A Fast, Accurate and Reliable Estimation for Rapid Design Space Exploration of Superscalar Architecture

Seung Bae Jee, Yeong Geol Kim and Tag Gon Kim

Systems Modeling Simulation Lab.,
Department of Electrical Engineering and Computer Science,
Korea Advanced Institute of Science and Technology (KAIST),
373-1 Kuseong-dong, Yuseong-gu, Daejon 305-701, Republic of Korea

E-mail: {ygkim,sbchi}@smslab.kaist.ac.kr, tkim@ee.kaist.ac.kr
TEL: +82-42-869-5454
FAX: +82-42-869-8054

Keywords
Cycle counts estimation, design space exploration, error bound, sampled-data simulation

ABSTRACT
This paper proposes a very accurate and relatively fast method of estimating cycle-counts of target applications to rapidly find architecture parameters of superscalar processors that satisfy user-provided real-time constraints. Furthermore, by giving a tight upper bound on the estimation error, user can convince himself of the estimation result. The method is based on a classification of benchmark characteristics into two pieces: architecture independent and architecture dependent ones. The speedup comes from the fact that we need only one full-simulation to obtain the architecture independent information instead of conducting simulation whenever parameters change. Moreover, we can reduce the simulation time in obtaining architecture-dependent ones through sampled-data simulation with a little loss of accuracy. Experimental results show 86 times of speedup against full data simulation with 0.8% estimation error on average, if the desired error bound is set to 5%.
When we set desired error bound to 20%, speedup increases to 134 with 2.8% estimation error on average.

I. INTRODUCTION

The demand for high-performance real-time application is increasing due to the growth of computing power and user's desire to new and better applications. One of the representative applications is the multimedia domain. Multimedia now defines a significant portion of the computing market, and this is expected to grow considerably. As a consequence, the processing demands for such applications are rapidly escalating. On the contrary, processor design time window is becoming shorter and requires an efficient design flow. To meet ever tightening requirements of processor development, this paper proposes a very accurate and relatively fast method of estimating cycle-counts of target applications to rapidly find architecture parameters of superscalar processors that satisfy user-provided real-time constraints. There are numerous technological alternatives that can be incorporated into a processor design. These include reservation station design, functional unit duplication, processor branch handling strategies, and instruction fetch, decode, issue width, and retirement policies. However, there are practical shortcomings with these technologies for efficiently culling a huge design space in an early design stage. Benchmarks execute billions of instructions to test the performance of the proposed system. Therefore, simulation time is getting prohibitive.
Researchers have proposed several solutions to reduce simulation time. One of the earliest studies takes a contiguous part of the trace sample [1]. This paper showed that a systematic sampling of contiguous traces could estimate processor performance with a relative error of only 13% and 15 times of speedup against traditional simulation. Recently, statistical simulation was proposed to speedup the simulation process. In statistical simulation proposed in [3], a statistical profile or a set of statistical program characteristics is extracted from a program execution, e.g., the instruction mix, the distribution of the dependency distance between instructions, etc. This statistical profile is then used to generate a synthetic trace that is subsequently

fed into a trace-driven simulator, which will compute the attainable performance for the micro architecture model. Thanks to the statistical nature of the technique, performance characteristics quickly converge to a steady state solution. Therefore, statistical simulation is a useful technique for culling huge design spaces in limited time [4]. However this approach shows large error from 6% to 22%. All of these approaches have relatively large error and they do not provide the error bound on the estimated value.

This paper proposes faster, more accurate method of estimating cycle-counts of high-performance applications, especially loop-intensive ones, which supports rapid design space exploration of superscalar processors. Furthermore, by giving a tight upper bound on the estimation error, user can convince himself of the estimation result. The rest of the paper is organized as follows. Section 2 presents an overview of the proposed approach, while Section 3 and 4 describes the details of the framework. After showing the experimental results in Section 5, we conclude in Section 6.

II. OVERVIEW OF THE PROPOSED METHOD

Before describing the details of the proposed method, let us describe some basic terminologies and assumptions. A program structure is a directed graph whose node and edge represent the basic block and control flow of the program, respectively. A basic block is 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. The proposed method is based on an assumption that the program structure does not change even though the architecture configuration changes. This assumption holds for our framework thanks to the nature of the run-time instruction scheduling feature of superscalar architecture. In other words, there is no need to recompile the benchmark program even when the parameters' values change. Furthermore, the number of each basic block being visited remains constant across parameter value change because the control flow of the program does no depend on the machine configuration. To summarize, the number of execution counts for each block is architecture-independent information. On the contrary, the execution cycle counts of basic block is surely architecture-dependent. For example, cycle counts taken for executing each basic block varies with change of parameters, such as number of functional units, branch prediction strategy, decode/issue width, and so on.

