Fast and Accurate TLM Computation Model Generation Using Source-Level Timing Annotation

Kai-Li Lin, Pei-Jia Lin, Cheng-Kang Lo, Ren-Song Tsay
Logos Advanced System Lab,
Dept. of Computer Science, National Tsing-Hua University, Taiwan
{klin, pjl, cklo, rstsay}@cs.nthu.edu.tw

Abstract—While Transaction Level Modeling (TLM) approach is widely adopted now for system modeling and simulation speed improvement, timing estimation accuracy often is compromised. To have reliable and accurate estimation results at system level, this paper proposes a timing annotation method for accurate TLM computation model generation considering processor architecture with pipeline and cache structures, which are challenging but critical to accurate timing estimation. The experiments show that our results are within 2% of cycle accurate results and the approach is three orders faster than conventional ISS approaches.

1. INTRODUCTION

TLM computation model has recently been applied to software performance evaluation [1]. Instead of conventional Instruction Set Simulator (ISS) simulation approaches, which simulate with target assembly code instruction-by-instruction [6] [8], computational TLM approach takes higher abstraction model directly on source program. The source program is annotated with execution time information of target processor and then compiled and executed natively on a host machine for fast simulation performance. In this way, total timing can be estimated by simply summing up the annotated timing numbers along execution path, while at the same time the functionality is accurately preserved.

Nevertheless, without proper partitioning of the source program into estimation units, the timing estimation can be very inaccurate. Additionally, critical timing factors, such as code optimization, pipeline saturation effects etc., are often ignored in existing timing annotation methods and hence give rise to inaccurate results [4] [5].

To address this issue, the authors propose a timing annotation method which uses basic block as a natural estimation unit for the source program and annotates timing accordingly. Each basic block’s execution time is then calculated according to the cross-compiled (and optimized) target assembly code. In addition, by analyzing the assembly code through a target processor model, this approach can take into account features which influence program execution time, such as pipeline saturation, branch prediction and cache architecture. As a result, our method generates accurate TLM computation models which give results within 2% error from cycle accurate simulation while run at speed close to three orders faster than cycle accurate simulation.

2. ESTIMATION UNIT

In this section, we give an observation of an ideal estimation unit of our approach. We take a bubble sort example and illustrate the effects to timing accuracy of different estimation unit sizes. In Fig. 1(a), each source program line is taken as an estimation unit and a fixed delay time is annotated to estimate the unit’s execution.

![Fig. 1: (a) A program line as an estimation unit; (b) the whole program as an estimation unit.](image)

![Fig. 2: (a) A source program is partitioned into six basic blocks; (b) The corresponding control flow graph of the source program.](image)
number, the estimation unit cannot include multiple execution paths. Therefore, the ideal estimation unit is essentially equivalent to the concept of basic block used in compiler optimization. Formally, a basic block is defined as a sequence of consecutive statements in which flow of control enters at the beginning and leaves at the end without halt or possibility of branching except at the end of this basic block [3].

For implementation, a source program is analyzed and represented by a Control Flow Graph (CFG). Each node of the graph is a basic block and each directed edge implies an execution order of the corresponding from-and-to basic blocks. Each program has one entry node that is where the execution starts. For example, the bubble sort code has six basic blocks as shown in Fig. 2(a) and the corresponding CFG is shown in Fig. 2(b).

2.1 Timing Annotation Flow

Our timing annotation flow is illustrated in Fig. 3 and it consists of two major parts: basic block analysis partitions a source program into basic blocks and timing estimation calculates each basic block’s execution time.

Basic block analysis consists of control flow graph generation and target compilation. At this step, a program is first partitioned into basic blocks. Then a CFG is generated to represent the structure of this program and guide the annotation process. Afterwards, a target cross-compiler translates the source program into target binary codes and then timing estimation is applied to analyze and estimate the cycle count of each basic block.

The second part of our approach is timing estimation for each basic block. A target processor model with pipeline architecture, branch prediction policy and cache structure is applied to calculate basic blocks’ execution time.

At the first step of timing estimation, Basic Block Cycle Calculation, it is assumed that no instruction pre-fetches across basic block boundary, or only pipeline dependency effects within the basic block is considered. Next step is Boundary Effect Correction. It considers pipeline effects across basic block boundaries and calculates a correction number for each transition edge between two basic blocks. Another correction comes from Cache Access Adjusting. Mainly, cache access miss introduces extra program execution time because the processor is stalled until the requested data are loaded into cache.

Finally, a TLM computation model is generated by combining static basic block cycle calculation, dynamic boundary effect correction and cache access time adjustment. The result is annotated to the source program and compiled, executed on a host machine to provide accurate estimation results with high simulation speed.

