Fast and Accurate Cache Modeling in Source-Level Simulation of Embedded Software

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

***Abstract*—Recently, source-level software models are increasingly used for software simulation in TLM (Transaction Level Modeling)-based virtual prototypes of multicore systems. A source-level model is generated by annotating timing information into application source code and allows for very fast software simulation.**

Accurate cache simulation is a key issue in multicore systems design because the memory subsystem accounts for a large portion of system performance. However, cache simulation at source level faces two major problems: (1) as target data addresses cannot be statically resolved during source code instrumentation, accurate data cache simulation is very difficult at source level, and (2) cache simulation brings large overhead in simulation performance and therefore cancels the gain of source level simulation. In this paper, we present a novel approach for accurate data cache simulation at source level. In addition, we also propose a cache modeling method to accelerate both instruction and data cache simulation. Our experiments show that simulation with the fast cache model achieves 450.7 MIPS (million simulated instructions per second) on a standard x86 laptop, 2.3x speedup compared with a standard cache model. The source-level models with cache simulation achieve accuracy comparable to an Instruction Set Simulator (ISS). We also use a complex multimedia application to demonstrate the efficiency of the proposed approach for multicore systems design.

I. Introduction

With the growing complexity of embedded applications, Multiprocessor System-on-Chips (MPSoC) are increasingly used as hardware platform of embedded systems. Design of such complex MPSoC systems usually starts at system level, in order to fix some important design issues in the early phases so that the design complexity of later phases can be significantly reduced. Typical design decisions to be made at system level include the number and type of processor cores and hardware accelerators, exploration of the memory architecture and the communication architecture, and task partitioning, mapping and scheduling etc. Most of these decisions rely on simulations that provide performance statistics.

Transaction Level Modeling (TLM) [1] has been proposed to increase the simulation speed by abstracting communication details. At transaction level the processing elements exchange data by calling common interfaces provided by the communication model. TLM is now a standard modeling style for system level design and is often associated with SystemC, a standard system-level design language. To obtain a system-level simulator of the whole system (often called a *Virtual Prototype (VP)*), processor models should be connected with TLM communication models. Traditional processor models are often built based on Instruction Set Simulators (ISS). An ISS-based processor model allows for cycle-accurate simulation of software. However, ISS-based VP has several limitations: (1) ISSs simulate too many unnecessary details of software execution and therefore are often very slow. By applying some advanced technique such as *just-in-time dynamic binary translation* state-of-the-art ISSs (JIT ISS) are faster than traditional interpretive ISSs. However, the simulation speed is still limited. Simulation speed of 33.5 MIPS is reported in [2]. Similar speeds are reported by commercial ISSs such as Synopsys CoMET VPM and nSim [3]. (2) They are very complex and need large effort to develop. It is difficult (if not impossible) to make VPs available in the early design phases for various design choices. (3) Difficulties in other issues, such as integrating ISSs into the SystemC environment and debugging.

Recently, Source-Level Simulation (SLS) has become a very popular technique for software simulation at system level [4], [5], [6], [7], [8], [9], [10], [11], [12]. A source-level model is actually application source code annotated with low-level timing information and is generated by source code instrumentation (SCI). Typical timing information includes wait statements that can cause simulation delays to capture the software execution times on the target processor and TLM interface calls that send transactions to simulate the communication and memory system. Compared with ISS-based VP, SLS-based VP has the following advantages: (1) Software simulation with SLS is much faster. (2) Source-level models need much less effort to generate. (3) Source-level models (annotated with approximate timing) can be available in very early design phases, even before instruction set and processor architecture are designed. (4) Debugging is much easier. Any debugging tools used for debugging C/C++ programs on PC can be used to debug software simulated on SLS-based VP.

To make SLS applicable for practical system design, a large body of recent work is focused on various issues of SLS: (1) accurate simulation of timing effects of processor's microarchitecture at source level [13], [14], [15], [16], (2) mapping between optimized binary code and source code for accurate back-annotation of timing information [7], [9], [11], [12], and (3) hybrid scheme for source level simulation of software with target dependent code [10].

Yet, there is still no efficient method for data cache simulation in SLS, which is however a critical issue in MPSoC design. The difficulty in data cache simulation at source level is that data memory accesses are only visible in low-level code and data addresses specified by register operations cannot be resolved at compile time. This paper is dedicated to address this problem. We propose methods for identifying memory accesses at source level and obtaining target data addresses for accurate data cache simulation. Since cache simulation will bring in a large overhead in simulation performance, we also propose a novel method to reduce this overhead.