Finally, there is a popular saying that most programs spend ninety percent of their execution time in ten percent of the code [5]. While the actual percentages may vary, it is often the case that a small fraction of a program accounts for most of the running time. Especially, a loop that contains no other loop is called an inner loop. These loops make execution of the same basic block over and over again. Therefore, we need not to repeatedly simulate same basic blocks again and again to obtain the execution cycle counts of basic blocks. In general, it can be obtained with negligible error by sampled data simulation, as our experimental results show. The estimation is done through following three-steps.

- Step 1: Full data simulation to obtain ***exact*** execution counts of basic blocks.
- Step 2: Sample data simulation to ***estimate*** cycle counts of basic blocks.
- Step 3: Estimate ***total execution cycle counts*** from these two information.

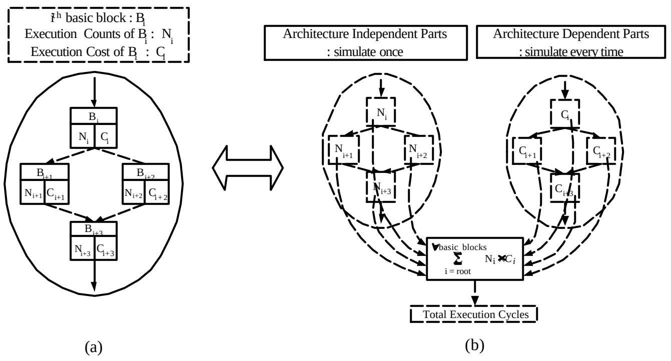

i^th basic block : B_i
Execution Counts of B_i : N_i
Execution Cost of B_i : C_i
Architecture Independent Parts : simulate once
Architecture Dependent Parts : simulate every time
B_i
N_i C_i
B_i+1
N_i+1 C_i+1
B_i+2
N_i+2 C_i+2
B_i+3
N_i+3 C_i+3
N_i
N_i+1
N_i+2
N_i+3
C_i
C_i+1
C_i+2
C_i+3
basic blocks Σ i = root N_i × C_i
Total Execution Cycles
(a)
(b)

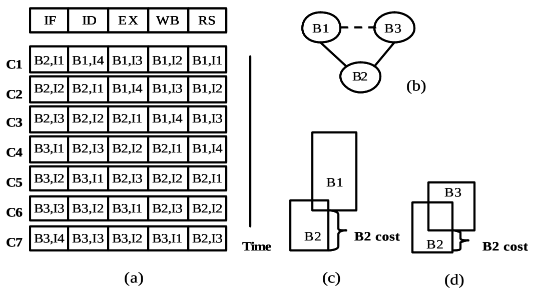
Figure 1. Basic concept of estimation (a) Result of full-data simulation (b) Basic block cost table

Fig. 1 (a) shows execution counts and cost information of basic blocks obtained by full-data simulation. In this figure, BB_i is the basic block identifier, N_i is the basic block execution counts, C_i is the execution cycles of basic blocks. The BB_i and N_i are architecture independent parts. And C_i is architecture dependent part. Fig.1(b) shows the flow of estimation to obtain total execution cycles using sampled-data simulation. The left side of Fig.1 (b) represents architecture independent information which is obtained by full-data simulation just once. And the right side of Fig.1 (b) represents architecture dependent information which is obtained by sampled-data simulation. We reconstruct Fig. 1(a) from (b) with following equation.

$$\text{Total Execution Cycles} = \sum_{i=1}^{\#\text{basic blocks}} C_i \times N_i \qquad (1)$$

When processor configurations vary, the N_i information will be reused. And C_i will be obtained from simulation using sample data which is much smaller than full-data. Therefore, we can reduce simulation time to obtain total execution cycles on various processor configurations. The problem is how we can obtain accurate estimation of total execution cycles with basic block cost obtained through sampled-data simulation.

III. BASIC BLOCK COST ESTIMATION



	IF	ID	EX	WB	RS
C1	B2,I1	B1,I4	B1,I3	B1,I2	B1,I1
C2	B2,I2	B2,I1	B1,I4	B1,I3	B1,I2
C3	B2,I3	B2,I2	B2,I1	B1,I4	B1,I3
C4	B3,I1	B2,I3	B2,I2	B2,I1	B1,I4
C5	B3,I2	B3,I1	B2,I3	B2,I2	B2,I1
C6	B3,I3	B3,I2	B3,I1	B2,I3	B2,I2
C7	B3,I4	B3,I3	B3,I2	B3,I1	B2,I3

Figure 2. Basic block cost (a) Pipeline status when a program is executed (b) Control flow between preceding and current basic block (c) Small basic block cost (d) Large basic block cost

