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

October, 23, 2006

Wonbok Lee, Kimish Patel, Massoud Pedram

Dept. of Electrical Engineering
University of Southern California

Contents

Introduction

- **Micro-architectural simulators:**
  - A software infrastructure that mimics micro-processor behaviors
- **Where to use:**
  - Performance projection during the pre-silicon phase of the chip design (HW)
  - Performance evaluation / architectural exploration (SW)
  - Hardware-software co-design/verification (HW + SW)
- **Strong points in micro-architectural simulators:**
  - Validate hardware before/without the actual implementation (cost)
  - Easy to explore micro-architectural design space (time)
- **Weak points in micro-architectural simulators:**
  - Slower than the native program execution (order of 4X ~ 5X)
  - Cannot properly handle system call & operating system related parts

- **Why micro-architectural simulators are slow?**
  - Pipeline simulation in software
  - Pipeline stages which are implemented with inter-stage queue management
- **Can we make it faster?**

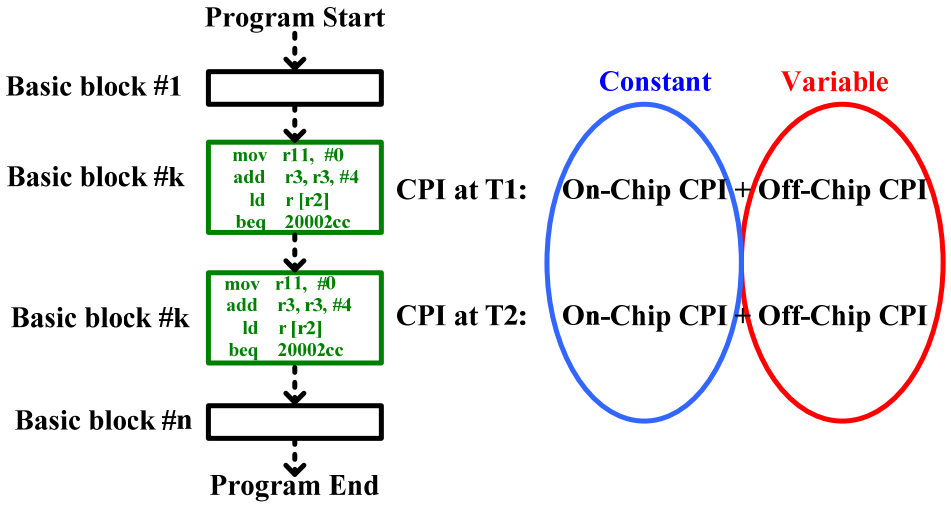
Prior Work for Reducing Simulation Time

- Simulation with Reduced Benchmark
  - MinneSPEC – A. J. KleinOsowski et al., Computer Architecture Letter, 2002.
  - SPEClite – R. Todi et al., WWC, 2001.
- (Periodic) Sampled Simulation
  - SMARTS – R. E. Wunderlich et al., ISCA, 2003.
- Program Behavior (Phase) Based Sampled Simulation
  - SimPoint – T. Sherwood et al., ISCA, 2003.
- Statistical Simulation
  - HLS++ (Hybrid Laboratory Simulator) – M. Oskin et al., ISCA, 2000.
- Accelerate the Warm-up
  - MRRL (Memory Reference Reuse Latency) – K. Skadron et al., ISPASS, 2003.
- Accelerate the Fast-forwarding
  - SimSnap – P. K. Szwed et al., Interact, 2004.
- Truncated Simulation
  - Joshua J. Yi et al., IEEE Trans. on Computer, 2006.
- Instruction Set compiled Simulator
  - ISS - M. Reshadi et al., DAC, 2003.

A Key Observation

- Once a program is compiled, program code structure does not change
  - The relative dependencies between instructions do not change
  - On-chip behavior of instructions does not change either
- Characterize the behavior of a basic block (BB) in terms of cycles:
  - On-chip cycles remain the same, since dependencies between instructions do not change
  - However, Off-chip cycles due to memory access changes due to the dynamic behavior of caches and TLBs

- On-Chip behavior (CPI) of a BB:
  - ALU operations
  - Register-Register transfer
  - Dependency, stall, etc.
- Off-Chip behavior (CPI) of a BB:
  - I/D-TLB accesses/misses,
  - I/D-Cache accesses/misses,
  - Memory accesses, etc.



Key Observation (Cont'd)

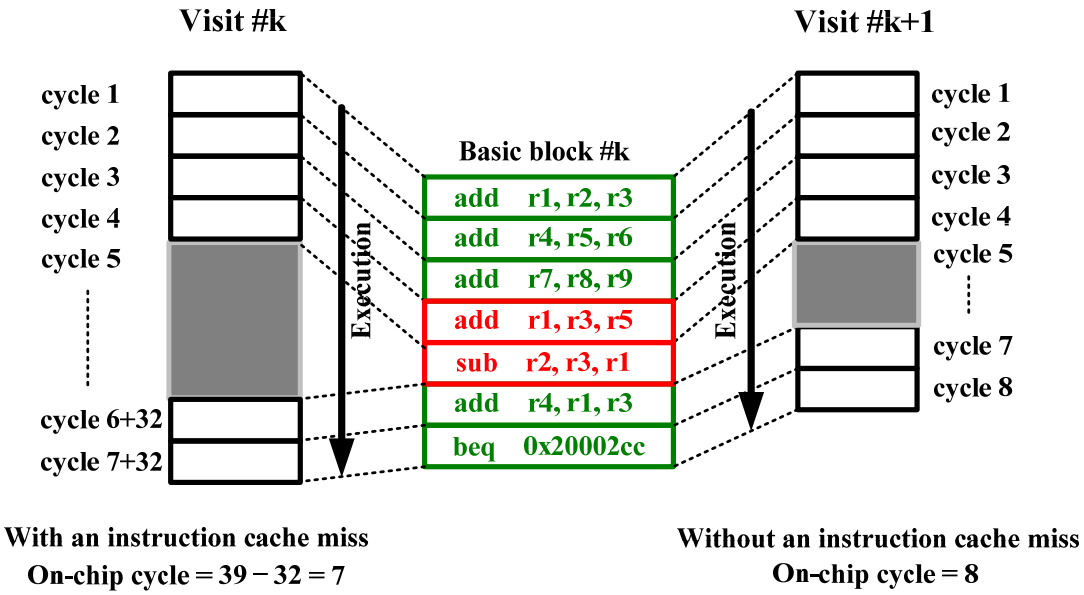
- On-Chip CPI of a BB is consistent
  - On-Chip CPI comes from (On-chip cycles = Total cycles – Off-Chip cycles)
    - Number of instruction in a BB is always fixed
  - Table shows some of the frequently visited BBs in their first 100 visits
- How to use this characteristic?
  - Once a BB is characterized in terms of its On-Chip cycles, we do not need to simulate the BB in the detailed pipeline simulator to get the On-Chip CPI