The rest of this paper is organized as follows: Section II presents an overview of related work. Then, Section III gives a brief introduction to source code instrumentation. After that, the proposed cache modeling 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

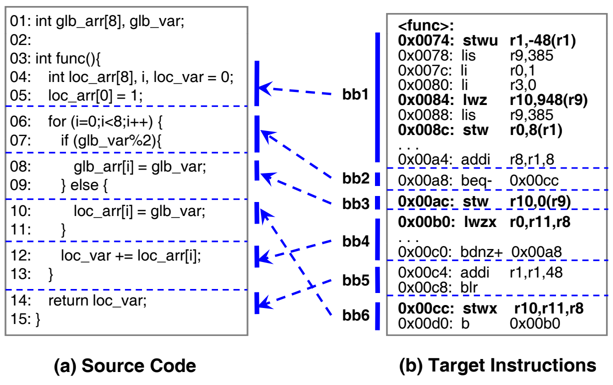
In this section we only discuss about previous work on cache simulation in SLS. For high-level data cache simulation, statistical models [5] have been used to randomly generate cache misses according to a pre-defined cache miss rate. This method is obviously

978-3-9815370-0-0/DATE13/©2013 EDAA

not accurate enough to capture the data access pattern of a specific program. In [16], code is added into source programs to calculate the target data addresses of global variables at run time. This method is limited to memory accesses to global variables, the target base addresses of which can be obtained from the symbol table.

To date, the most widely used way of high-level data cache simulation is using host data addresses [4], [15], [7], based on the assumption that the same data allocated in the host memory and in the target memory has very similar locality of reference. This is not true if the target compiler and the host compiler allocate data in different ways. Furthermore, it is also difficult to expose memory accesses in source code.

A recent work [14] proposes to combine source level simulation with static cache analysis. This approach is not suitable for design space exploration of multicore systems for two reasons: (1) the static cache analysis is pessimistic, i.e., it estimates only the worst case, and (2) it cannot obtain the data addresses. For design space exploration, compared to estimating cache misses or hits, it is of more importance for the processor models to generate transactions over the communication architecture to the target memory, in order to evaluate the whole architecture. This can be precisely done, only when the target addresses are known.



```
01: int glb_arr[8], glb_var;
02:
03: int func(){
04:   int loc_arr[8], i, loc_var = 0;
05:   loc_arr[0] = 1;
06:   for (i=0;i<8;i++) {
07:     if (glb_var%2){
08:       glb_arr[i] = glb_var;
09:     } else {
10:       loc_arr[i] = glb_var;
11:     }
12:     loc_var += loc_arr[i];
13:   }
14:   return loc_var;
15: }
```

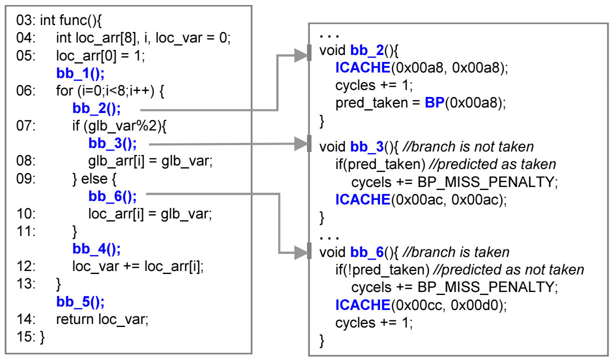
```
<func>:
0x0074:  stwu   r1,-48(r1)
0x0078:  lis    r9,385
0x007c:  li     r0,1
0x0080:  li     r3,0
0x0084:  lwz    r10,948(r9)
0x0088:  lis    r9,385
0x008c:  stw    r0,8(r1)
. . .
0x00a4:  addi   r8,r1,8
0x00a8:  beq-   0x00cc
0x00ac:  stw    r10,0(r9)
0x00b0:  lwzx   r0,r11,r8
. . .
0x00c0:  bdnz+  0x00a8
0x00c4:  addi   r1,r1,48
0x00c8:  blr
0x00cc:  stwx   r10,r11,r8
0x00d0:  b      0x00b0
```

Fig. 1. Example Program

III. Source Level Simulation

We will explain Source Level Simulation (SLS) by demonstrating how it applies to the code in Figure 1(a). The idea of SLS is to use application source code directly as function model. Since native execution of source code by itself cannot provide estimates of execution time, annotation of timing information is needed. Automatic generation of source-level simulation models consists of three steps: timing analysis, mapping and timing annotation. Timing analysis is aimed to estimate the execution time of each basic block. Some timing effects such as the pipeline effects can be statically analyzed to avoid repetitive dynamic simulation, while some other timing effects such as the cache and branch prediction effects are highly context-related and therefore must be dynamically simulated.

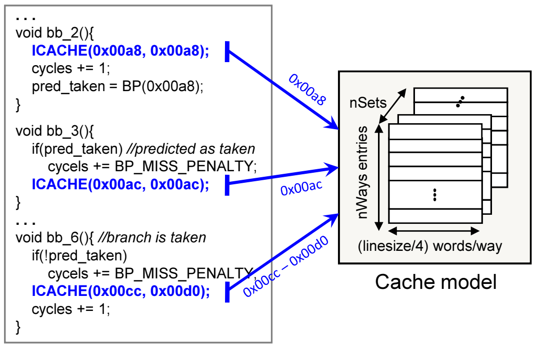
The back-annotation of timing information relies on the mapping between binary code and source code. We adopt the method from [12] for generating accurate mapping information for compiler-optimized programs. For the example in Figure 1, the mapping established between the source code and the basic blocks of binary code is illustrated with dashed lines. Following this mapping the source code is then annotated a call to a function containing the timing information of the corresponding basic block. Finally, we obtain the instrumented source code as shown in Figure 2. For example, *bb_2()*



```
03: int func(){
04:   int loc_arr[8], i, loc_var = 0;
05:   loc_arr[0] = 1;
      bb_1();
06:   for (i=0;i<8;i++) {
        bb_2();
07:     if (glb_var%2){
          bb_3();
08:       glb_arr[i] = glb_var;
09:     } else {
          bb_6();
10:       loc_arr[i] = glb_var;
11:     }
        bb_4();
12:     loc_var += loc_arr[i];
13:   }
      bb_5();
14:   return loc_var;
15: }
```

```
. . .
void bb_2(){
  ICACHE(0x00a8, 0x00a8);
  cycles += 1;
  pred_taken = BP(0x00a8);
}
void bb_3(){ //branch is not taken
  if(pred_taken) //predicted as taken
    cycles += BP_MISS_PENALTY;
  ICACHE(0x00ac, 0x00ac);
}
. . .
void bb_6(){ //branch is taken
  if(!pred_taken) //predicted as not taken
    cycles += BP_MISS_PENALTY;
  ICACHE(0x00cc, 0x00d0);
  cycles += 1;
}
```