Firstly, we define the cost of basic block before describing the process of total cycle counts estimation. Since superscalar architecture is pipelined, we need to consider overlapping effects of instructions between consecutive basic blocks in defining the basic block cost. Therefore, we define the cost of a basic block as the number of cycles taken *from* the last instruction of a preceding basic block being retired *to* the last instruction of current basic block being retired. To see Fig. 2(a) for example, the first instruction I_1 of basic block B_2 enter into pipeline stage at C_1, while the last instruction I_3 of basic block B_2 is retired at C_7. However, the instructions of the preceding basic block B_1 remain in the stage from C_1 to C_3. Therefore, the execution cycles purely devoted to the basic block B_2 can be thought of as C4 to C_7, which are 3 cycles. Fig. 2 (b)-(d) shows that cost of the same basic block can vary for each preceding basic block. The current basic block B_2 has many predecessors. If predecessor B_1 has large execution cycles and many instructions of B_2 are overlapped into B_1, the execution cycles of B_2 will be small (Fig. 2 (c)). If predecessor B_3 has small execution cycles and a few instructions of B_2 are overlapped into B_1, the execution cycles of B_2 will be larger than previous case (Fig. 2 (d)).

3.1 Cost Classifier

The basic block cost defined above is affected by following factors.

- All execution paths from the root to current basic block in the program structure.
- Number of cache/branch prediction misses for every basic block in the execution path.

However, considering all of these factors for accurate cost calculation becomes impractical due to its large overhead. Through many experiment, we found that following simplified version suffices for the calculation of the basic block cost while maintaining high accuracy.

Predecessor	Current basic block	Miss flag	Execution cycles

(a)

Branch prediction miss	#of L2 data cache read misses	# of L1 data cache read misses	# of Data TLB misses	# of L2 inst. cache misses	# of L1 inst. cache misses	# of inst. TLB misses

(b)

Figure 3. Data structure for basic block cost (a) Cost classifier and cost (b) Miss flag field

- An execution path from the *immediately preceding* basic block to the current basic block.
- Number of cache / branch prediction misses in the *curr ent* basic block.

We define these pair of information as a *cost classifier* in a sense that the cost of the basic block is uniquely determined by them. In other words, a basic block can have different values of cost by the number of distinct cost classifiers. Since the cache and branch prediction miss information affects the basic block cost, we need to conduct full-data sim ulation whenever the cache/branch prediction parameter changes. In spite of this limitation, we can reuse architecture independent information when all other processor configurat ions vary, whose number is much larger as will be shown in our experimental results.

IV. DETAILS OF THE ESTIMATION METHOD

This section describes the details of the proposed cycle-counts estimation methodology. After brief presentation of the data structures used for estimation, actual estimation scheme is explained.

4.1 Data structure for Estimation

Fig. 3(a) shows the cost structure for a basic block consisting of the cost classifier and associated cost. As previously said, the cost classifier is a triple (preceding basic block id, current basic block id, miss flag). The miss flag includes the number of instruction cache/TLB misses, data cache/TLB read misses and branch prediction miss in the current basic block as shown in Fig. 3(b). This miss flag largely affects execution cycles of a basic block, because the cache and branch prediction miss penalty stall the pipeline stages of a processor. For example, the instruction cache miss stall the fetch stage in the pipeline and data cache read miss delay issue of the instruction, and branch prediction miss stall the fetch stage and flush pipeline during penalty cycles. Now, we define a data structure for basic block execution counts, which is filled in after full-data simulation. As shown in Fig. 4(a), there are two fields for every cost classifier.

- # of execution : number of occurrences of (preceding basic block id, current basic block id, miss flag)

- resource usage : number of resource usage to execute all instructions in the current basic block

As will be described later in detail, the resource usage field is useful to calculate worst-case maximum execution cycles of the basic block when there is no matching classifier in the sampled-data simulation result.

Predecessor	Current basic block	Miss flag	Execution counts	Resource usage

(a)

# of long latency floating point computation instructions	# of long latency integer computation instructions	# of floating point computation instructions	# of memory access instructions	# of instruction computation and control instructions

(b)

Figure 4. Data structure for basic block execution counts (a) Overall fields (b) Resource usage field

4.2 Estimation of Total Cycle Counts

Fig. 5 shows the overall flow of estimating total cycle counts with the data structures described previously. While we obtain execution counts table using the execution counts and cost filed after the full-data simulation, the execution cost table is constructed after the sampled-data simulation, as shown in Fig. 5(a) and (b), respectively. Therefore, we can estimate execution cycles of a program from them by joining table to apply Equation (1), as shown in Fig. 5(c).

We need to obtain execution counts table again only when the structures of cache/branch predictor vary. To see Fig. 5(b) and (c) again, we can observe that the costs for BID_2's are not found in the tables. Note that this is hard-to-avoid consequence due to the nature of sampled-data simulation. This kind of situation requires worst-case analysis, which will be described in detail in the following subsection.