Program	BB Size (inst #)	No. of Visit	Average On-chip CPI	On-chip CPI Variance
gzip - graphic	21	41.9M	1.333	2.0e-4
	45	12.0M	1.356	5.3E-6
gcc - expr	12	77.1M	1.667	0
	37	27.2M	1.215	2.9e-5
bzip - program	44	35.5M	1.408	2.3e-4
	69	20.8M	1.231	1.9e-5
mcf - inp.in	16	93.2M	1.624	3.9e-5
	137	16.5M	1.400	6.4e-5
vortex - lendian1	20	44.6M	1.596	7.9e-4
	38	26.3M	1.581	1.3e-3

- How about Off-Chip CPI of a BB?
  - Cache simulator is needed all the time
    - Cache miss counts
    - Memory latency
- Key Idea of B^2Sim
  - If a new BB is executed in the detailed pipeline simulator, characterize its On-Chip cycles
  - For subsequent visits of the BB, avoid detailed pipeline simulation

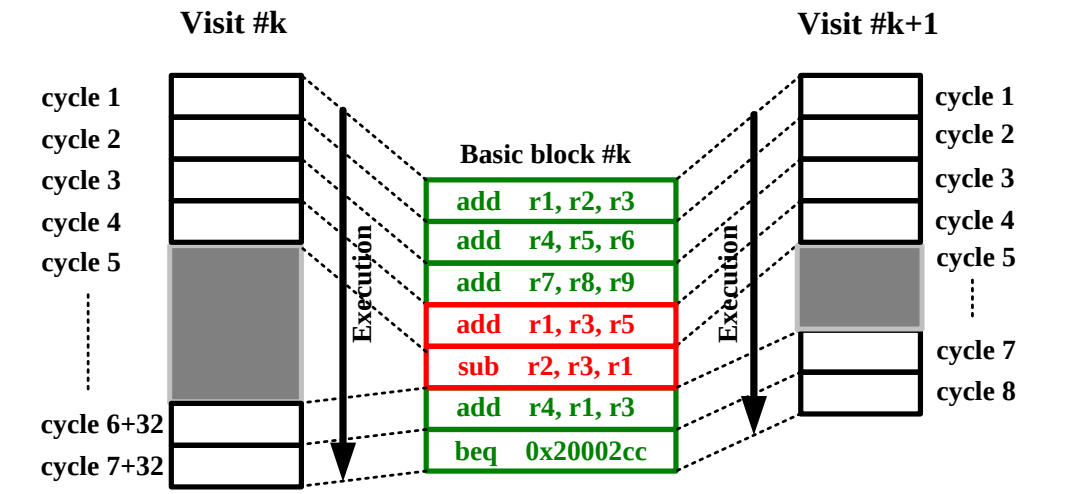
BB's On-Chip CPI Characterization

- Is it really true that (On-Chip cycles = Total cycles – Off-Chip cycles) ?
  - Given I-cache miss latency of 32 cycles
  - Assume data dependency between 4th (add r1, r3, r5) & 5th (sub r2, r3, r1) instructions
- Consider that an I-cache miss occurs in the 4th instruction
  - On-Chip cycle of BB #k is 39 - 32 = 7 (shown on the left)
- Consider that an I-cache miss does not occur in the 4th instruction
  - On-Chip cycle of BB #k is 8 (shown on the right)



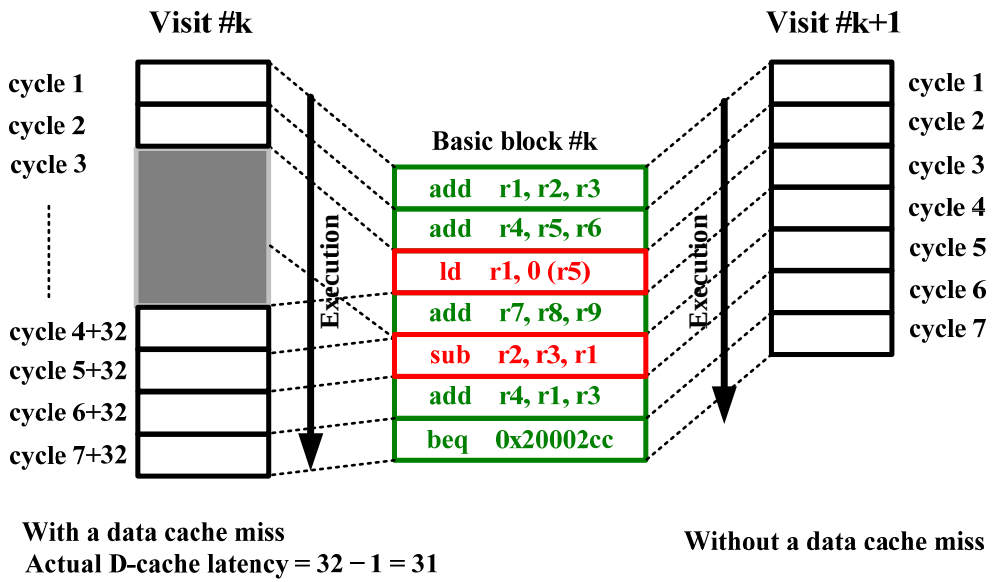
BB's On-Chip CPI Characterization (Cont'd)

- What happens? Some I-cache misses may hide the data dependencies
  - By the time 5th instruction is fetched (in a miss), 4th instruction is finished, which depends on 5th instruction
  - With an I-cache miss, no wait cycle for 5th instruction once it is fetched
  - Without an I-cache miss, dependency introduces a wait cycle to 5th instruction
- Solutions to characterize On-chip cycle?
  - Do not characterize On-Chip cycle of a BB till there is no I-/D-cache misses
  - Allow a trial of threshold value to the characterization of On-Chip cycle of a BB