Fig. 2. Instrumented Source Code



```
. . .
void bb_2(){
  ICACHE(0x00a8, 0x00a8);
  cycles += 1;
  pred_taken = BP(0x00a8);
}
void bb_3(){
  if(pred_taken) //predicted as taken
    cycles += BP_MISS_PENALTY;
  ICACHE(0x00ac, 0x00ac);
}
. . .
void bb_6(){ //branch is taken
  if(!pred_taken)
    cycles += BP_MISS_PENALTY;
  ICACHE(0x00cc, 0x00d0);
  cycles += 1;
}
```

Fig. 3. Instruction Cache Simulation

contains the timing information of bb_2. The delay value that accounts for the pipeline effects is accumulated by a variable *cycles* and the function calls **ICACHE**(...) and **BP**(...) are for dynamic simulation of the instruction cache and branch prediction effects, respectively. Figure 3 illustrates instruction cache simulation. At simulation run-time **ICACHE**(...) will send the instruction addresses to a cache simulator, which then estimates whether a cache hit or a miss occurs according to the current cache content. Unlike instruction addresses which are known at compile time, data addresses are specified by register operations and cannot be statically resolved. Therefore, data cache simulation is not as easy as instruction cache simulation. The problem of data cache simulation and our solutions are discussed in the next section.

IV. The Proposed Approach for Data Cache Modeling

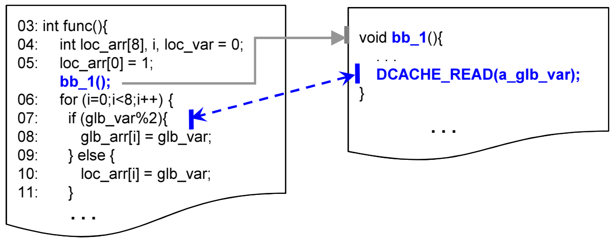
A. Source Code Instrumentation for Data Cache Simulation

1) Identifying Memory Accesses: For correct source code instrumentation for data cache simulation, we need first to identify which source line will generate how many memory accesses. For most RISC processors, the actual memory accesses are visible only in the binary code in form of load/store instructions. Due to the presence of compiler optimizations, it is often error-prone to guess which source variables will generate load/store instructions.

```
for (i=0;i<8;i++) {
     DCACHE_READ(a_glb_var);
   if (glb_var%2){
     DCACHE_READ(a_glb_var);
     glb_arr[i] = glb_var;
   } else {
     DCACHE_READ(a_glb_var);
     loc_arr[i] = glb_var;
   }
   ...
}
```

For example, in the above instrumented code, the global variable *glb_var* is supposed to generate memory accesses, and therefore, **DCACHE_READ()** is annotated at each use of *glb_var*. These **DCACHE_READ()** are annotated in the loop and will be executed for 8 times. Totally, data cache is simulated 24 times for *glb_var*. However, in reality, the load instruction generated by *glb_var* is moved out of the loop by the compiler and will be executed only once. Obviously, this identification of memory accesses at source level will result in a large simulation error.

Our solution is to identify the memory accesses at binary level. The method is illustrated in Figure 4 using the same example. Since the load instruction generated by *glb_var* is in *bb1*, only one **DCACHE_READ()** is annotated in *bb_1()*, independent of the position of the source variable. The relation between the source variable and the load instruction (illustrated as the dashed line) is only used to obtain the address of the memory access *a_glb_var*. This relation can be obtained from the debugging information. In this example, *glb_var* is a global variable, so *a_glb_var* can be statically resolved and obtained from the symbol table. In other cases, the address can be an expression that generates data addresses at runtime. This issue is discussed in the next sub-section.