4.3 Calculation of estimation-error bounds

We need all execution costs for basic blocks to estimate total execution cycles. However, C_3, C_4, and C5 are left blank at the fourth column in Fig. 5(c). Actually, there can be two kinds of missing in the cost table obtained through sample data simulation – no cost for MF or BID. Fig. 6 (a) and (b) show these two cases, respectively. These two kinds of missing make an error and there is a chance to decrease reliability of estimation. Therefore, we consider execution cost bounds of missing basic blocks, and find the worst-case error from it. Consequently, a designer can obtain total execution cycles of a program within the desired error.

- Bounds of Execution Cycles in the absence of MF

Fig. 6(a) shows the case that there is identical BID but no identical MF (Miss Flag) between execution counts table and cost table. Even in this case, neighboring cost information with the same BID and similar MF can exist in the cost table. Therefore, it is possible to find either maximum execution cycles or minimum execution cycles

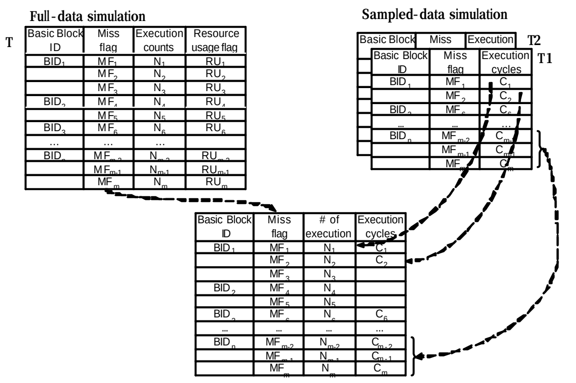
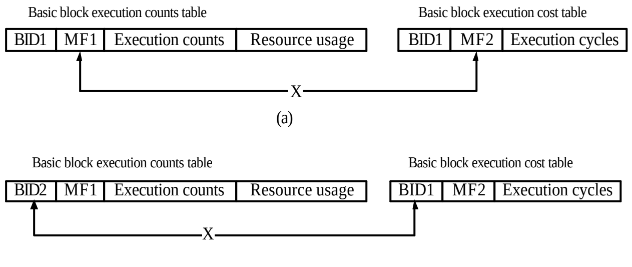


Figure 5. Total cycle counts estimation (a) Execution counts table (b) Execution cost table (C) Joined table for estimation

about the missing cost. For example, Fig.7 (a) shows the case that there are missing MF, and we can find maximum execution cycles in the basic block cost table. Let us assume that the L2 cache miss penalty is 32 cycles and the L1 cache miss penalty is 6 cycles. The MF, 0x00010130, represent that there were one L1 data cache read miss, L2 instruction cache miss and three L1 instruction cache misses in executing this basic block.



Basic block execution counts table
Basic block execution cost table
BID1
MF1
Execution counts
Resource usage
BID1
MF2
Execution cycles
X
(a)
Basic block execution counts table
Basic block execution cost table
BID2
MF1
Execution counts
Resource usage
BID1
MF2
Execution cycles
X

(b)

Figure 6. Two kinds of missing in estimating execution cycles (a) No identical MF in execution cost table (b) No identical BID in execution cost table.

The execution cycles of {BID_1, 0x00010130} is smaller than that of {BID1, 0x00000210} because, when the BID is identical, the execution cycles only depends on cache/branch prediction miss penalty. Therefore, the maximum execution cycles of {BID_1, 0x00000210} will be C_1. The execution cycles of {BID_1, 0x00100310} and {BID_1, 0x0011032} may become the candidates of the maximum execution cycles. However, it is excluded to get the tight bounds of execution cycles. The minimum execution cycles will be 0 because, if the execution cycles of the predecessor block is large and that of the current basic block is small, the last instruction of these two basic blocks may be retired simultaneously. In Fig. 7(b), we can only find minimum execution cycle counts to the contrary. In this case, we use resource usage information to calculate the maximum execution cycles. Therefore, if we know the resource usage information to execute all instructions of the current basic block and the latency of resources, the maximum execution cycles will be the form of weighted sum between resource usage counts and its latency, because all instructions will be executed sequentially in the worst case. In addition, the cache/branch prediction penalty will be added. The minimum execution cycles will be 0, because the last instruction of the current basic block may be retired with the last instruction of the predecessor at the same time.