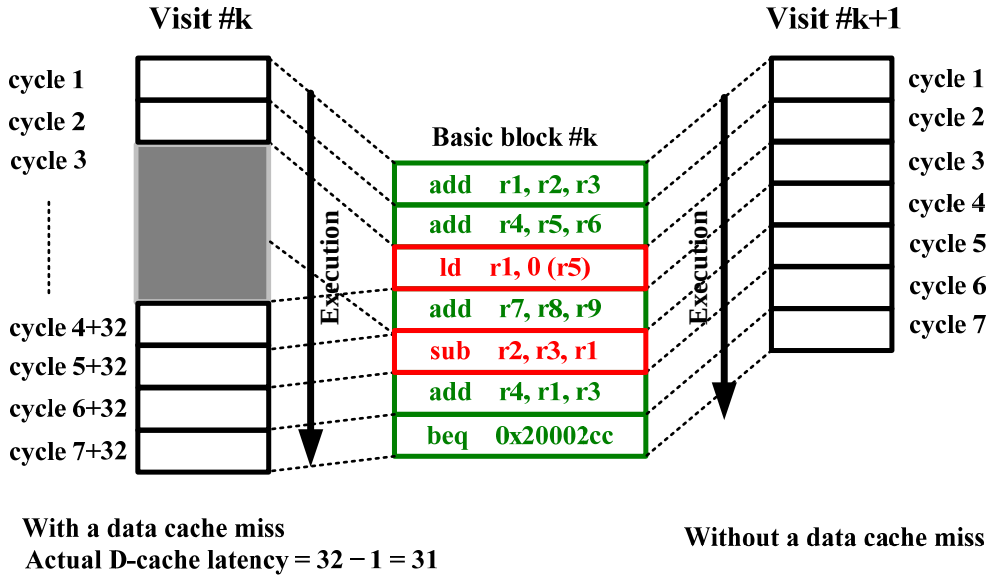
BB's Off-Chip CPI Calculation

- Is Off-Chip cycle always equal to the memory latency?
  - Given D-cache miss latency of 32 cycles
  - Assume data dependency between 3rd (ld r1 0(r5)) and 5th (sub r2, r3, r1) instructions
- Consider that a D-cache miss occurs in the 3rd instruction
  - D-Cache miss latency is 32 - 1 = 31 (shown in the left)
- Consider that a D-cache miss does not occurs in the 3rd instruction
  - D-Cache miss latency is 32 (shown in the right)



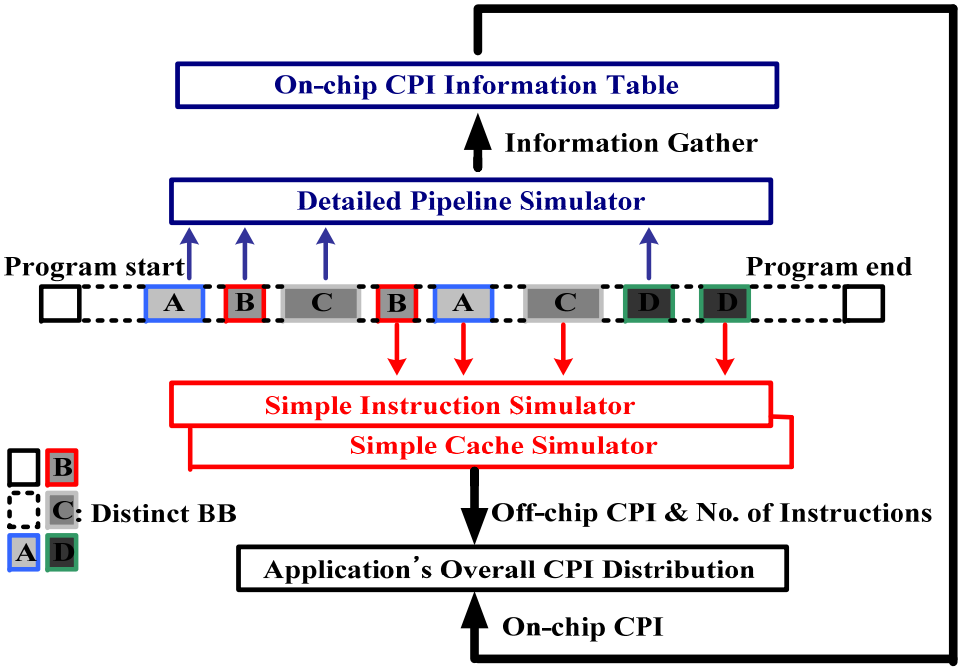
BB's Off-Chip CPI Calculation (Cont'd)

- What happens?
  - Pipeline does not stall immediately on D-cache miss
  - Instead, it stalls after one more (4th) instruction is executed
- The compiler puts independent instructions btw. dependent instructions
  - Some memory latency might be hidden since independent instructions are scheduled in between
  - Hidden latency is hard to characterize and we do not know the exact distance
- Solutions?
  - For a BB that has D-cache miss, use the average distance value



Simulator Framework

- Basic Block characterization (in terms of On-Chip & Off-Chip CPIs) based Simulator (B^2Sim)
- For each of BBs,
  - Execute sim-outorder for(/to) the first run(/visit)
  - Execute sim-cache for its subsequent run
  - BB's total cycle: On-Chip cycle + Off-Chip cycle



- Build an On-Chip CPI table that stores each BB's On-Chip CPI
  - On-Chip CPI table has;
    - # of visit
    - # of cycle
    - # of instruction in BB
- Program's CPI can be derived by the accumulation of On-Chip & Off-Chip CPIs of BBs

Simulation & Benchmark Programs

- Benchmark programs in the simulation
  - SPEC2000 INT
    - Reference/Train input files
  - MediaBench with custom input files
- Platforms for the simulation: 3 Linux machines
  - Athlon 2500+, Pentium 4 2.5GHz, Pentium 4 1.8GHz
- Table shows the architectural parameters used in B^2Sim