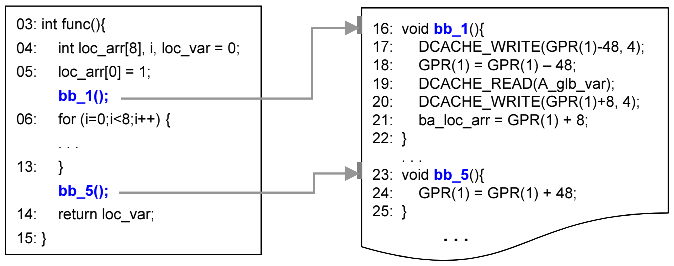
```
03: int func(){
04:    int loc_arr[8], i, loc_var = 0;
05:    loc_arr[0] = 1;
       bb_1();
06:    for (i=0;i<8;i++) {
07:      if (glb_var%2){
08:         glb_arr[i] = glb_var;
09:      } else {
10:         loc_arr[i] = glb_var;
11:      }
       . . .
```

```
void bb_1(){
   . . .
   DCACHE_READ(a_glb_var);
}
   . . .
```

Fig. 4. Method for Identifying Memory Accesses

2) Obtaining Data Addresses: Typical ways of memory allocation for different variables include static memory allocation, dynamic memory allocation and automatic memory allocation. According to the complexity of addressing, we group all memory accesses into six categories:

- **Category 1:** Accesses made in a function's prologue and epilogue for building up and cleaning up the call stack frame. The memory locations referenced by these accesses change with the location of the stack frame and cannot known statically. In addition, these memory accesses are done automatically and hidden from the programmer.
- **Category 2:** Accesses to a local scalar variable that could not fit into the designated registers or to a local composite variable (i.e. data structure) with a constant index or a fixed pointer. The data of such variables is allocated in the stack frame of the function call. The memory locations change with the location of the stack frame.
- **Category 3:** Accesses to a global/static scalar variable or accesses to a (global/static) composite variable with a constant index or a fixed pointer will always reference the same memory location. Since memory for global/static variables is statically allocated, the memory locations of the accesses in this category can be known at compile-time.
- **Category 4:** Accesses to a global/static composite variable with a variable index or a moving pointer. The base address of a global/static composite variable can be known at compile-time, but the offset to the base address changes dynamically.
- **Category 5:** Accesses to a local composite variable with a variable index or a moving pointer. Since local composite variables are usually allocated in the stack frame, the base address of a local composite variable changes with the location of the stack frame. Moreover, the offset to the base address also changes dynamically.
- **Category 6:** Accesses to a dynamically allocated variable. Its base address is known only at runtime after it is allocated in the heap. If it is a composite variable and is accessed by a moving pointer, the offset to the base address also changes dynamically.



```
03: int func(){
04:    int loc_arr[8], i, loc_var = 0;
05:    loc_arr[0] = 1;
       bb_1();
06:    for (i=0;i<8;i++) {
       . . .
13:    }
       bb_5();
14:    return loc_var;
15: }
```

```
16:  void bb_1(){
17:     DCACHE_WRITE(GPR(1)-48, 4);
18:     GPR(1) = GPR(1) – 48;
19:     DCACHE_READ(A_glb_var);
20:     DCACHE_WRITE(GPR(1)+8, 4);
21:     ba_loc_arr = GPR(1) + 8;
22:  }
     . . .
23:  void bb_5(){
24:     GPR(1) = GPR(1) + 48;
25:  }
       . . .
```

Fig. 5. Simulating the Stack Pointer for Obtaining Data Addresses

In the example in Figure 1, there are 6 load/store instructions. The first one (*stwu r1,-48(r1)*) is used to build a stack frame and belongs to Category 1. As discussed before, the second one (*lwz r10,948(r9)*) is generated by a global scalar variable (*glb_var* at line 7) and therefore belongs to Category 3. The third load/store instruction (*stw r0,8(r1)*) is generated by access to a local array with a constant index (*loc_arr[0]* at line 5) and therefore belongs to Category 2. The fourth one is generated by access to a global array with a variable index *i* (*glb_arr[i]* at line 8) and is in Category 4. The last two load/store instructions are generated by accesses to a local array with a variable index *i* (*loc_arr[i]* at line 10 and line 12) and belong to Category 5.

Previous work considers only memory accesses made by global variables [16] or uses host data addresses for data cache simulation [15], [4], [7], based on the assumption that the same data allocated in the host memory and in the target memory has very similar locality of reference. Using the former method, only memory accesses in Category 3 and Category 5 are covered, while using the latter method, there are two problems: (1) the difference between the locality of target data and host data will lead to cache simulation errors, and (2) as discussed before, it is very error-prone to identify memory accesses at source level. In addition, memory accesses made in a function's prologue and epilogue are not visible in the source code and therefore cannot be taken into account in the previous approaches.

Our approach is aimed to obtain the target addresses of memory accesses in all the aforementioned categories. Except for Category 3, target data addresses of other memory accesses cannot be statically resolved. Therefore, instead of constant values, we need to annotate code to dynamically generate data addresses at simulation runtime. For memory accesses in Category 1 and Category 2, our solution is based on the observation that the memory locations of local variables change due to dynamic allocation of the stack frame every time a function is called, whereas the local variables (those that do not fit into registers) are usually assumed to reside at a fixed offset from the stack pointer. The offset of a local variable to the stack pointer can be obtained by decoding load/store instructions. Therefore, if the content of the stack pointer can be maintained in the source-level simulation, the target data addresses of the local variables can be obtained. Figure 5 illustrates the code annotated for simulating operations on the stack pointer. Here, we demonstrate our method with PowerPC processors, but similar rules can be established for other widely used RISC processors. In PowerPC processors, GPR 1 is dedicated as the stack pointer. In *bb_1()*, which contains the prologue of the function, the *stwu* instruction (store word with update) stores the old stack pointer and grows the stack to build a new stack frame. Code at line 17 and 18 simulate this operation. The address of the *stw* instruction in Category 2 is obtained by adding an offset *8* to GPR 1. Therefore, *GPR(1)+8* is annotated to generate the address at runtime. In *bb_5()*, the *addi* instruction shrinks the stack and the stack frame is deleted before the return of the function.