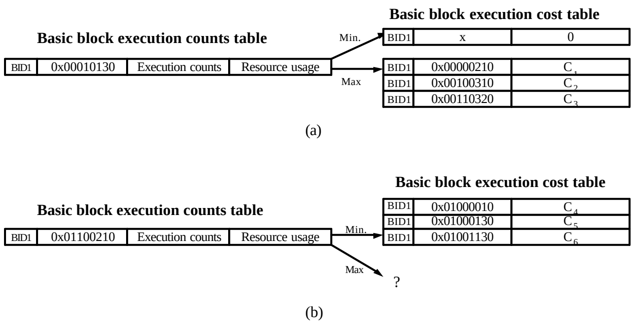


Figure 7. Bounds of execution cycles when there is no MF (a) Maximum execution cycles are found (b) Minimum execution cycles are found.

As a result, all of the execution cost of {BID_i, MF_i}s which are included in the execution counts table will be determined with execution cost. We finally reconstruct the table having all C_i to estimate total execution cycles of a program. We can easily estimate the bounds of the total execution cycle counts from this table as follows.

$$\sum_{i=1}^{m} N_i \times C_i^{\min} \le \text{Total Execution Cycle Counts} \le \sum_{i=1}^{m} N_i \times C_i^{\max} \quad (2)$$

The actual execution cycles of a program exist between the estimated minimum total execution cycles and the estimated maximum total execution cycles. Therefore, the total execution cycles of the estimation result will be

- Total Execution Cycles $_{\text{estimation}}$ =

$$\frac{\text{Total Execution Cycles}_{\min} + \text{Total Execution Cycles}_{\max}}{2} \quad (3)$$

From Eq. (2) and(3), we can obtain the worst-case error as follows.

- Worst-case error(%) =

$$\frac{\text{Total Execution Cycles}_{\text{estimation}} - \text{Total Execution Cycles}_{\min}}{\text{Total Execution Cycles}_{\min}} \times 100 \quad (4)$$

If we measure the actual cycle counts by full-simulation to verify the quality of the proposed method, the measured error becomes the difference between the actual execution cycles and the estimated total execution cycles as follows. It is easy to show that this measured error is always smaller than the worst-case error, hence the latter being the error *bounds*. Therefore, a designer can obtain the estimation result within the desired error by using the worst-case error.

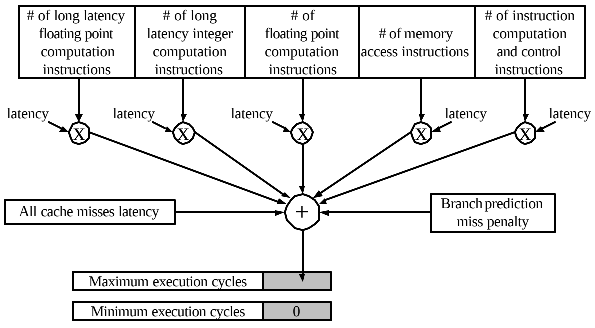

# of long latency floating point computation instructions
# of long latency integer computation instructions
# of floating point computation instructions
# of memory access instructions
# of instruction computation and control instructions
latency
latency
latency
latency
latency
X
All cache misses latency
+
Branch prediction miss penalty
Maximum execution cycles
Minimum execution cycles
0

Figure 8. Calculation of Maximum Execution Cycles with resource usage information

- Measured error (%) =

$$\frac{\text{Total Execution Cycles}_{\text{estimation}} - \text{Total Execution Cycles}_{\text{actual}}}{\text{Total Execution Cycles}_{\text{actual}}} \times 100 \quad (5)$$

4.4 Adaptive Sample Size Determination

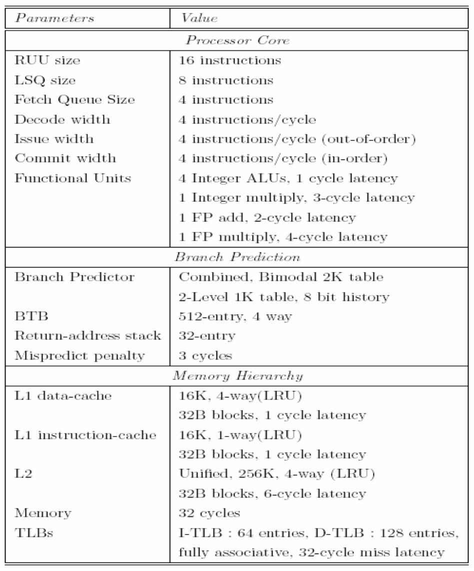
Since sampled-data size is closely related to the speedup and estimation error, sample size determination strategy is important. As the size of the sample gets larger, estimation error is reduced with the sacrifice of speedup. Moreover, the size of sample satisfying the desired error bound is different according to applications. Therefore, we propose an efficient heuristic to determine the sample data size. We define the sample size set as follow.