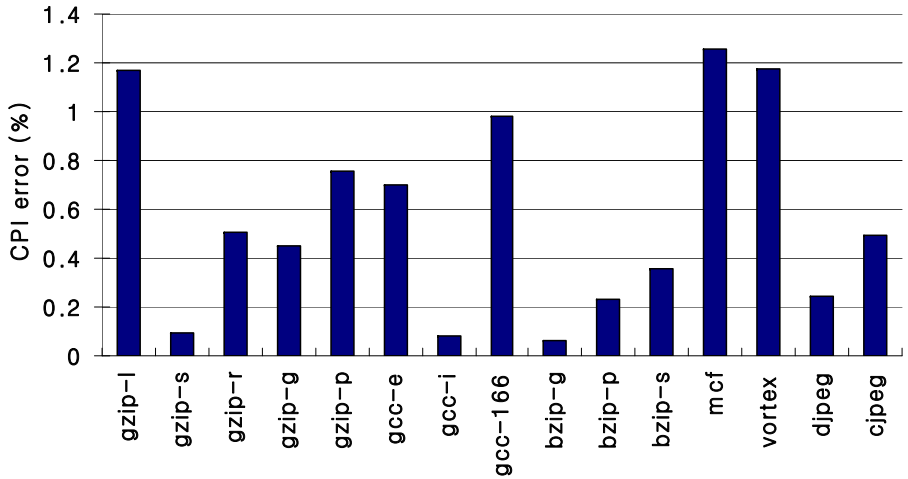
Main Memory Latency	32 cycles
L1 I/D Cache	32KB 32-way 32Byte block, 1 cycle hit latency
L2 I/D Cache	None
I/D-TLB	4-way 1024 entries, 32 cycles miss latency
Branch Predictor	Bimodal 128 Table
Functional Units	1 Integer ALU, 1 Integer MULT/DIV, 1 FP ALU, 1 FP MULT/DIV
RUU/LSQ size	8/8
Instruction Fetch Queue	8
In order Issue	True
Wrong Path Execution	True

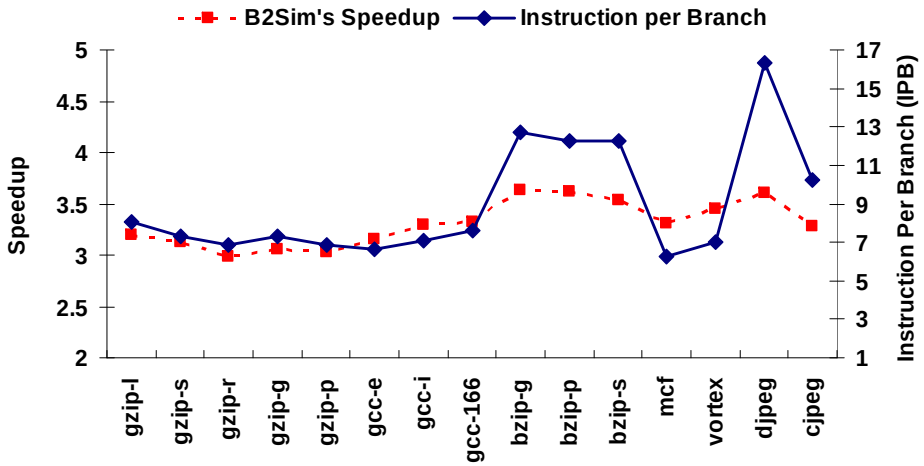
Experimental Results

Name	Native exec. time	Number of instruction	Simulation Time (minutes)			Speedup
			sim-cache	*sim-outorder*	B^2Sim	
gzip-log	2.9	4.4	28	160	50	3.20
gzip-source	5.3	8.8	55	334	107	3.12
gzip-random	5.8	7.8	51	295	99	2.98
gzip-graphic	6.3	9.4	60	365	119	3.06
gzip-program	8.9	16.8	105	646	213	3.03
gcc-expr	11.9	15.1	62	417	132	3.16
gcc-integrate	14.2	16.5	66	444	135	3.29
gcc-166	62.3	57.0	233	1505	452	3.33
bzip-graphic	15.7	24.5	129	696	191	3.64
bzip-program	12.4	20.1	107	590	162	3.62
bzip-source	10.1	16.7	90	489	135	3.54
mcf-inp.in	55.8	20.1	116	739	223	3.31
vortex-lendian1	16.6	13.0	60	404	117	3.45
djpeg-custom	1.7	2.9	17	94	26	3.61
cjpeg-custom	3.2	5.3	32	187	57	3.28

Experimental Results (Cont'd)

- Overall average CPI error on B²Sim: 0.57%
- Feasible reasons of CPI error:
  - Inter-BB data dependency
  - Sporadic I-cache misses which are not captured in the middle of On-Chip CPI characterization
- Overall average speedup of B²Sim: 3.31X
- Speedup vs. IPB (Instruction Per Branch)
  - Size and occurrence of each BB determines the speedup
  - IPB has the combined nature of both