For memory accesses in Category 4 we can obtain the base addresses of composite variables from the symbol table and calculate at runtime the target addresses by adding dynamically obtained offsets to the base addresses. Array variables are the most widely used composite variables. In an array the data elements are stored contiguously. The offset of the currently pointed element to the base address can be calculated as follows:

$$Offset_{target} = (Addr_{host} - BaseAddr_{host}) * \frac{size_{target}}{size_{host}} \quad (1)$$

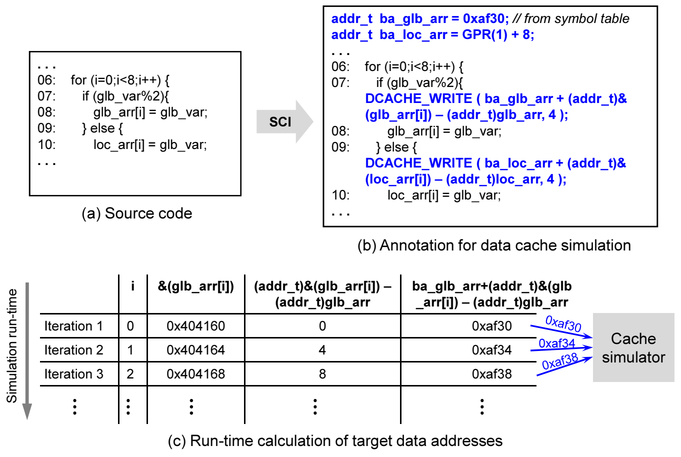
where $(Addr_{host} - BaseAddr_{host})$ calculates the offset of the host address, which is then translated to the offset of the target address considering the difference of the size of the same data type on the target processor ($size_{target}$) and on the host machine ($size_{host}$). For memory accesses in Category 4 $BaseAddr_{target}$ is statically known and both $Addr_{host}$ and $BaseAddr_{host}$ can be obtained at runtime by using the dereference operator (&), we can dynamically calculate $Addr_{target}$ for data cache simulation. This approach is illustrated using the global array *glb_arr[ ]* in Figure 6 (assume that $size_{host} = size_{target}$).

For some other data structures where the memory for the data elements is not contiguously allocated, address offsets obtained during native execution are not the same as the target address offsets, leading to a certain estimation error. However, compared to other methods that directly use the host addresses, our method can better capture the locality of reference of the target data. Furthermore, such data structures are not heavily used in embedded applications, compared to arrays.

The addressing of memory accesses in Category 5 is more complicated. It involves both the base address that changes with the call stack frame and the dynamically changing offset. The problem can be solved by combining the solution for Category 2 and the one for Category 4. Namely, we can obtain the base address at runtime by simulating the stack pointer and calculate the offset by referring to the host addresses. This approach is also illustrated in Figure 6 using the global array *glb_arr[ ]*.

For memory accesses in Category 6, we need to simulate the heap pointer to obtain the base address of each newly allocated variable. Unlike the simulation of the stack pointer, the operations on the heap pointer are not visible in the binary code. We use a *heap manager*, which is part of our runtime simulation system, to manage the heap pointer. At runtime, when a new variable is allocated, the heap manager assigns the current value of the heap pointer to the base address of this variable and then grows the heap by the size of the variable. When a variable is freed, the heap pointer moves down accordingly. The offset to the base address can be obtained by the method introduced before.

Limitation: a limitation of our method is that it has difficulty in dealing with pointer aliasing. If a variable is accessed with a pointer that aliases it, manual analysis is needed to find out which variable this pointer points to.


```
. . .
06:   for (i=0;i<8;i++) {
07:     if (glb_var%2){
08:       glb_arr[i] = glb_var;
09:     } else {
10:       loc_arr[i] = glb_var;
. . .
```

(a) Source code

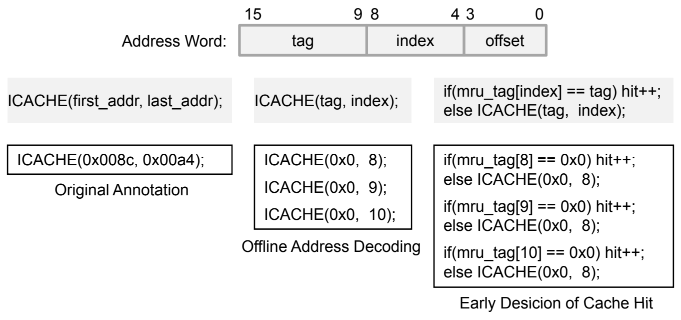
```
addr_t  ba_glb_arr = 0xaf30; // from symbol table
addr_t  ba_loc_arr = GPR(1) + 8;
. . .
06:   for (i=0;i<8;i++) {
07:     if (glb_var%2){
      DCACHE_WRITE ( ba_glb_arr + (addr_t)&
      (glb_arr[i]) – (addr_t)glb_arr, 4 );
08:       glb_arr[i] = glb_var;
09:     } else {
      DCACHE_WRITE ( ba_loc_arr + (addr_t)&
      (loc_arr[i]) – (addr_t)loc_arr, 4 );
10:       loc_arr[i] = glb_var;
. . .
```

