B^2Sim: A Fast Micro-Architecture Simulator Based on Basic Block Characterization

Wonbok Lee
Department of Electrical Engineering
University of Southern California
Los Angeles CA 90089
(213) 821-4206
wonbokle@usc.edu

Kimish Patel
Department of Electrical Engineering
University of Southern California
Los Angeles CA 90089
(213) 821-4206
kimishpa@usc.edu

Massoud Pedram
Department of Electrical Engineering
University of Southern California
Los Angeles CA 90089
(213)-740-4458
pedram@usc.edu

ABSTRACT
State-of-the-art architectural simulators support cycle accurate pipeline execution of application programs. However, it takes days and weeks to complete the simulation of even a moderate-size program. During the execution of a program, program behavior does not change randomly but changes over time in a predictable/periodic manner. This behavior provides the opportunity to limit the use of a pipeline simulator. More precisely, this paper presents a hybrid simulation engine, named B^2Sim for (cycle-characterized) Basic Block based Simulator, where a fast cache simulator e.g., sim-cache and a slow pipeline simulator e.g., sim-outorder are employed together. B^2Sim reduces the runtime of architectural simulation engines by making use of the instruction behavior within executed basic blocks. We have integrated B^2Sim into SimpleScalar and have achieved on average a factor of 3.3 times speedup on the SPEC2000 benchmark and Media-bench programs compared to conventional pipeline simulator while maintaining the accuracy of the simulation results with less than 1% CPI error on average.

Categories and Subject Descriptors
C.4 [**Performance of Systems**]: Measurement techniques, Modeling techniques B.8.2 [**Performance and Reliability**]: Performance Analysis and Design Aids

General Terms
Measurement, Performance, Design

Keywords
Micro-architecture simulation, Program behavior, Basic block

1. INTRODUCTION
Every program shows behavior changes during its execution time. In general, program behaviors are represented in terms of cycle per instructions (CPI), I/D-cache miss count, number of branches, power, etc. Over an execution of the program, this behavior changes appear irregular but they often possess certain repetitive patterns. For example, general MPEG player programs repeatedly read, decode, and dither the image in each frame of a video stream. In code level, this phenomenon is highly expected since same piece of source code, i.e. functions, libraries, system calls, are being executed again and again in a recurring manner.

CODES+ISSS'06, October 22-25, 2006, Seoul, Korea.

There are a number of architectural simulators that emulate the real program's behaviors and basically they have been developed in two directions: the first direction is to provide simple simulators that can trade-off the amount of information they generate versus the amount of time they spend for the simulation. In this case, the architectural simulator provides 'just enough' information about the micro-architecture that developers need but takes short simulation times. The second direction is to provide all the detailed pipeline simulation information that micro-architecture developer needs by sacrificing the simulation speed. Both directions are important in the sense that they capture either a specific or a general system's behavior, respectively.

Especially for the detailed pipeline simulators, their simulation time becomes order of five to six times slower compared to the program's actual execution time in a real system [1]. For this reason, micro-architecture researchers have tried either to develop a faster simulator or to find methods to obtain the simulation results from the existing simulators in a short time.

Program phase detection [2][3][4] has proven to be an effective method of rapidly obtaining the simulation results (albeit with some accuracy losses.) By nature, however, this phase detection approach has three obstacles to overcome: difficulty of picking the correct granularity for phase detection, "hasty" phase determination during the program initialization, and need for correctly and uniquely determining the program phase. The first problem is under what interval the phase is to be detected. Depending on this granularity, simulation results can be interpreted in a different way. The second problem hinders us from detecting the correct phase since it generates misleading information when we profile/detect phases in an early stage of a program execution. During the initial execution time in most of the programs, the behavior changes in a quite irregular/unexpected manner with mostly no cyclic/repeated behavior observed. The third problem is the most costly one since we need to run pipeline simulation from a program start to the end in order to both check the phase similarity of each interval and determine the uniqueness of each phase.

In this paper, we propose a novel hybrid micro-architecture simulator that analyzes the code behavior in a fine granularity, extracts some architectural metrics, and calculates the overall distribution of CPI in a program based on a memory access behavior in a basic block, which leads to a dramatic reduction of simulation time and ultimately solve the above three obstacles.