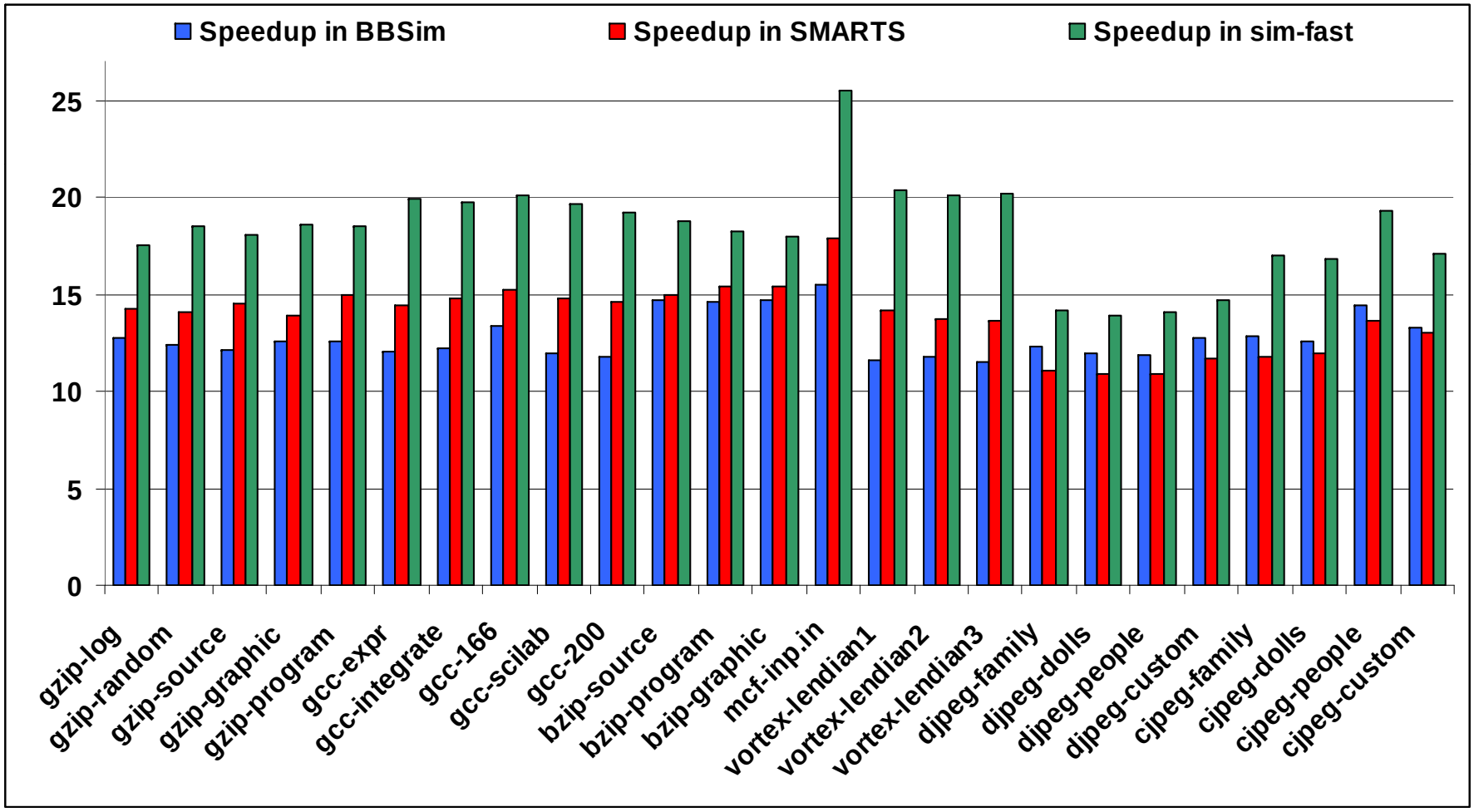


Conclusion & Future Work

- B^2Sim differs from previous simulation acceleration approaches in:
  - Minimally use detailed pipelined simulator if it is really needed
  - Do not need off-line profiling, fast-forwarding (FF) and warm-up
  - Deterministic in that it generates consistent performance over multiple runs
  - High level of granularity to capture the cycle-accurate information
- B^2Sim analyzes and utilizes BB level program behavior and information
  - On-Chip CPI of each BB is consistent
  - Once a BB is characterized with its On-Chip CPI in the pipelined simulator:
    - Functional simulator is used for those BBs afterwards
  - Off-Chip CPI of each BB temporally changes
    - I/D-cache level simulator is used for BBs all the time
- Experimental results and analysis:
  - Speed comparison (sim-cache): (sim-outorder): (B^2Sim) = 1:6:2
  - Ideal speedup of B^2Sim can reach to the speed of sim-cache. However:
    - Every BB need to be executed in the pipelined simulator at least once
    - Branch prediction and wrong/speculative execution are need over BBs
- Work to speedup B^2Sim is ongoing!

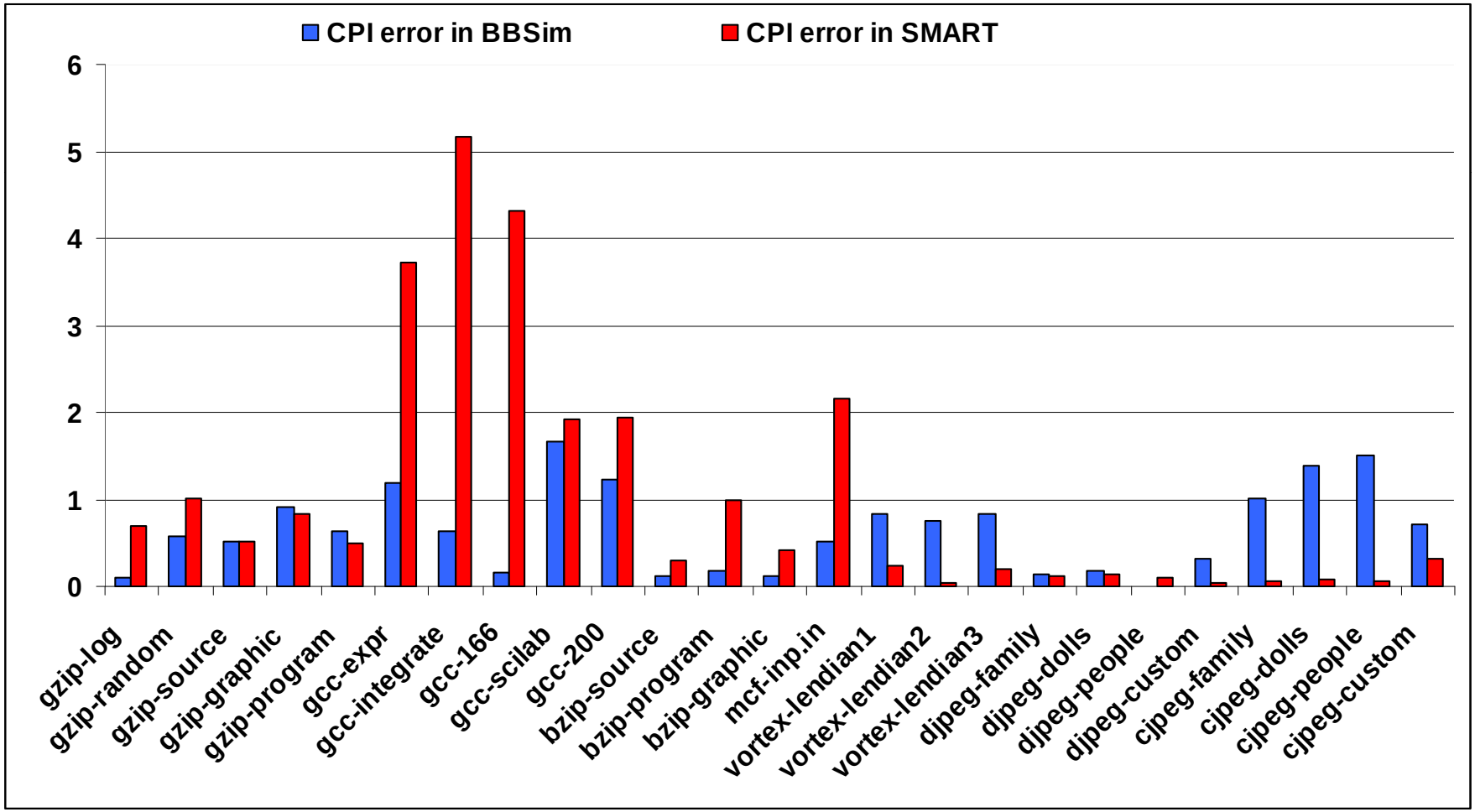
Backup: Results in Enhanced B²Sim

- Speed-up comparison (over the original sim-outorder)