(b) Annotation for data cache simulation

	i	&(glb_arr[i])	(addr_t)&(glb_arr[i]) – (addr_t)glb_arr	ba_glb_arr+(addr_t)&(glb_arr[i]) – (addr_t)glb_arr
Iteration 1	0	0x404160	0	0xaf30
Iteration 2	1	0x404164	4	0xaf34
Iteration 3	2	0x404168	8	0xaf38
⋮	⋮	⋮	⋮	⋮

(c) Run-time calculation of target data addresses

Fig. 6. Approach for Obtaining Target Addresses of Array Variables



```
ICACHE(first_addr, last_addr);
```

```
ICACHE(0x008c, 0x00a4);
```

Original Annotation

```
ICACHE(tag, index);
```

```
ICACHE(0x0,  8);
ICACHE(0x0,  9);
ICACHE(0x0,  10);
```

Offline Address Decoding

```
if(mru_tag[index] == tag) hit++;
else ICACHE(tag,  index);
```

```
if(mru_tag[8] == 0x0) hit++;
else ICACHE(0x0,  8);
if(mru_tag[9] == 0x0) hit++;
else ICACHE(0x0,  8);
if(mru_tag[10] == 0x0) hit++;
else ICACHE(0x0,  8);
```

Early Desicion of Cache Hit

Fig. 7. Techniques for Fast Cache Modeling

B. Fast Cache Modeling

At runtime, the annotated function calls will trigger simulation of the caches in an interpretive manner. The pseudo code of instruction cache simulation is presented in Algorithm 1. As shown, for each cache line the whole loop is executed once. This will introduce a large overhead to simulation performance.

Algorithm 1 Instruction Cache Simulation

```
1:  Definitions:
2:  nWays: cache associativity
3:  nSets: the number of cache sets
4:  linesize: cache line size
5:  Input:
6:  first_addr: the first instruction address of a basic block
7:  last_addr: the last instruction address of a basic block
8:
9:  $address \leftarrow$ AlignAddress($first\_addr$)
10: while $address \leq last\_addr$ do
11:     $(tag, index) \leftarrow$ DecodeAddress($address$)
12:     $way \leftarrow$ Search(tag, index)
13:     if $way == 0$ then
14:         cache miss
15:         wait($cycles * T$) //T: time of each CPU cycle
16:         $cycles \leftarrow 0$
17:         Send out a transaction for memory access
18:         $way \leftarrow$ ReplaceLine($tag, index$)
19:     end if
20:     $address += linesize$
21: end while
22: UpdateAge($way, index$)
```

To reduce this overhead, we apply two techniques in the cache model. The first one is to move the workload of address decoding from simulation runtime to instrumentation time. This is feasible for all the instruction cache accesses and part of data cache accesses, the addresses of which are known during the instrumentation. This

TABLE I
COMPARISON OF SIMULATION RESULTS BETWEEN ISS AND SLS

	ISS			SLS (with cache model)			
	Data cache hits	**Data cache misses**	**Cycle count**	**Data cache hits**	**Data cache misses**	**Cycle count**	**Cycle count error**
nsichneu	2024	7	12461	2024	7	11385	-8.6%
crc	1207	36	12830	1207	36	10726	-16.4%
compress	1238	61	3957	1238	61	3927	-0.8%
ludcmp	517	27	3461	517	27	3276	-5.3%
blowfish	56718	268	169145	56718	268	168898	-0.1%
matmult	26682	154	101748	26682	154	97458	-4.2%
coremark	93152456	87	371302183	93152335	85	351155007	-5.4%

technique is illustrated in Figure 7. Addresses are decoded to get *tag* and *index* bits during source code instrumentation and function calls *ICACHE(tag, index)* are annotated instead. For instruction cache simulation, if the instruction addresses of a basic block cross *n* cache lines, *n* calls to the cache model will be annotated. As shown in the example in Figure 7, the cache line size is 16 bytes and the basic block, the instruction addresses of which range from *0x008c* to *0x00a4*, crosses three cache lines, so three function calls are annotated.

The second technique is to make early decision of cache hits to bypass *tag* searching in the cache model. The idea is based on the fact that if the *tag* of the current address is the same as the most recently used *tag* in the same cache set, this cache access will certainly hit cache. This technique is also illustrated in Figure 7. We create an array *mru_tag[nSets]* to record the most recently used *tag* of each cache set. If data cache and instruction cache are separate, two such arrays are needed. If the condition *mru_tag[index] == tag* evaluates to true, then the cache simulation can be skipped. If the target data cache uses the *write back* scheme, each cache line has a *dirty bit* and each write access may have to update the corresponding dirty bit. In this case, we need an extra array *mru_dBit[nSets]* to record the dirty bit of the most recently accessed cache line of each cache set. For each cache write, only when both conditions, *mru_tag[index] == tag* and *mru_dBit[index] == 1*, evaluate to true, we can skip the cache simulation. In the case of a cache hit, if it is found that *mru_dBit[index] == 0*, we still need to access the cache model to find out the cache line and set the dirty bit. The combination of the two techniques will significantly improve the cache simulation speed.