$$S = \{S_k \mid 0 \prec k \prec N,\ S_{k-1} \prec S_k\}$$

- S_k: size of kth sample data
- N : number of admissible sample data

First of all, designer need to prepare the sample size set which is divided into equal space or user defined space. For example, if there are 512KB input data, a designer can divide it into equal space with 0.1KB. In this case, the number of element becomes 5120 in the sample size set. However, for some application such as EPIC, the sample size must be power of two. In this case, designer needs to prepare only feasible sampled-data set. After this sampled-data set is given, we use binary search to determine best-fit size of sample data through the design space exploration. Note that the purpose of this procedure is trying to make the sample size converge to the best-fit one, as soon as possible, where the speedup is maximized while maintaining the estimation error under the desired error bounds. This can be practically important because even a single target application would require lots of simulations by changing the parameter configuration, in which case the gross speedup over the design space exploration would be largely dependent upon the chosen size of sampled-data.

Table 1. Baseline architecture configurations



Parameters	*Value*
Processor Core	
RUU size	16 instructions
LSQ size	8 instructions
Fetch Queue Size	4 instructions
Decode width	4 instructions/cycle
Issue width	4 instructions/cycle (out-of-order)
Commit width	4 instructions/cycle (in-order)
Functional Units	4 Integer ALUs, 1 cycle latency
	1 Integer multiply, 3-cycle latency
	1 FP add, 2-cycle latency
	1 FP multiply, 4-cycle latency
Branch Prediction	
Branch Predictor	Combined, Bimodal 2K table
	2-Level 1K table, 8 bit history
BTB	512-entry, 4 way
Return-address stack	32-entry
Mispredict penalty	3 cycles
Memory Hierarchy	
L1 data-cache	16K, 4-way(LRU)
	32B blocks, 1 cycle latency
L1 instruction-cache	16K, 1-way(LRU)
	32B blocks, 1 cycle latency
L2	Unified, 256K, 4-way (LRU)
	32B blocks, 6-cycle latency
Memory	32 cycles
TLBs	I-TLB : 64 entries, D-TLB : 128 entries,
	fully associative, 32-cycle miss latency

V. EXPERIMENTAL RESULTS

We use the SimpleScalar simulator environment [2] as a basic tool-set upon which the estimation capability is built. Table 1 shows the baseline architecture configurations, selected from 37 parameters provided by SimpleScalar, to verify the performance of the proposed method. Furthermore, we selected Mediabench [6] as target applications for the experiment because it is most popular and well-organized benchmarks for various media applications, ranging from signal and image processing to cryptography. We estimated total execution cycles of Mediabench applications on various processor configurations using the proposed method. Mediabench includes a set of media applications and associated reference data. However, the size of input data is closely related to simulation time, hence the speedup. A skillful designer would use a part of reference data which is enough to represent the performance of a processor. Generally, the CPI (cycle per instruction) is well-known metric for performance evaluation. When we simulate a program with reference data, the CPI will converge to the steady state value. Therefore, we can obtain the CPI characteristics by simulating the data until the CPI goes into the steady state region. Therefore, we determine the actual input data size as the value at which the CPI just goes into the steady state region with 1% stable margin. For example, the reference data size of the RASTA application is 256KB, and its CPI is approximately 0.862 on the baseline architecture. However, we can obtain the approximate CPI value of this application with only 32KB input data. Table 2 shows the size of original reference data and actual input data according to the applications.

Table 2. Comparison of reference and reduced input data size.

Applications	Reference data (KB)	Input data (KB)
JPEG	100	43
GSM	289	5
G.721	100	22
RASTA	256	32
EPIC	256	256
ADPCM	289	51

Table 3. Comparison of Speedup with the 5% desired error

Applications	Reference data (KB)	Input data (KB)
JPEG	5.43	3.63
GSM	54.26	1.22
G.721	1886.05	434.6
RASTA	28.15	4.09
EPIC	8.16	8.16
ADPCM	212.86	64.88
Average	632.9	86.1

The reason why we reduce the reference data this way is to fairly compare the speedup of the proposed method to others as much as possible. The rest of reference data except for the actual input data will be the redundant part which increases speed up. Table 3 compares the speedup when we use reference data and reduced input data for estimation with the 5% desired error. In the rest of this paper, we will regard the *speedup with reduced input data* as the speedup of this approach. This metric will represent the usefulness of the proposed method with the measured error.

5.1 Performance measurement through design space exploration

From now on, lots of graphs are going to be presented to show the worst-case error and actually measured error for the result of the proposed estimation method when processor configurations vary. Note that the measured error (distance between the true cycle counts and estimated value by the proposed tool) is always less than the worst-case error, i.e., reported error bound. We followed the procedure described in Section V, and gather the estimation results with converged sample size and 5% desired error.