Backup: Results in Enhanced B²Sim (Cont'd)

- CPI Error comparison (over the original sim-outorder)



Backup: Some details in Implementation

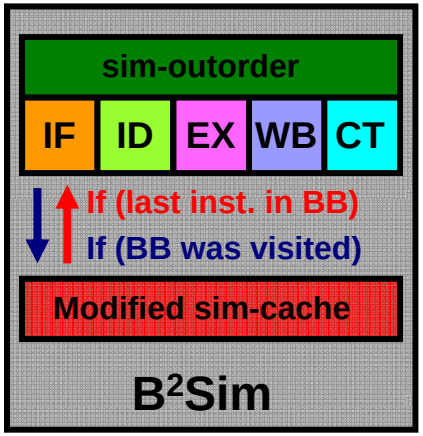
- Anatomy of B²Sim (Source Level)

```
while (inst. for the speculative & wrong path execution) {
   If (branch miss prediction occurs) {
      take branch miss prediction penalty cycle into account in switching
      // B²Sim does know on-chip/off-chip CPI of BB. However,
      // B²Sim does not know branch miss prediction penalty;
   }
}
```

```
// Basically, no new operations on this stage
  // Instructions which arrive at this stage are supposed to be executed;
```

```
If (inst. from miss predicted branch) {
   take branch miss prediction penalty cycle into account;
   // Compared to ID stage, prediction penalty cycle differs
   // Simply, one cycle difference;
}
```

```
If (BB is visited)
   switch to modified sim-cache);
else {
   while (fetch inst. until fetch-> dispatch Q fills) {
      access I-$ & I-TLB to fetch inst.;
      update I-$/I-TLB misses of BB;
   }
}
Update branch prediction( );
```




```
// Basically, no new operations on this stage
  // Instructions arrive at this stage are supposed to be committed;
```

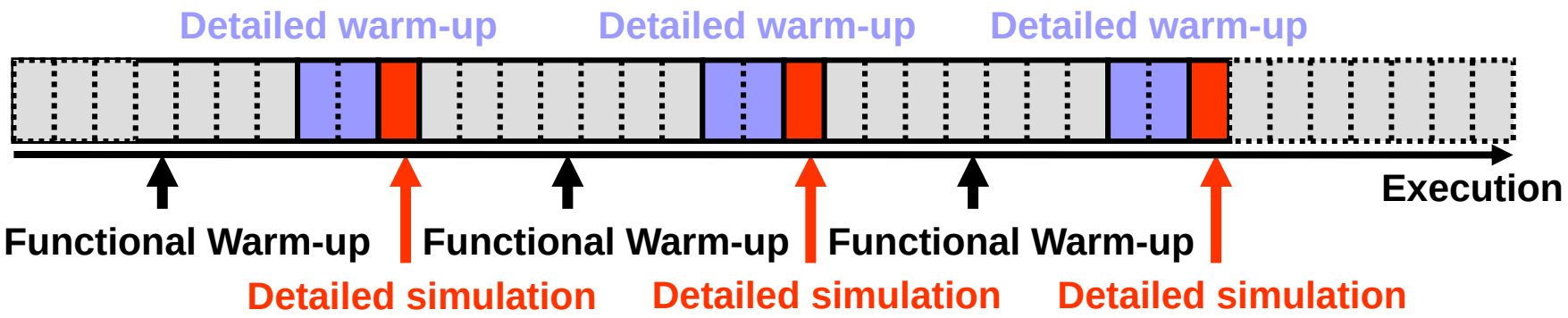
```
for (number of inst. in a BB) {
   If (total I-$ & I-TLB miss in a BB is changed) {
      access I-$ & I-TLB to update status;
   }
   access D-$ & D-TLB to update status;
}
Update cycles in a BB;
Update instruction count in a BB;
```

```
// How to measure pipeline stall count?
   // On each cycle level, if none of the pipeline stages or functions are called, we think it a pipeline stall cycle

// In Simplescalar
   // ID = Dispatch, EX = Issue
```

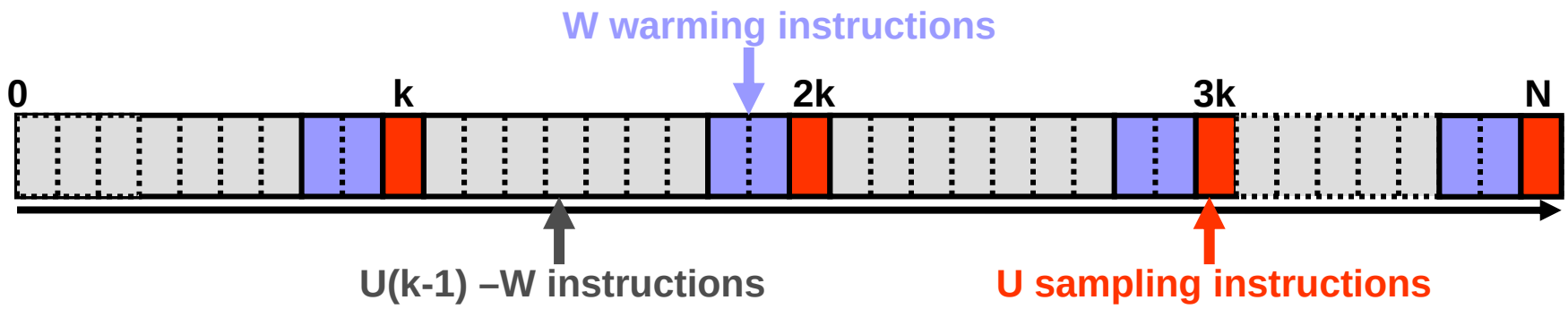
Backup: SMARTS

- SMARTS (Sampling Micro-ARchiTectural Simulator)
  - Theory of sampling:
    - Select minimal but representative samples that catch the whole program behavior by executing a periodical sampling
  - Simply, combine sim-outorder, sim-cache and sim-bpred
  - Periodically switches among three modes of operations:
    - Sampling mode, i.e., full detailed simulation and gather data
      - All micro-architectural details are accounted for
    - Detailed warm-up mode
      - Detailed simulation mode but do not gather data
    - Functional warm-up mode
      - Only cache and branch predictor are updated



