
10    unsigned short ans= ...;
       T_1();
11     for (i=0;i<8;i++) {
          T_2();
12       if (ans & 0x8000){
            T_3();
13          ans = (ans <<= 1) ^ 4129;
14        } else {
            T_5();
15          ans <<= 1;
16        }
17    }
       T_6(); T_4();
18    return ans;
19  }
```

```
int lastBB = 0;
bool pred_taken = false;
. . .
void T_5(){
   . . .
   pred_taken = BP(0x0080);
   lastBB = 5;
}
void T_6(){
  if(lastBB == 5){
     if(pred_taken)
        cycles += BP_Miss_Penalty;
     cycles += 1;
     instruction += 2;
     icache(0x0084, 0x008c);
     . . .
  } else …
}
```

Fig. 4. Our Approach for Generation of Mapping Information

the local timing effects and 2) code for dynamic simulation of the global timing effects. For example, for instruction cache simulation, a function call *icache(...)* is annotated, which will send instruction addresses of each basic block to a cache simulator for the estimation of cache hits and misses at simulation run-time. Data cache simulation is more complex because target data addresses cannot be resolved at compile time. A solution is to use host data addresses [2], [13]. In Fig. 4(e) code for the branch prediction simulation is also shown. As modeling these timing effects is out of the scope of this paper, we do not go into the details for the space reason.

Given the accurate mapping information obtained in the last step, the timing information can be accurately back-annotated. To get a clean instrumented program, we put the timing annotations of each basic block in a function and annotate the function call into the source code or POSC, as shown in Fig. 4(d) and (e).

V. Experimental Results

In the experiments, we evaluated the proposed SCI approach in terms of simulation accuracy and performance, by means of a set of benchmark programs that are often used for research on source-level simulation [2], [3], [4]. PowerPC 603e with 16KB instruction cache and 16KB data cache was selected as the target processor. A standard interpretive ISS was used as a reference to evaluate the accuracy of the proposed approach.

The simulation accuracy metrics include instruction count accuracy and cycle count accuracy. Instruction count accuracy depends solely on the instrumentation approach and therefore can directly reflect the accuracy of the SCI methods, while cycle count accuracy is also dependent on the performance model. Table I shows the estimated instruction counts of all programs. We used a cross-compiler ported from **gcc** of version 4.1.1. All programs were compiled with both the optimization level **-O2**, which is the standard optimization level for software development, and the highest optimization level **-O3** to test the heaviest optimizations.

We also simulated the programs using the basic Source-Level Simulation (SLS) approach illustrated in Fig. 1. This basic approach is referred to as **SLS** in the following discussion, while the proposed approach is called **SLS+** to be differentiated from the basic approach. The simulation results from SLS are used to show the largeness of the estimation error caused by the mapping programs. As SLS is not able to handle wrong debugging information and code motion, the simulation has an average instruction count error of 140.2%.

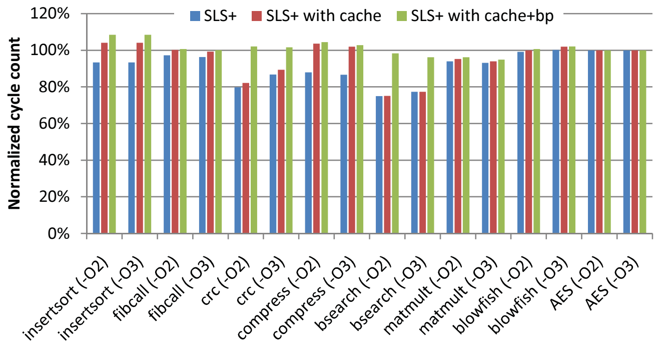


Fig. 5. Cycle Count Accuracy

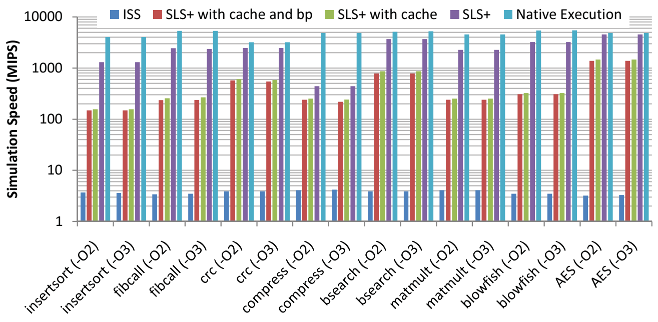


Fig. 6. Comparison of Simulation Performance

For the proposed approach, in order to show the efficiency of the Fine-Grained Flow Mapping (FGFM) and the Partly Optimized Source Code (POSC) for the respective mapping problems, we purposely evaluate them separately. First, we generate the simulation models of all programs from the original source code using mapping information obtained by means of FGFM. The simulation results of these models are shown in the column *FGFM* in the table. For 11 programs, the simulation models achieve very high accuracy (here the same program compiled with the two different optimization levels is counted as two programs). However, for the other 5 programs there are large errors, especially for *crc (-O3)*. For these programs, POSC generation is needed. By means of the loop structure analysis, 7 programs are identified to be transformed to POSC. The results from the simulation models generated by instrumenting the POSC are shown in the column *POSC+FGFM*. These simulation models achieve very high estimation accuracy. Only for *compress (-O2)* there is an error of 6.3%, caused by the different timing behavior of the source code and the POSC. This means that the IR-level C code might have (slightly) different timing behavior from the original

TABLE I
COMPARISON OF INSTRUCTION COUNT ACCURACY

	ISS	**SLS**		**SLS+ (*FGFM*)**		**SLS+ (*POSC+FGFM*)**	
Benchmark Programs	**Instruction Count**	**Instruction Count**	**Est. Error**	**Instruction Count**	**Est. Error**	**Instruction Count**	**Est. Error**