2. PRIOR WORK
Due to innate slow nature of the pipeline micro-architecture simulators, lots of research effort has been concentrated in making these simulators faster. Previous acceleration approaches in micro-

architecture simulation can be divided mainly in two categories: sampling/phase based approaches and instruction set compiled simulation based approaches. Sampling/phase based approaches are based on selecting intervals that are representatives of the program behavior, executing detailed pipeline simulation only for those intervals, resulting in high speedup. On the other hand, instruction set compiled simulation based approaches translates the executable binary from target machine's instructions to the host machine's instruction either by static or dynamic compilation and achieve high speedup.

Many of the recently published papers in this domain are based on sampling/phase based approaches. In [2][3][4], Sherwood et al, suggest the use of repetitious manner of a program as 'phases' with the SPEC95 benchmark. Basically, they found two folds in a program behavior: 1) it converges to a steady state 2) it has a cyclic behavior. Based on those observations, they proposed a phase detection technique to find all the distinct phases of a program, and use it to reduce simulation time by running the pipeline simulation only once for each of those distinct phases which are representatives of the whole program behavior. Later, Hamerly et al. utilize this phase idea and made a simulator called *SimPoint* [5], which performs off-line phase classification. In [1], Roland et al. propose a sampling based micro-architecture simulation framework to enable fast and accurate performance measurements. They accelerate simulation by selecting minimal subset of instruction stream from a total instruction stream and developed a simulator called SMARTS. In [6], Dhodapkar et al. detect a program phase with the instruction working set. They also make a comparative study in three phase detection techniques: instruction working set based, phase based, and conditional branch counts based. Their experimental results show that phase based technique achieve better performance than the other techniques.

The other category of simulation speed acceleration techniques include the use of compiled instruction set simulators [7][8][9]. Compiled instruction set simulators are fast since they remove the expensive decoding stage of the interpretive simulators like, *Simplescalar* [11] at the expense of flexibility and accuracy. In [7], authors try to combine the benefits of the traditional interpretive simulator, namely the flexibility, with the speed of compiled instruction set simulator by re-decoding the instructions dynamically in case of previously decoded instructions have changed. In [8], authors propose an idea of not interpreting the instruction but translating it to the host machine's only if it is more profitable (in terms of time), and make use of 'Just in Time' compiler to do dynamic translation of multiple basic blocks in order to exploit instruction level parallelism (ILP) offered by the host processor.

In [10], the authors accelerate the simulation for the instruction cache. They build an abstract model of each piece of program code, use an approximate model of the instruction cache, and propose a conflict-based simulation time reduction technique for instruction cache under a multi-tasking paradigm. This approach is applicable solely to instruction cache simulator.

Existing 'simulation time reduction' approaches for architectural simulation are mainly based on either phase identification or statistical sampling. Especially for the former approaches, they need functional simulation level pre-processing to profile phase information, determine the phase similarities, and fulfill the pipeline simulation for the distinct phases that they found ahead.

The contribution and the distinction of the paper are 5 folds.

- **Efficiency**: B^2Sim uses pipeline simulator minimally i.e., every basic block will be executed in pipeline simulator once and redundant execution is avoided.
- **Accuracy**: B^2Sim is deterministic and fully acknowledges the unpredictable program behaviors during the initial program execution time and accurately computes CPI.
- **No need for static profiling:** B^2Sim does not have to identify the region of unique behaviors (phases) with a run of functional simulation. Neither any clustering method nor any profiling is necessary for the phase similarity check. [5]
- **No need for fast-forwarding or warming**: B^2Sim does away with multiple fast-forwarding/warming steps, which have been inevitable in previous simulation acceleration techniques. [1]
- **Granularity**: B^2Sim has high accuracy and detects phases in a flexible fine-granularity manner because the granularity of B^2Sim is that of a basic block.

The remainder of the paper is organized as follows: In section 3, we specify the motivational observations of this paper. Section 4 covers the theoretical backgrounds and section 5 will show the simulator framework. In section 6, we show the simulation environment and section 7 shows the experimental results. Lastly, section 8 is the conclusions.