2.2 Basic Block Cycle Calculation

To calculate the cycle count of a basic block, we follow the abstract pipeline model in [2] to track pipeline status. Mainly, the data dependency may cause pipeline stalls and introduce extra cycles while forwarding may reduce cycles.

The idea is illustrated by the example shown in Fig. 4 where tables 4(a) and 4(b) each shows the pipeline status of the corresponding basic block execution. The rows in the table are the instruction indexes. For the basic blocks in Fig. 4(a) and 4(b), each alone with flushed pipeline will consume 10 cycles and 9 cycles respectively for execution. In practice, either ISS, static pipeline analysis, or physical processor can be used to calculate, or measure, the execution time. Of course, if ISS or a tool is used, its accuracy will influence the accuracy of our approach.

Our approach is efficient, because each basic block only needs to be analyzed once. After the cycle count is calculated, we map each basic block with annotated cycle count to the corresponding CFG node (Fig. 5).

2.3 Boundary Effect Correction

In this section, we will focus on boundary effect correction, which is mainly caused by branch prediction uncertainty and pipeline execution between two basic blocks.

A processor may predict branching results and pre-fetch instructions accordingly. Since branches always occur on basic block boundaries, for a successful prediction, the branch will take fewer cycles than that of a flushed pipeline calculation due to the pipeline execution between two basic blocks. Hence, boundary effect occurs when two basic blocks are executed consecutively.

Essentially, each CFG edge represents a potential pair of consecutive execution basic blocks. We annotate a correction factor on each edge by analyzing the cycle saving of successful prediction statically. For each basic block pair connected by an edge, the cycle saving comes from the difference of the cycle count of concatenated block and the sum of the cycle counts of the two blocks.

![Fig. 4: (a) The pipeline status of basic block i alone. (b) The pipeline status of basic block j alone. (c) The pipeline status of concatenated basic blocks i and j.](image-url)
Fig. 4 shows an example of two basic blocks, *i* and *j*. Fig. 4(a) and (b) are the executed pipeline statuses of two basic blocks respectively with initial pipeline flushed. They consume 10 and 9 cycles each. After concatenation, only 15 cycles are needed (Fig. 4(c)). Therefore the correction factor is -4 (i.e. 15-9-10).

Since whether a correction should be done or not depends on the branching result, a branch prediction procedure, prediction(), is embedded into each basic block, as shown in Fig. 5(a). The procedure takes basic block index as its argument. It first checks whether current block is the predicted successor from its preceding block. If right, it returns the correction factor. Before returning, it actually also makes a prediction of next block.

The CFG in the Fig. 5(b) shows all boundary effect correction factors on edges. The number on each edge means the cycle count to be corrected when the edge is the predicted branch. Assume that a branch to D is predicted after C. If the execution path goes through C to D and then definitely continues to E, a total of -5 cycles has to be corrected, since the path matches with the prediction. In contrast, if the path is C to E, then no correction is done, although its correction factor is not zero.

![Fig. 5: (a) Each basic block is annotated with a prediction function; (b) A cycle correction number is stored on each edge.](image)

### 2.4 Cache Access Adjusting

Another source of timing errors is cache access overhead, for cache miss introduces additional cycles. In general, there are two cache types, i.e. I-Cache (instruction cache) and D-Cache (data cache).

#### 2.4.1 I-Cache

To model instruction cache miss overhead, we first analyze statically which I-Cache blocks are used by which basic blocks. Then, a cache access procedure is inserted into each basic block to check whether the active I-Cache blocks of the basic block reside in cache. A record of global cache information is used to identify cache status at run time. The example in Fig. 6 illustrates how I-Cache blocks are mapped to a basic block. Since instruction addresses are generated by the cross-compiler at static time and I-Cache block size is fixed, we can use the address information to identify which I-Cache blocks are used by each basic block. For example, basic block *D* contains three I-Cache blocks, CB1, CB2 and CB3 as shown in Fig. 6. During simulation, if a cache miss occurs, it returns a cycle count for miss penalty which will be added to the block’s execution time. Global cache information is updated after each I-Cache block access.

![Fig. 6: Identify I-Cache blocks in basic block.](image)

#### 2.4.2 D-Cache

In order to precisely calculate D-Cache access latency, the exact address of data access is required. However, the address cannot be known without executing the target assembly code because indirect addressing (i.e. addressing by register content) is commonly used for data access instructions. It hence incurs the limitation to accurately estimate D-Cache access latency for source-level approaches.

Therefore, an average cycle number from statistical analysis is used to model the delay of each data access.