- Variation in the degree of parallelism

There are four parameters in charge of the degree of parallelism(instruction fetch queue size, decode width, issue width, commit width). From the baseline parameter (4, 4, 4, 4), we estimate the execution cycles of two different configurations using sampled-data simulation. Fig. 9(a) and (b) show the estimation results including the worst-case error and the measured error.

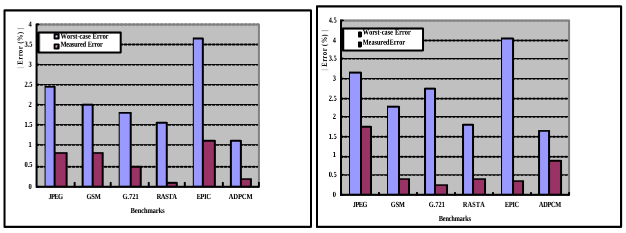


Figure 9. Variation of Instruction fetch size, decode/issue/commit width (a) (2,2,2,2) (b) (16,8,8,8)

- Variation of Reorder Buffer: RUU, LSQ

We consider a pair of RUU and LSQ size for design space exploration, varying from the baseline parameter (16, 8) to (4, 4) and (32, 16). Fig.10 (a) and (b) show the estimation results.

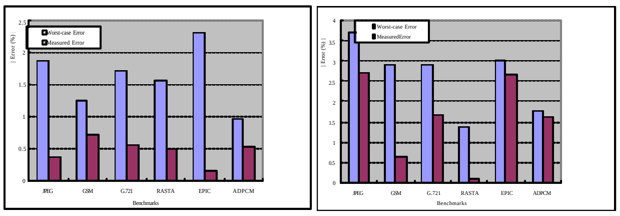


Figure 10. Variation of RUU, LSQ size

- Variation of Memory & FU latency

There are various memory and functional units with different latency.

- Memory latency group
  (Branch prediction penalty, L2 data cache hit latency, L1 data cache hit latency, L2 instruction cache hit latency, TLB hit latency, TLB miss latency, Memory access latency)
- Functional units latency group
  (IALU latency, IMUL latency, FALU latency, FMUL latency)

The memory latency group varies from (3,6,1,6,1,1,30,8) to (7,12,1,12,1,1,25,64). And the functional unit latency group is varied from (1,3,1,4) to (3,6,5,8). Fig. 11 (a) and (b) show the estimation result with errors for each group, respectively.

- Functional Unit Duplication

The number of functional unit is directly related to the execution cycles. However, it is not always proportional to execution cycles. Therefore, a designer should carefully consider the number of functional units, too. We grouped four kinds of functional units as follows.

- (# of Integer ALUs, # of Integer Multipliers, # of Floatin g point ALUs, # of Floating point Multipliers)

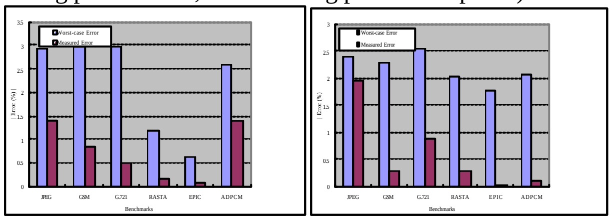


Figure 11. Variation of memory latency and functional unit latency

We estimate the execution cycles from the base parameters (4,1,4,1) to two other parameter group, (1,1,1,1) and (8,2,4,2). Fig.12 shows the estimation results.

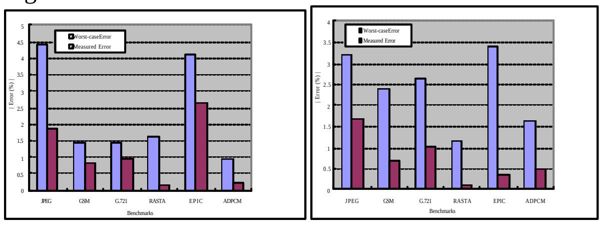


Figure 12. Functional unit duplication

5.2. Speedup and Measured Estimation Error

In previous subsection, we looked at the worst-case and the measured error assuming 5% desired error. If a designer sets the desired error larger, the size of sampled-data will decrease. As a result, we obtain large speed up and the measured error. We summarize all of the estimation results according to the user-defined desired error and show the loop information which is useful to partially guess the complexity of the programs in Table 4. Average measured error represents average of the difference between the total execution cycles and the estimated total execution cycles on the nine configurations, which is actual error by using this estimation method. The speed up and the measured error are dependent on applications, because the complexity of a program structure is different. If the program structure is simple like as ADPCM or G.721, we can obtain high speed up and small measured errors through the proposed estimation method. JPEG and RASTA applications are just the opposite. The measured error is very small in spite of estimation with large desired error. The average speed up is 86 times faster than full data simulation, and average of all measured error is 0.8% with the 5% desired error. When we set the 20% desired error, average speed up is 134 times faster and average of all measured error is 2.79%. All of the estimation results are satisfactory in speedup and measured estimation error compared to previous research works. For comparison, in the related work, the average speed up is 15 times faster and the average error is approximately 13% in trace sampling method. Also, the result of statistical simulation show 13% error, 6% error and 22% error in JPEG, GSM and G.721 applications, respectively. Lastly, note that the speedup in table 4 is the result when the input data size is reduced from the original reference data. To see the table 3 again, we have speedup of 632.9 on average instead of 86.1 for the desired error of 5%.