Table 1. Micro-Architecture Parameters

Main Memory Latency	32 cycles
L1 I/D Cache	32KB 32-way 32Byte block 1 cycle hit latency
I/D-TLB	4-way 1024 entries 32 cycles miss latency
Branch Predictor	Bimodal 128 Table
Functional Units	1 INT ALU, 1 INT MULT/DIV 1 FP ALU, 1 FP MULT/DIV
RUU/LSQ size	8/8
Instruction Fetch Queue	8
In order Issue	True
Wrong Path Execution	True

3. MOTIVATIONAL OBSERVATTIONS

Once compiled, the structure of a program remains the same and the static scheduling of instructions made by the compiler does not change at run time. The same is true for input/output data dependencies that exist between different instructions. The on-chip behavior of a block of code is thus defined by the data dependencies that exist between different instructions of the block. Hence, every time a block of code, for example, a basic block, is executed, the number of on-chip cycles it takes remains the same. However, the same cannot be said about the number of CPU cycles due to off-chip accesses caused in the basic block.

During a program's execution, many blocks of the code are visited multiple times. As shown in Figure 1, for example, block #k is visited twice in the execution timeline. In both block invocations, the structure of the code and existing data dependencies among the instructions in the block do not change.

Hence, the on-chip behavior of this block remains the same. However, this does not guarantee that the number of cycles taken by this block, at times T1 and T2 are the same. This is due to the memory related instructions, i.e., loads and stores, inside the block whose behavior changes over the execution and the off-chip behavior depends on cache and/or TLB configuration and status. As a result, the total cycle count taken by the block may vary. This is the core observation of this paper. We separate out the on-chip and off-chip behaviors of a block and utilize the consistency of on-chip behavior of a block over the execution.

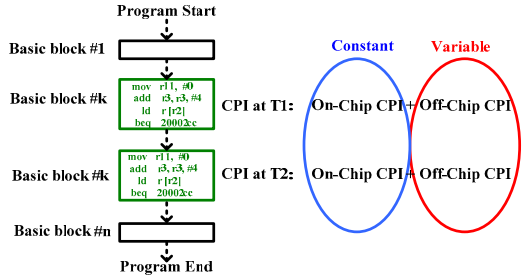

Figure 1. Motivational Observations

Table 2 shows the on-chip CPI variance of some of the most frequently visited basic blocks for a number of benchmark programs. (The benchmark programs will be explained in section 4). We choose two frequently visited basic blocks in each programs, calculate their average on-chip CPI, and further calculate the variance of on-chip CPI for the first 100 invocations of each of these basic blocks. As shown, the CPI variances are quite small, i.e. on-chip CPI within each basic block remains nearly constant over visit times. Actually, on-chip CPI variance can be zero, but very small error in CPI variance comes from the '*first visit phenomenon*' explained in section 4.2. Moreover, if we calculate the variance for even larger visit count of each basic block, then the on-chip CPI variance would go to zero.

Table 2. On-chip CPI Variation of Some Frequently Visited Basic Blocks for its First 100 Visits

Program	BB Size (inst #)	Number of visit	Avg. on-chip CPI	On-chip CPI variance in first 100 visits
gzip-g	21	41.9M	1.333	2.0e-4
	45	12.0M	1.356	5.3E-6
gcc-e	12	77.1M	1.667	0
	37	27.2M	1.215	2.9e-5
bzip-p	44	35.5M	1.408	2.3e-4
	69	20.8M	1.231	1.9e-5
mcf	16	93.2M	1.624	3.9e-5
	137	16.5M	1.400	6.4e-5
vortex	20	44.6M	1.596	7.9e-4
	38	26.3M	1.581	1.3e-3
djpeg	34	21.6M	1.384	2.3e-4
	181	2.7M	1.143	1.5e-5
cjpeg	37	21.6M	1.403	2.6e-4
	154	3.5M	1.117	0

4. THEORETICAL BACKGROUND

During a program execution, program behavior (which we call a phase) may vary over time. Generally, these behavior changes are represented by micro-architecture metrics such as the I/D-cache miss count, on/off-chip stall counts, cycle per instruction (CPI), etc. In this paper, we verify the validity of B^2Sim with CPI errors. Table 1 shows the micro-architecture parameters that we used for our simulation. Most of the parameter settings are obtained from [12], except the 'in order issue' parameter.