### 2.5 ANNOTATION ALGORITHM

**Algorithm TAB**

1. **Input**: A control flow graph G and an initial vertex *s*
2. **Output**: A time annotated control flow graph
3. **Q**: vertex queue = { *s* }
4. **G**: control flow graph = 0
5. **Begin**
6. while Q is not empty do
7. for each vertex *u* in Q do
8. for all edge(*u,v*) do
9. calculate the BE correction factor
10. if vertex *v* is *un-visited* then
11. calculate the delay time of *v*
12. calculate I-Cache block numbers used in *v*
13. calculate the data access delay of *v* = the number of data reference instructions * average data access cycles
14. insert *v* into Q
15. mark *v* visited
16. end if // vertex *v* is *un-visited*
17. end of for // all edge(*u,v*)
18. end of for // each vertex *u* in Q
19. end of while
20. End

Based on the discussions above, an annotation algorithm TAB (Timing Annotation in Basic blocks) is presented in this section. The algorithm traverses whole program and calculates the execution time of each basic block, the boundary effect correction factors, the I-Cache blocks used in each basic block and D-Cache access overhead.

The input of this algorithm is a control flow graph *G*=(V,E), which presents the structure of source program. The output is a control flow graph annotated with basic block cycle counts,
boundary effect correction factors, and I-Cache blocks used in each basic block.

Basically, the algorithm follows Breadth First Search (BFS) algorithm. It traverses the program from the initial node of the program. During the traversing process, if a node is not visited before, the algorithm calculates the basic block execution time and the I-Cache blocks used in this node (line 11 to 13). The boundary effect correction factors are calculated for each edge in CFG (line 8, 9). Since the algorithm is based on BFS search, the complexity is the same as BFS, i.e. O(\(|E|+|V|\)).

### 3. EXPERIMENTS

The experiments run on a cygwin environment of Windows XP. The host processor is Pentium D 3.4 GHz and equipped with 1GB RAM. The selected target processor is SimpleScalar PIZA [8]. The benchmark test cases are mostly from MiBench [7].

<table>
<thead>
<tr>
<th>Simulation Speed (MIPS)</th>
<th>Original</th>
<th>TAB</th>
<th>Function ISS</th>
<th>CA-ISS</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bubble Sort</td>
<td>1030</td>
<td>1010</td>
<td>15.8</td>
<td>0.88</td>
</tr>
<tr>
<td>JPEG Encoding</td>
<td>1370</td>
<td>1181</td>
<td>15.1</td>
<td>0.86</td>
</tr>
<tr>
<td>dijkstra</td>
<td>693</td>
<td>658</td>
<td>19.0</td>
<td>0.88</td>
</tr>
<tr>
<td>qsort</td>
<td>1443</td>
<td>1381</td>
<td>8.0</td>
<td>0.85</td>
</tr>
<tr>
<td>FFT</td>
<td>413</td>
<td>411</td>
<td>13.3</td>
<td>0.91</td>
</tr>
<tr>
<td>Average speed up (compared to CA-ISS)</td>
<td>1140</td>
<td>1069</td>
<td>16</td>
<td>1</td>
</tr>
</tbody>
</table>

**Table 1. Simulation speed comparison**

Table 1 shows the simulation speed comparison with other simulation approaches. Original is the original source program with no timing annotation. TAB is the annotated program generated by our approach. Functional ISS is for functionally SimpleScalar instruction set simulator and CA-ISS is a cycle accurate instruction set simulator.

The experimental results show that TAB modified programs run at the speed comparable to the original programs. In other words, the overhead introduced by TAB essentially can be ignored. Yet, CA-ISS is three orders slower than TAB.

Fig. 7 shows accuracy comparison. To save space, only JPEG Encoding tested results are showcased, since results from the other four applications are similar.

The result from CA-ISS is used as the golden reference. The experiment shows that TAB has less than 2% error rate. The errors mainly come from the compiler optimization differences caused by marker insertions in implementation to the source program. Additionally, the latency value of each D-Cache access, assumed fixed and derived from the statistical average value collected from each test-bench, also contributes.

For comparison purpose, we also implement the following approaches. Posadas’s approach [4] annotated each program line according to the operators in the line. The cycle counts for operators are profiled before simulation. The result has about 45% errors.

Meyerowitz [5] annotates each program line with a performance trace file generated by an instruction set simulator and it cannot be applied to optimized compiled codes. The results are obviously not preferable since most programs are optimized in final implementation.

Fig. 8 shows the statistics of estimation error rate of different benchmarks using our approach. The error rates of all cases are within 2% from CA-ISS.

![Fig. 8: The error rate statistics of different benchmarks from TAB approach.](image)

### 4. CONCLUSION

We have proposed and implemented a timing annotation approach for accurate and efficient TLM computation model generation. The encouraging experimental results prove that the approach can be beneficial to software development and processor selection during early SoC design stage.

One future work is to modify the annotation approach and devise an appropriate system-level D-Cache model for further accuracy improvement.

### 5. REFERENCES