Table 4. Speed Up and Measured Error

			5% Desired Error		*10% Desired Error*		*20% Desired Error*	
Appli-cations	Execution time ratio of loops (%)	# of loops	Speed Up	Avg. Meas-ured Error (%)	Speed Up	Avg. Meas-ured Error (%)	Speed Up	Avg. Meas-ured Error (%)
JPEG	98.44	278	3.63	1.61	3.78	1.72	4.00	2.14
GSM	96.97	27	1.22	0.63	9.92	1.67	15.04	1.16
G.721	99.78	10	436.6	0.72	505.08	1.50	667.43	2.89
RASTA	91.84	108	4.09	0.29	5.87	0.85	5.98	2.1
EPIC	98.07	183	8.16	0.84	29.49	0.97	34.07	1.9
ADPCM	98.87	1	64.88	0.72	70.77	1.87	77.85	6.55
Average			86.1	0.8	104.15	1.43	134.06	2.79

There can be some argues which data size is practically reasonable. One thing we can tell is that the size of the input data should be determined according to the objectives of the simulation. If one of the design objectives is related to check the system behavior against some constraints on peak values, such as peak power, it would be better not to reduce the size of the data. In this case, the speedup achievable by proposed framework would increase as table 3 implies.

VI. CONCLUSION

This paper proposed a very accurate, significantly fast method of estimating cycle-counts of high-performance applications, especially loop-intensive ones, which supports rapid design space exploration of superscalar processors. Furthermore, by giving a tight upper bound on the estimation error, user can convince himself of the estimation result. In the proposed method, architecture independent information, such as number of execution counts for every basic block, is obtained with a full-data simulation. This information is reused over and over again when the architectural parameters change in the course of design space exploration, during which sampled-data simulation is sufficient to obtain the architecture-dependent information, namely basic block cost. The only exception to this statement is the cache/branch prediction parameters. Although full-data simulation is needed for such parameters change, this is acceptable because there are much more parameters requiring only sampled-data simulation, as shown in Section V. After applying the proposed method to Mediabench applications, the results showed 86 times of speedup against full-data simulation with 0.8% estimation error on average, when the desired error bound was set to 5%. With desired error bound of 20%, speedup increased to 134 with 2.8% estimation error on average. The results were much more promising than those of previous research works such as trace sampling method and statistical method, in terms of speedup, accuracy and reliability. Future works include developing more accurate basic block cost model related to execution path and cache/branch prediction misses characteristics. In addition, the power/area estimation scheme will be integrated to extend the applicability of the proposed framework.

REFERENCE

[1] T.M. Conte, M.A. Hirsch, and K.N. Menezos. Reducing state loss for effective trace sampling of superscalar processors. In *Proceeding of the 1996 International Conference on Computer Design* (ICCD-96), pages 468-477, October, 1996.

[2] T.M. Austin. A User's and Hacker's Guide to the SimpleScalar Architectural Research Tool Set, January, 1997

[3] M. Oskin, F.T. Chong, and M. Farrens. HLS: Combining statistical and symbolic simulation to guide microprocessor design. In *Proceedings of the 27th Annual International Symposium on Computer Architecture* (ISCA-27). pages 71-82, June. 2000.

[4] L. Eeckhout, K.D. Bosschere, Increasing the accuracy of statistical simulation for modeling superscalar processors. Performance, Computing, and Communications, 2001. IEEE International Conference on. , pages 196-204, 2001

[5] J.D. Ullman, Compilers - Principles, Techniques, and Tools, Addison-Wesley, 1986.

[6] Chunho Lee, Miodrag Potkonjak, William H. Mangione-Smith, MediaBench: A Tool for Evaluating and Synthesizing Multimedia and Communications Systems. In *Proceedings of MICRO'97,* pages 330-335, December 1997.

[7] Victor V. Zyuban. Inherently Lower-power High-Performance Superscalar Architectures. PhD paper, Department of Computer and Science Engineering, Notre Dame, Indiana, 2000.

[8] Subbarao Palacharla, Norman P.Jouppi and J.E. Smith. Complexity-Effective Superscalar Processors. In *Proceedings of the 24th Annual International Symposium on Computer Architecture*(ISCA-24). pages 206-218, June. 1997