4.1 Basic Block Classification

The basic block (BB) is defined as a set of instructions which has a single entry point and a single exit point, between which there is no incoming/outgoing control-related instructions. At the source code level, a BB contains instructions starting from a branch's target instruction up to the next branch instruction in a sequential code flow. A BB is a suitable entity to capture the program phase within which the source code flow is always the same on every visit, consequently, there is a high chance to repeat its architectural behavior inside.

Conceptually speaking, a BB can be classified into two groups: static/spatial basic blocks (SBB) and temporal basic blocks (TBB). SBB is a unit within which the overall CPI remains relatively constant. It has either no memory-related instructions or has memory-related instructions but no cache miss occurs in every visit. On the contrary, TBB is a unit within which the overall CPI varies by a large amount on every visit to the BB due to the memory-related instructions and its dependences on the cache/TLB status. For either type of the BB, we can decompose the overall CPI in two components: on-chip CPI and off-chip CPI. On-chip CPI comes from instructions which cause on-chip latencies due to register-register transfers, arithmetic-logic operations, data and control hazards, pipeline stalls, branch miss prediction, etc. Off-chip CPI comes from instructions which cause off-chip latencies due to main memory accesses such as I/D-cache miss, I/D-TLB miss, etc. The next two subsections present the details about how the on-chip and off-chip cycles of a BB are characterized and computed, respectively.

4.2 On-chip CPI Characterization

To determine the number of on-chip cycles a BB takes, a BB must be executed in a pipeline simulator e.g., *sim-outorder*. This on-chip cycle characterization is performed when the BB is visited/executed for the first time. Since the total number of cycles taken by the BB is comprised of on-chip and off-chip cycles, and the off-chip cycles are those cycles during which the whole pipeline is stalled i.e., the processor is waiting for the memory operation to finish, the on-chip cycle for a BB is obtained by subtracting the cycles for which the whole pipeline is stalled from the total number of cycles taken by the BB.

However, it does not imply that the on-chip cycles for a BB can be characterized during any of its invocations by running it in a pipeline simulator. Consider a BB that is visited for the first time and has some instruction cache misses (since the instruction cache is initially cold, the chances for such occurrences are high). We may not correctly calculate the on-chip CPI for this BB on its first visit since the instruction cache miss on the following instruction may hide the data dependencies that exist between the missed instruction and the previous instructions on which it is dependent. We call this as a '*first visit phenomenon*' and this phenomenon prevails in the early time of a program execution.

Figure 2 shows an example to illustrate this situation. In the center of the figure, a BB being executed is shown. Assume that all

instructions in the BB take a single cycle and the 5th instruction (sub r2, r3, r1) is dependent on the 4th one. If there is no instruction cache miss, then the 5th instruction will have to wait for the 4th one to finish. Now consider that the 5th instruction misses in the cache (shown on the left side of Figure 2). In this case, the fetch is delayed by the memory latency and this buys enough time for the 4th instruction to finish. When the 5th instruction is on the execution stage, the data it needs is ready. Hence, we may obtain two different 'on-chip' cycle counts for the same BB under these two cases.

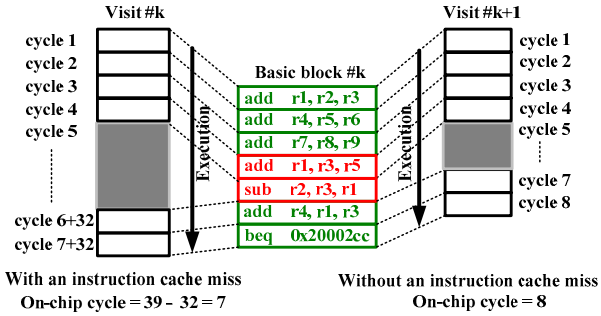

Figure 2. Example of On-chip CPI Characterization w/ and w/o an Instruction Cache Miss in a Basic Block

To avoid this misleading situation, we have executed a pipeline simulation for this type of BBs and give another chance to characterize. When there is no instruction cache miss on the following visit to BB (shown on the right side of Figure 2), we list this BB to the on-chip CPI information table. A threshold value of three for the number of times a BB is executed in the pipeline simulator is used before we decide to on-chip cycle of the BB to the table. If a BB cannot be characterized within 3 visits, the on-chip cycle extracted in the last attempt is used.