V. EXPERIMENTAL RESULTS

In the experiments, we first evaluate the proposed cache simulation approach in terms of simulation accuracy and speed by means of a set of benchmark programs. A PowerPC processor with 16KB instruction cache and 16KB data cache was selected as the target processor. Then, we use a case study of a video application to demonstrate how the proposed approach facilitates memory architecture exploration in MPSoC design.

A. Evaluation of the Proposed Approach for Cache Simulation

Since our major contribution is the method for data cache simulation. We show only the data cache simulation results in Table I. MicroLib PowerPC ISS [17] was used as a reference to evaluate the simulation accuracy. It is an functional interpretive ISS. We extended it with a microarchitecture model for performance simulation. As shown, seven benchmark programs were tested. *CoreMark* is an open-source EEMBC benchmark [18], *Blowfish* is a well-known cipher, and the other programs are from Mälardalen WCET Benchmarks [19]. We select them because we need benchmarks with source code completely available. For programs with many library calls or target dependent code, a hybrid simulation scheme as proposed in [10] is needed. Although most selected programs do not stress much data cache, they are enough to demonstrate the accuracy of our cache modeling method.

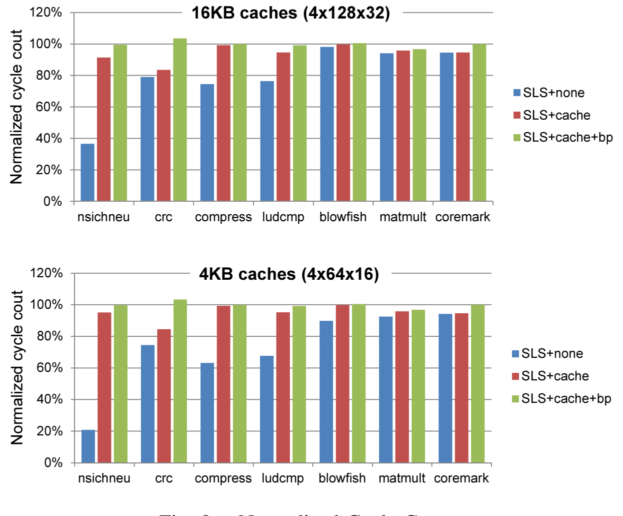


Fig. 8. Normalized Cycle Counts

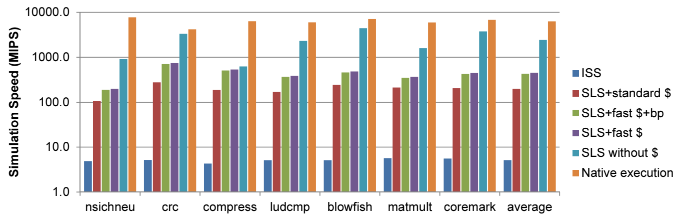


Fig. 9. Comparison of Simulation Speeds

As shown in Table I, our SLS allows for very accurate estimation of data cache hits and misses. The geometric mean of the cycle count error is 2.8%, mainly caused by ignoring branch prediction effects. When branch prediction effects are also simulated, the error is reduced to 1.2% on average. When cache effects are ignored, then the geometric mean of the error becomes 12.6%. Figure 8 shows the *normalized cycle counts* of all programs obtained by simulation using only SLS, SLS with cache model, and SLS with both cache and branch prediction models. The *normalized cycle count* ($= \frac{CycleCount_{SLS}}{CycleCount_{ISS}} \times 100\%$) represents the simulation accuracy. As shown in Figure 8, the importance of cache simulation increases when the cache size becomes smaller. When the cache size is reduced from 16KB to 4KB, ignoring cache effects results in the geometric mean of the error of 19.3%. Here, we assume a cache miss leads to a fixed delay of 10 cycles. When this delay is larger, the cache simulation will be more important.

Figure 9 illustrates the simulation speed of the ISS, SLS, SLS with a standard cache ($) model (cf. Algorithm 1), SLS with the proposed fast cache model, SLS with both cache and branch prediction (*bp*)

TABLE II
SIMULATION OF THE SEQUENTIAL PROGRAM

	ISS	SLS	Sim. Error
Simulated cycle count	55217100	56120311	1.6%
Data cache hits	10142102	10145222	0.0%
Data cache misses	526022	522902	-0.6%