Backup: SMARTS (Cont'd)

- Some details about the sampling
  - N: Benchmark length (No. of instruction), e.g., N=10 Million
  - U: Sampling unit size (No. of instruction), e.g., 1000
  - W: Detailed warm-up size (No. of instruction), e.g., 2000
  - k: Fixed sampling interval (No. of instruction), e.g., k = 1000
  - n = N/k: Sampling is executed in this many time, e.g., n=10,000
  - How to determine U? (What amount do we sample at a time?)
    - See CPI coefficient of variance w.r.t. increasing U size
    - 1000 < U is enough to saturate the variance (see the backup slide)
  - How to determine n/k? (How often we sample in overall execution?)
    - n > 10,000 yields 99.7% of confidence interval which is 4% CPI err



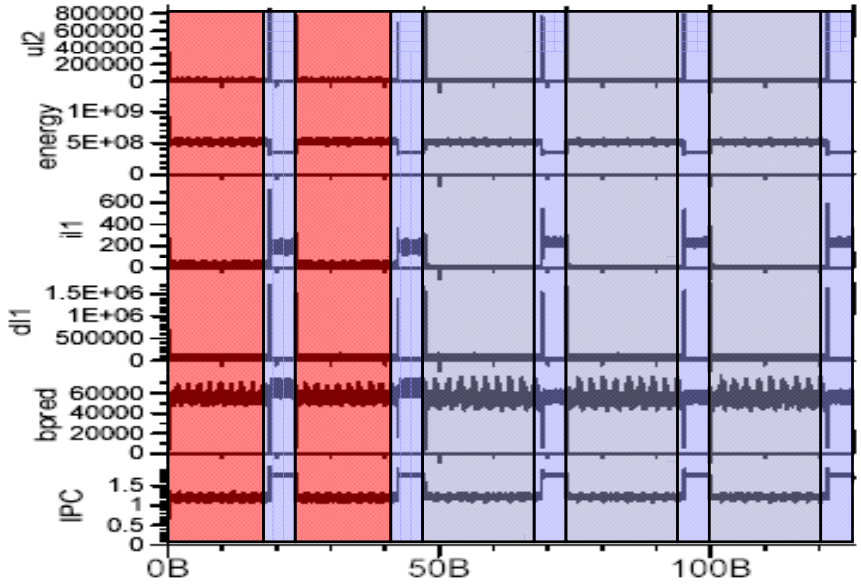
Backup: SMARTS (Cont'd)

- Performance of SMARTS
  - Speedup: 35X ~ 60X (over sim-outorder) but 1X-1.5X (over sim-fast)
    - N=10,000, U=1,000, W=2,000
  - Average CPI error: 0.64%
- Observations in SMARTS:
  - Combine wattch makes sim-outorder to be slow
    - Reference simulation time: (outorder) : (cache) : (fast) is 18:6:1
  - Aggressive code optimization
    - Simulation time acceleration by data structure change?

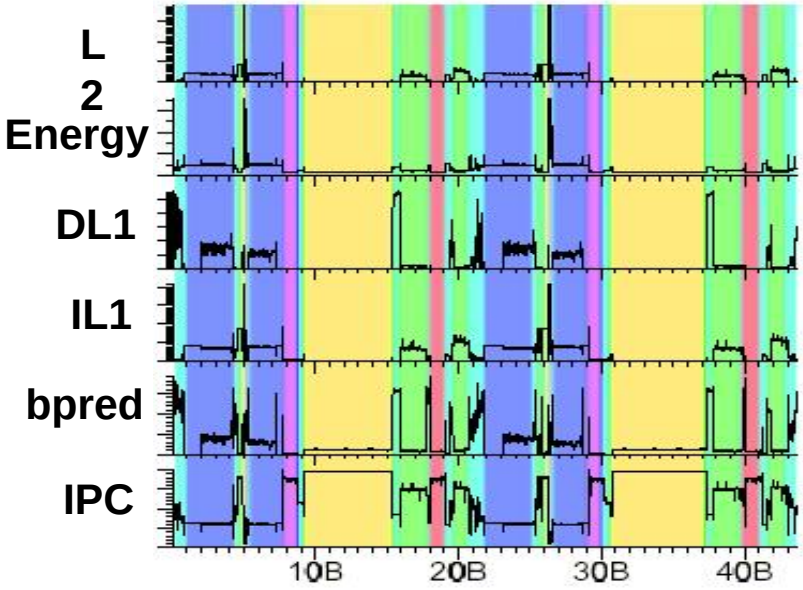
Runtime	parser	migrid	twolf	mesa	ammp	gap	swim	average
sim-outorder	541	414	343	279	323	266	223	98
sim-fast	9.2	7.0	5.8	4.7	5.5	4.5	3.8	1.7
SMARTS	15.8	12.1	10.0	8.1	9.6	7.8	6.5	2.9

Backup: SimPoint

- SimPoint 3.0
  - Program behavior (phase) based sampling simulation
- How and why it works?
  - Program behaviors change over time. However, they are not random but repeat themselves in some cyclic patterns
  - Sample some periods/points that reflect the whole program behavior
- Problem
  - Phase characterization in a program may not easy



gzip



gcc

Backup: SimPoint (Cont'd)

- How to determine the similarity between any two intervals
  - Each predefined interval size is ranged from 10M ~100M cycles
  - Compare basic block vector (BBV) distance of each interval
  - Draw a similarity matrix

```
                                ID:  1   2  3   4  5  ..
                  BB Exec Count: <1, 20, 0,  5, 0, …>
              Weigh by Block Size: <8,  3, 1,  7, 2, …>
                                 = <8, 60, 0, 35, 0, …>
                Normalize to 1 = <8%,58%,0%,34%,0%,…>
```