Once recorded on the table, the on-chip CPI information of each BB is never updated. Table 3 shows the information that is recorded for each of BBs in the on-chip CPI information table. The most important item is the number of cycles since the cache simulator does not know the on-chip latencies which can only be measured in the pipeline simulator.

Table 3. Items in the "On-chip CPI Information Table"

BB #	No. of Visit	No. of cycles	On-chip CPI

4.3 Off-Chip CPI Calculation

To get the off-chip cycles of a BB takes, a cache simulator e.g., *sim-cache* is used. Since the cache simulator can give the information of number of cache misses of a BB, this number can be simply used in conjunction with memory latency to derive the off-chip cycles taken by the BB. However, because the cache miss information can include the data cache miss, the information is not enough to generate the correct 'off-chip' cycles. This is due to the fact that not every data cache miss generates pipeline stall cycles equal to the memory latency specified for the architecture.

In Figure 3, for example, the 3rd instruction (ld r1 0(r5)) loads the data in the register used by the 5th instruction (sub r2, r3, r1). Assume that the compiler has scheduled the 4th instruction (add r7, r8, r9) between two dependent instructions, 3rd and 5th. When this load instruction misses in the data cache, it does not stall the pipeline immediately since the following 4th instruction can still go on to finish, and only after that it stalls the pipeline since a dependent instruction is encountered. Due to this phenomenon, the actual pipeline stall cycle count in off-chip access is less than the actual memory latency.

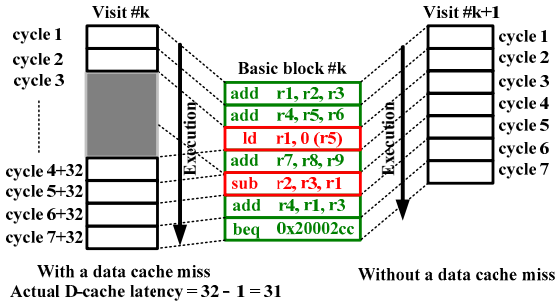

Figure 3. Example of Off-chip CPI Calculation w/ and w/o a Data Cache Miss in a Basic Block

To circumvent this situation, we have used an approximation technique; During a BB characterization, if the current instruction is dependent on the previous load instruction and the dependencies are not satisfied yet then we will find the distance between these two instructions in terms of the number of instructions in between and use the distance as an approximate value of cycles that hide the actual memory latency. Hence the actual off-chip cycle counts are obtained by subtracting this distance from the memory latency. For example in Figure 3, this "latency-hiding" distance is 1 since there is only one instruction between two dependent instructions.

For this situation, two factors should be considered: 1) The aforementioned calculation can only be done during the BB characterization step since the dependency check is made only in the pipeline simulator. 2) It may not always be possible to find this distance. If the load instruction does not result in a data cache miss during the characterization, then the dependency will be satisfied, therefore we will not be able to observe the dependency.

Apart from this specific example, there can be multiple load instructions and multiple dependencies in a BB. When the same BB is executed in the cache simulator without knowing which load instructions end up with data cache misses, it is difficult to determine how many cycles the pipeline will be stalled due to the off-chip accesses. For this reason, we use the average distance to find the off-chip cycles when data cache miss occurs while the BB is being executed in the cache simulator.

5. SIMULATOR FRAMEWORK

In general, an application's simulation time between the simple functional simulator (*sim-fast*) and the detailed pipeline simulator (*sim-outorder*) is one to two orders of magnitude different. [15]. This large performance difference results from two reasons: the amount of information that each simulators generates and the innate expensive nature of pipeline simulation. For the first reason, as an example, *sim-fast* generates number of instruction, *sim-cache* additionally generates number of cache access/hit/miss and number of TLB access/hit/miss, and *sim-outorder* generates all the necessary micro-architectural information by the simulation in the pipeline. For the second reason, executing all the way through the pipeline stages, i.e., fetch/dispatch/execute/write-back/commit, needs huge amount of time for calculations, status updates, dependency checks, branch prediction, speculative/wrong path execution, etc.