	ISS	SLS	Speedup
Simulation time	10.16s	0.12s	84.5

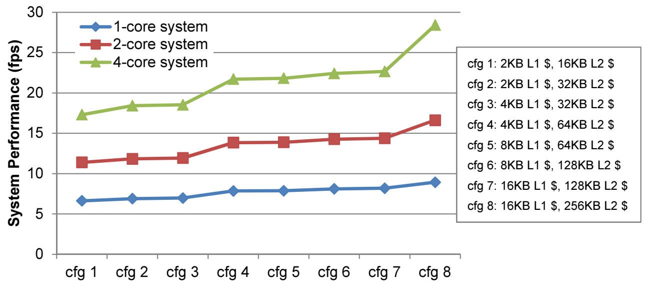


Fig. 10. Memory Architecture Exploration for the Video Downscaler

models, and also the native execution speed. The measurement was done on a laptop equipped with Intel Core i5-3360M CPU@2.80GHz and 4GB RAM. The average simulation speed of SLS is 2421 MIPS. It achieves 475x speedup compared to the ISS (5.1 MIPS on ave.). Although this ISS cannot represent the state-of-the-art, the speed of SLS is also much faster than a fast JIT ISS with microarchitecture modeled (20 – 50 MIPS). When caches are simulated using a standard cache model, the average speed of SLS is reduced to 200 MIPS. Using our fast cache model instead increases the simulation speed to 451 MIPS and achieves 2.3x speedup, compared to the standard cache model.

B. Case Study of MPSoC Design

The proposed cache simulation method is applied in memory system exploration of an MPSoC to justify its efficiency. The application is a video Downscaler, which converts the original CIF video frames (352x288) to the frames with a lower resolution (132x128). It consists of 8 tasks. The tasks are mapped to a SMP architecture with 3 PowerPC cores. Each core has a L1 data cache and instruction cache. All the cores share a L2 cache.

Given a sequential implementation of the application, we use our source code instrumentation approach to annotate the timing information into the source code. Table II shows the accuracy of the source-level model compared to the ISS. Then, the application is partitioned into a set of tasks which are mapped to the processor cores.

So far, we always assume a fixed delay for a cache miss. Now that the whole system is simulated, the delay caused by a cache miss is dynamically determined by the bus and memory simulation. In Figure 10 we show the performance of the system with different configurations. The performance is represented with the video processing rate in frames per second (fps). The simulation time of processing 10 video frames ranges from 56 seconds to 159 seconds, mainly depending on the configuration of the memory sub-system. More cache misses lead to more time-consuming simulations of the bus and the memory. This fast simulation speed allows for evaluation of many system configurations in a very short time.

VI. CONCLUSIONS

This paper presented an efficient approach for accurate data cache simulation in source-level simulation (SLS). The most difficult issue in source-level data cache simulation is the way of obtaining data addresses. We grouped the memory accesses into six categories and proposed methods for obtaining data addresses of memory accesses in each category. In addition, we also presented a fast cache model to reduce the simulation slowdown due to the cache simulation. The experiments showed high simulation accuracy as well as high performance (x2.3 speedup) of the proposed cache model. SLS with the proposed cache model can achieve an average speed of 451 MIPS.

REFERENCES

[1] "OSCI TLM-2.0 Language Reference Manual," Online available at http://www.accellera.org/, 2009.

[2] I. Böhm, B. Franke, and N. Topham, "Cycle-accurate performance modelling in an ultra-fast just-in-time dynamic binary translation instruction set simulator," in *Proceedings of the International Conference on Embedded Computer Systems (IC-SAMOS)*, vol. 10, 2010, pp. 1–10.

[3] "Synopsys," http://www.synopsys.com/.

[4] 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.

[5] Y. Hwang, S. Abdi, and D. Gajski, "Cycle-approximate retargetable performance estimation at the transaction level," in *Proceedings of the conference on Design, automation and test in Europe (DATE'08)*, 2008.

[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] 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.

[8] 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.

[9] 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 (DATE'11)*, 2011.

[10] Z. Wang and J. Henkel, "Hycos: hybrid compiled simulation of embedded software with target dependent code," in *Proceedings of the eighth IEEE/ACM/IFIP international conference on Hardware/software codesign and system synthesis (CODES-ISSS 2012)*, 2012, pp. 133–142.

[11] 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.

[12] Z. Wang and J. Henkel, "Accurate source-level simulation of embedded software with respect to compiler optimizations," in *Proceedings of the conference on Design, automation and test in Europe (DATE'12)*, 2012.

[13] 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.

[14] S. Stattelmann, G. Gebhard, C. Cullmann, O. Bringmann, and W. Rosenstiel, "Hybrid source-level simulation of data caches using abstract cache models," in *Proceedings of the conference on Design, automation and test in Europe (DATE'12)*, 2012.

[15] E. Cheung, H. Hsieh, and F. Balarin, "Memory subsystem simulation in software tlm/t models," in *Proceedings of the 2009 Asia and South Pacific Design Automation Conference*, ser. ASP-DAC '09, 2009, pp. 811–816.

[16] A. Pedram, D. Craven, and A. Gerstlauer, "Modeling cache effects at the transaction level," in *Analysis, Architectures and Modelling of Embedded Systems*, vol. 310. Springer Boston, 2009, pp. 89–101.

[17] D. Perez, G. Mouchard, and O. Temam, "MicroLib: A case for the quantitative comparison of micro-architecture mechanisms," in *Proceedings of the 37th annual IEEE/ACM International Symposium on Microarchitecture*. IEEE Computer Society, 2004, pp. 43–54.

[18] "CoreMark Benchmark," http://www.coremark.org/.

[19] "Mälardalen WCET Benchmarks," http://www.mrtc.mdh.se/projects/wcet/benchmarks.html.