insertsort (-O2)	630	2063	227.5%	638	1.3%	630	0.0%
insertsort (-O3)	630	2063	227.5%	638	1.3%	630	0.0%
fibcall (-O2)	4511	27970	520.0%	4511	0.0%	–	–
fibcall (-O3)	4382	46665	964.9%	4382	0.0%	–	–
crc (-O2)	17201	25001	45.3%	17201	0.0%	–	–
crc (-O3)	12128	24607	102.9%	24607	102.9%	12042	-0.7%
compress (-O2)	3752	4082	8.8%	4005	6.7%	3987	6.3%
compress (-O3)	3681	7171	94.8%	4358	18.4%	3680	0.0%
bsearch (-O2)	59004	70010	18.7%	59010	0.0%	–	–
bsearch(-O3)	50003	59005	18.0%	61003	22.0%	50003	0.0%
matmult (-O2)	92019	91924	-0.1%	92019	0.0%	–	–
matmult (-O3)	89234	101261	13.5%	101514	13.8%	89234	0.0%
blowfish (-O2)	262435	262351	0.0%	262435	0.0%	–	–
blowfish (-O3)	160536	163287	1.7%	160536	0.0%	–	–
AES (-O2)	3624426960	3624571340	0.0%	3624442140	0.0%	–	–
AES (-O3)	3624394964	3624541250	0.0%	3624411080	0.0%	–	–

source code. That is an important reason why we do not transform all programs to the IR level. The entire proposed approach allows for simulating 14 out of 16 programs with 100% instruction count accuracy. The average error is only 0.4%.

The cycle count accuracy and the simulation performance are compared in Fig. 5 and Fig. 6, respectively. Cycle count accuracy is represented by the *normalized cycle count*:

$$Normalized\ cycle\ count = \frac{CycleCount_{SLS+}}{CycleCount_{ISS}} \times 100\% \qquad (1)$$

As mentioned, cycle count accuracy is based on instruction count accuracy and depends additionally on the accuracy of performance modeling. As the target processor has large caches and all programs are in small size, ignoring cache effects does not have a large impact on the accuracy. However, for simulation of practical applications modeling these timing effects is very important. Fig. 5 shows the normalized cycle counts of all programs obtained by simulation using SLS+, SLS+ with the cache model, and SLS+ with both cache and branch prediction (*bp*) models.

As shown in Fig. 6, the average simulation speed of SLS+ is 2550 MIPS. It achieves 680x speedup compared to the ISS and has only around 2x slowdown compared to the native execution. When the caches are simulated, the average speed is reduced to 520 MIPS, while the branch prediction simulation further reduces the speed to 487 MIPS. It is still much faster than the ISS.

VI. CONCLUSIONS

This paper presented an efficient SCI approach for automatic generation of source-level software TLMs. Especially, we addressed the mapping problems caused by compiler optimizations. We categorized the mapping problems into two types. For each problem, a solution was presented. We proposed *fine-grained flow mapping* for obtaining accurate mapping information. For programs with code that is heavily optimized by the compiler, we proposed to transform this part of code to optimized IR-level code resulting in Partly Optimized Source Code (POSC), so that there is a mapping between binary code and POSC. The experimental results demonstrated the accuracy and performance of the source-level TLMs generated by our approach.

REFERENCES

[1] 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.

[2] 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.

[3] Z. Wang, K. Lu, and A. Herkersdorf, "An approach to improve accuracy of source-level TLMs of embedded software," in *Proceedings of the Conference on Design, automation and test in Europe*, 2011, pp. 1–6.

[4] S. Stattelmann, O. Bringmann, and W. Rosenstiel, "Fast and Accurate Source-Level Simulation of Software Timing Considering Complex Code Optimizations," in *Proceedings of the 48th Annual Design Automation Conference (DAC'11)*, San Diego, California, 2011.

[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. Isshiki, D. Li, H. Kunieda, T. Isomura, and K. Satou, "Trace-driven workload simulation method for Multiprocessor System-On-Chips," in *Proceedings of the 46th Annual Design Automation Conference (DAC'09)*, 2009, pp. 232–237.

[7] Y. Hwang, S. Abdi, and D. Gajski, "Cycle-approximate retargetable performance estimation at the transaction level," in *Proceedings of the DATE*, 2008, pp. 3–8.

[8] 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.

[9] 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.

[10] 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.

[11] E. Cheung, H. Hsieh, and F. Balarin, "Framework for fast and accurate performance simulation of multiprocessor systems," in *Proceedings of IEEE International Workshop on High Level Design Validation and Test*, 2007, pp. 21–28.

[12] J.-Y. Lee and I.-C. Park, "Timed compiled-code simulation of embedded software for performance analysis of SOC design," in *Proceedings of the Design Automation Conference (DAC'02)*, 2002, pp. 293–298.

[13] T. Kempf, K. Karuri, S. Wallentowitz, G. Ascheid, R. Leupers, and H. Meyr, "A SW performance estimation framework for early system-level-design using fine-grained instrumentation," in *Proceedings of DATE*, 2006, pp. 468–473.

[14] 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.