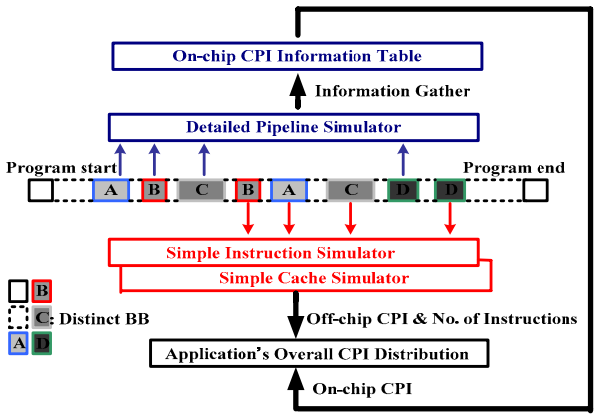

Figure 4. B²Sim Flow Diagram

We thus present a framework for hybrid simulation engine, called B²Sim. Figure 4 depicts the flow diagram of B²Sim. In this figure, each of the small shaded boxes (A, B, C and D) represents a distinct BB within a program run and there are two types of micro-architecture simulators: a pipeline simulator (*sim-outorder*), and a simple cache simulator (*sim-cache*). When the simulation of an application program starts on B²Sim, each BB runs under the pipeline simulator. If the BB is encountered for the first time, on-chip CPI characterization is performed as explained in section 4.2. If the same BB is encountered again at a later time and its on-chip CPI is characterized then simple cache simulator will be utilized. B²Sim executes the application program and updates the total simulation cycles based on the on-chip CPI from the table and off-chip cycle calculation based on cache miss data for that BB. Any time a new BB comes in, B²Sim switches to the pipeline execution mode, calculates and inserts the on-chip CPI for the new BB, if characterized. This simulator mode switching continues until the end of the simulation.

Table 4. Benchmark Programs and Their Input Files

Program	Reference input file	Actual execution time (sec)	Number of instructions (billion)
gzip	input.log 4	2.9	4.4
	input.source 4	5.3	8.8
	input.random 4	5.8	7.8
	input.graphic 4	6.3	9.4
	input.program 4	8.9	16.8
gcc	expr.i	11.9	15.1
	integrate.i	14.2	16.5
	166.i	62.3	57.0
bzip	input.graphic 4	15.7	24.5
	input.program 4	12.4	20.1
	input.source 4	10.1	16.7
mcf	inp.in in train	55.8	20.1
vortex	lendian.raw in train	16.6	13.0
djpeg	custom.jpg	1.7	2.9
cjpeg	custom.gif	3.23	5.3

Once a BB is characterized in terms of on-chip cycles it takes, the subsequent visits to this BB are run under cache simulator. However, during a subsequent visit to the BB, when we finish executing it and hence we encounter a branch instruction at the end of a BB we switch back to the pipeline simulator. This has to be done for two reasons. 1) We need to do branch prediction update since cache simulator does not support branch prediction update and we need to update the Branch Target Buffer (BTB) for consistent branch prediction 2) The pipeline simulator needs to do wrong path/speculative execution in order to get the correct CPI, because speculative execution is a part of the program behavior.

6. SIMULATION ENVIRONMENT

For the experiments of B²Sim, SPEC2000 benchmarks under different input files [13] are used. Details about application programs and the respective input files are reported on Table 4**.** Moreover, two programs from Media-bench [14] are used to complement the computationally intensive nature of SPEC2000 benchmarks. In each experiment, programs are executed from start to end. (Herein, each set of benchmark program and its input file will be specified in an abbreviated manner e.g., gzip with input.log 4 is denoted by gzip-l.)

For the host, three linux machines are used. They are AMD Athlon 2500+ with 1GB memory, Intel Pentium 4 2.5G with 1GB memory, and Intel Pentium 4 1.8G with 2GB memory. Each benchmark program is executed with one of the aforesaid host machines and the measurements of a program behavior under different input files are consistently done on the same host.