BB	Assembly Code of bzip

1	srl a2, 0x8, t4
	and a2, 0xff, t12
	addl zero, t12, s6
	subl t7, 0x1, t7
	cmpeq s6, 0x25, v0
	cmpeq s6, 0, t0
	bis v0, t0, v0
	bne v0, 0x120018c48

2	subl t7, 0x1, t7
	cmple t7, 0x3, t2
	beq t2, 0x120018b04

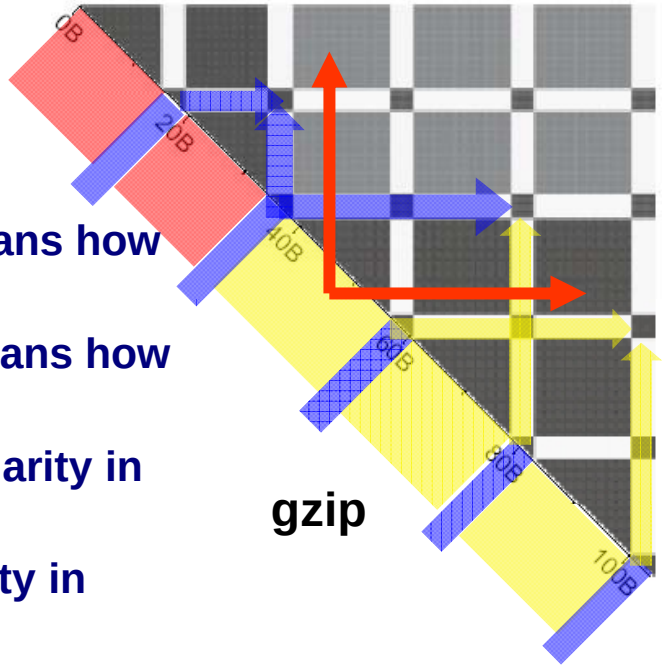
3	ble t7, 0x120018bb4

4	and t4, 0xff, t5
	srl t4, 0x8, t4
	addl zero, t5, s6
	cmpeq s6, 0x25, s0
	cmpeq s6, 0, a0
	bis s0, a0, s0
	bne s0, 0x120018c48

5	subl t7, 0x1, t7 gt t7, 0x120018b90
...	...

< Example of BBs >

- In a similarity matrix
  - Darker gray area means how similar they are
  - Lighter gray area means how different they are
  - Horizontal line: Similarity in future sample
  - Vertical line: Similarity in previous sample



gzip

Backup: SimPoint (Cont'd)

- No. of simulation points (K) for interval=100K (djpeg) and interval=10M (gzip)

Program (input)	Total IPC	Total Time	K=1			K=2			K=3			Simulation Points
			IPC	Weight	Time	IPC	Weight	Time	IPC	Weight	Time	
djpeg (pic.jpg)	0.654	103.1	0.654	1.0	103.1							6543
	0.714	249.6	0.714	0.542	91.2	0.712	0.458	158.4				4955, 10133
	0.716	218.1	0.747	0.198	36.9	0.684	0.262	31.0	0.718	0.540	150.2	2374, 1983, 9575
	0.715	292.4	0.756	0.139	22.6	0.718	0.540	150.2	0.679	0.210	85.1	1442, 9575, 5493, 2151
gzip (log)	0.6895	379.5	0.689	1.0	379.5							283
	0.6973	723.2	0.740	0.155	343.7	0.689	0.845	379.5				195, 283
	0.6989	1032.8	0.689	0.435	379.9	0.694	0.410	309.2	0.740	0.155	343.6	308, 163, 195
	0.7136	1356.3	0.689	0.437	379.9	0.729	0.178	331.5	0.740	0.155	343.6	308, 171, 195, 85

- djpeg-pic.jpg
  - Number of instruction: 1.014B, Simulation cycles: 1.325B, Total IPC: 0.7648
  - Total simulation time in original sim-outorder: 2034 seconds
- gzip-log
  - No. of instruction: 4.444B, Simulation cycles: 6.070B Total IPC: 0.7321
  - Total simulation time in original sim-outorder: 1356 seconds
- Conclusions: The performance and the accuracy of SimPoint largely varies over the determination of K. Also, the interval size matter.

Backup: Other Prior Works

- Simulation with Reduced Benchmark
  - MinneSPEC: Modify the reference input dataset such that it still retains the characteristics of the reference input dataset yet reduces the simulation time
  - SPEClite: Alternating UA (Microarchitectural mode) and FM (Functional mode) and sample the representative 20-50 periods of one million instructions
- Statistical Simulation (HLS++)
  - Step 1: Profiling the benchmark program to measure a collection of its execution characteristics to create statistical profile.
  - Step 2: Generate a synthetic trace using the statistical profile
    - Synthetic trace consists of instructions contained in basic blocks that are linked together into a control flow graph (CFG)
    - Instead of actual arguments and op-codes, each instruction is composed of a set of statistical parameters such as instruction type, I/D-cache hit probability, I-/D-TLB hit probability, branch miss probability, dynamic data dependency distance (to determine how far a consumer instruction is away from its producer), etc.
  - Step 3: Simulating the instructions in the synthetic trace on a trace driven simulator to obtain the performance estimate (Statistical simulation)

Backup: Other Prior Works (Cont'd)

- Accelerate the Warm-up
  - To provide the correct cache & branch predictor states and handle the cold start
  - Checkpoint: Store the hardware state at the beginning of each sample and impose this state when simulating the sampled trace
  - Prime: an empty cache at the beginning of each sample and uses a certain percent of each sample to warm-up the cache
  - MRRL (Memory Reference Reuse Latency): A histogram of the number of memory references between two references to the same memory location is used to determine when to start the warm-up
- Accelerate the Fast-forwarding (SimSnap)
  - Fast-forwarding takes time and it usually employs functional simulation within the cycle-accurate simulator model, e.g., (native): (functional): (cycle-accurate): (1):(32):(32X42)
  - Substitute functional simulation with native (real-time) execution using checkpoint to transfer application state to a simulator at the desired simulation points (a.k.a. Native FF)
  - Specifically, application level check-pointing (ALC) insert code to save function call sequence along with run-time information of data size, layout, etc.
  - Modify the program to capture its own internal, dynamic state such that program can restart