Table 5. Simulation Time Comparisons

Program	Simulation Time (minutes)			Speedup
	sim-cache	*sim-outorder*	B²Sim	
gzip-l	28	160	50	3.20
gzip-s	55	334	107	3.12
gzip-r	51	295	99	2.98
gzip-g	60	365	119	3.06
gzip-p	105	646	213	3.03
gcc-e	62	417	132	3.16
gcc-i	66	444	135	3.29
gcc-166	233	1505	452	3.33
bzip-g	129	696	191	3.64
bzip-p	107	590	162	3.62
bzip-s	90	489	135	3.54
mcf	116	739	223	3.31
vortex	60	404	117	3.45
djpeg	17	94	26	3.61
cjpeg	32	187	57	3.28

7. EXPERIMENTAL RESULTS

We measure the performance of B²Sim and compare the results in terms of the simulation time speedup and the average CPI error with respect to the pipeline simulator. [11]. Table 5 and Figure 5 show the experimental results.

In Table 5, we show the overall simulation time for seven benchmark programs under cache simulator, pipeline simulator, and B²Sim, respectively. The corresponding simulation time speedup between B²Sim and pipeline simulator is also shown. The simulation time of cache simulator is shown as a reference. Based on our experiments, the simulation time speedup of B²Sim over detailed pipeline simulator ranges from a factor 2.98 to 3.64 while the average simulation time speedup is a factor of 3.31.

In Figure 5, we show the CPI error obtained in between B²Sim and the pipeline simulator. The CPI error ranges from

0.05% to 1.25% while the average CPI error is just 0.57%. Note that the new simulator not only generates the CPI information but also generates other results including I/D-cache access/hit/misses, I/D TLB access/hit/misses, etc. As explained in section 4.2, BBs are characterized when there are no instruction cache misses, but there may be instruction cache misses on some consecutive visits to the same BB. Hence, the on-chip cycles accounted for this BB may or may not be larger than the actual on-chip cycles. This introduces some errors in CPI calculation. However, since instruction cache misses are quite low, the error introduced is very small. Note also that some CPI errors come from inter-basic block data dependency which is hard to characterize.

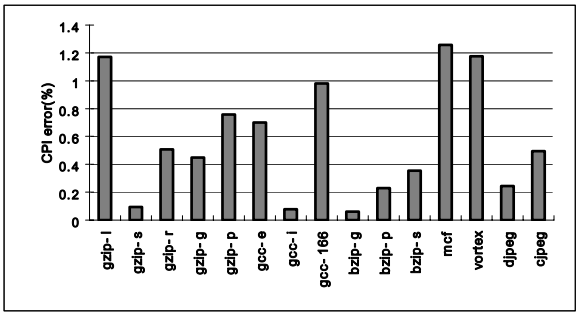

Figure 5. Overall Average CPI Error

Figure 6 shows the relationship between the speedup that we achieved and the instruction per branch (IPB) which is extracted from the simulation output log file. The points in the dotted line denote the speedup factor whereas the points in the solid line denote the IPB. Basically two types of code structural factors affect the speedup of B^{2}Sim: 1) size of the BBs that program visit 2) occurrence frequency of BBs that the programs execute. Since the IPB has a combined nature of these two factors, the value changes of IPB closely correlated with the speedup that we achieved in B^{2}Sim. The minor variation between the two plots comes from the non linear relationship between speedup and IPB.

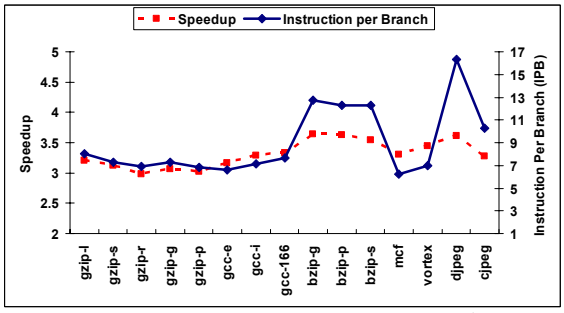

Figure 6. Comparison Between the Speedup of B^{2}Sim and the Instructions Per Branch (IPB)

Clearly, the maximum speedup that B^{2}Sim can achieve is the speed of cache simulator which is about 6X based on Table 5**.** However, since every BB is run at least once in a pipeline simulator for the on-chip CPI characterization and due to branch prediction, some difference exists between the *sim-cache* speedup and the actual speedup of B^{2}Sim.

8. CONCLUSIONS

In this paper, we proposed a novel simulator called B^{2}Sim, which dramatically reduces the simulation time of a program within a micro-architecture simulator using the ideas of: 1) Separating the on-chip and off-chip cycles within a basic block. 2) Making use of pipeline simulator for the on-chip CPI characterization in each basic block and saving the on-chip CPI value in a lookup table to avoid the redundant use of pipeline simulator for the same basic block. 3) Making use of a fast cache simulator for the off-chip CPI calculation of all the basic blocks.

Our experimental results show that we speedup simulation time by 3.31 on average compared to the detailed pipeline simulator with an average CPI error of less than 1%. Compared to previous popular approaches [1][5], speedup that B^{2}Sim has achieved is not an order of magnitude reduction, however, it is highly accurate, deterministic, and has finer level of granularity.

REFERENCES

[1] R. E. Wunderlich, T. F. Wenisch, B. Falsafi, J. C. Hoe, "SMARTS: Accelerating Micro-architecture Simulation via Rigorous Statistical Sampling," In *Proceedings of Int'l Symposium on Computer Architecture (ISCA)*, Jun. 2003.

[2] Tim. Sherwood, B. Calder, "Time Varying Behavior of Programs," *UCSD Technical Reports*, Aug. 1999.

[3] Tim. Sherwood, E. Perelman, B. Calder, "Basic Block Distribution Analysis to Find Periodic Behaviors and Simulation Points in Applications," In *Proceedings of Int'l Conference on Parallel Micro-architectures and Compilation Techniques (PACT)*, Sep. 2001.

[4] Tim. Sherwood, Suleyman Sair, B. Calder., "Phase Tracking and Prediction," In *Proceedings of Int'l Symposium on Computer Architecture (ISCA)*, Jun. 2003.

[5] G. Hamerly, E. Perelman, J. Lau, B. Calder, "Simpoint 3.0: Faster and More Flexible Program Analysis," *Journal of Instruction Level Parallelism 7 (JILP)*, 2005.

[6] A. S. Dhodapkar, J. E. Smith, "Comparing Program Phase Detection Techniques," In *Proceedings of Int'l Symposium on Micro-architecture (MICRO)*, Dec. 2003.

[7] M. Reshadi, P. Mishra, Nikil Dutt, "Instruction Set Compiled Simulation: A Technique for Fast and Flexible Instruction Set Simulation," In *Proceeding of Design Automation Conference (DAC)*, Jun. 2003.

[8] W. Mong, J. Zhu, "DynamoSim: A Trace-based Dynamically Compiled Instruction Set Simulator," In *Proceedings of Int'l Conference on Computer Aided Design (ICCAD)*, Oct. 2004.

[9] A. Nohl, G. Braun, O. Schliebusch, R. Leupers, H. Meyer, A. Hoffman, "A Universal Technique for Fast and Flexible Instruction-Set Architecture Simulation," In *Proceedings of Design Automation Conference (DAC)*, June, 2002.

[10] M. Lajolo, L. Lavagno, A. S. Vincentelli, "Fast Instruction Cache Simulation Strategies in a Hardware/Software Co-design Environment," In *Proceedings of Asia and South Pacific Design Automation Conference (ASP-DAC)*, 1999.

[11] SimpleScalar LLC: http://www.simplescalar.com/

[12] T. Meyerowitz, A. S. Vincentelli, "Modeling Micro-architectural Performance using Metropolis: Memory System Modeling," *SRC Report*, Feb. 2003.

[13] Standard Performance Evaluation Corporation (SPEC) http://www.spec.org

[14] C. Lee, M. Potkonjak, W. H. Mangione-Smith, "MediaBench: A Tool for Evaluating and Synthesizing Multimedia and Communications Systems," In *Proceedings of Int'l Symposium on Micro-architecture (MICRO)*, 1997.

[15] Simple tutorial version 4 at: http://www.simplescalar.com/