Automatically Characterizing Large Scale Program Behavior

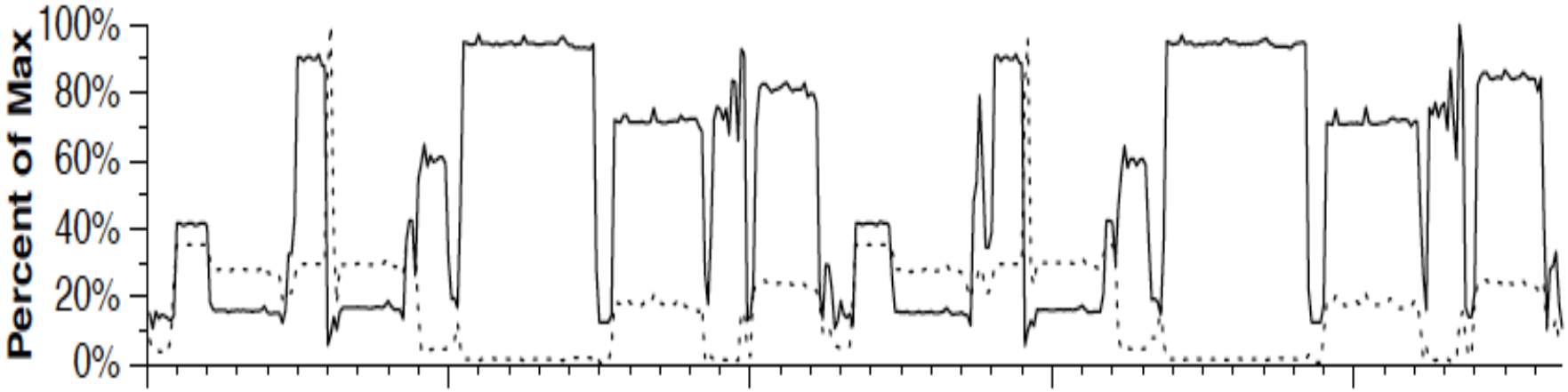
Timothy Sherwood Erez Perelman Greg Hamerly Brad Calder

Department of Computer Science and Engineering
University of California, San Diego

{sherwood,eperelma,ghamerly,calder}@cs.ucsd.edu

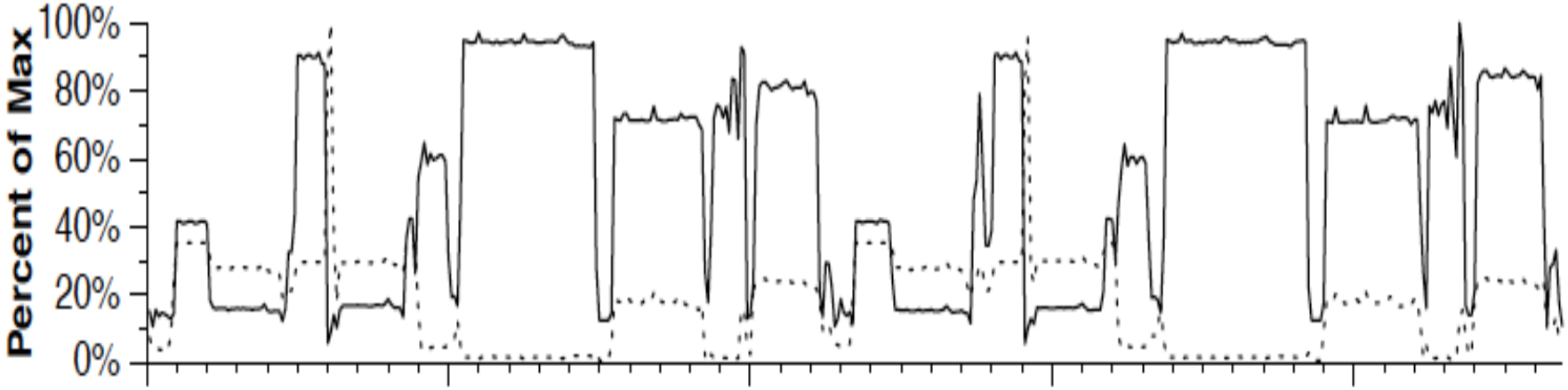
Phases in Programs

1. Architecture
2. Dynamic Optimizations
3. Compiler Optimizations
4. Power Management



Phases in Programs

1. Fine-grain (1-10 instructions)
2. Coarse grain (1000-10000 instructions)
3. Large Scale (eg: 100 M chunks)



BBV

1. Basic Block Vector
2. What is a Basic Block?
3. A continuous sequence of code with one entry point and one exit point
4. How many basic blocks in the following code

```
L1:   I1
L2:   I2
L3:   I3
      I4
      If cond true go to L3
L6:   I6
L7:   Loop L1
```

Basic Blocks

1. How many basic blocks in the following code

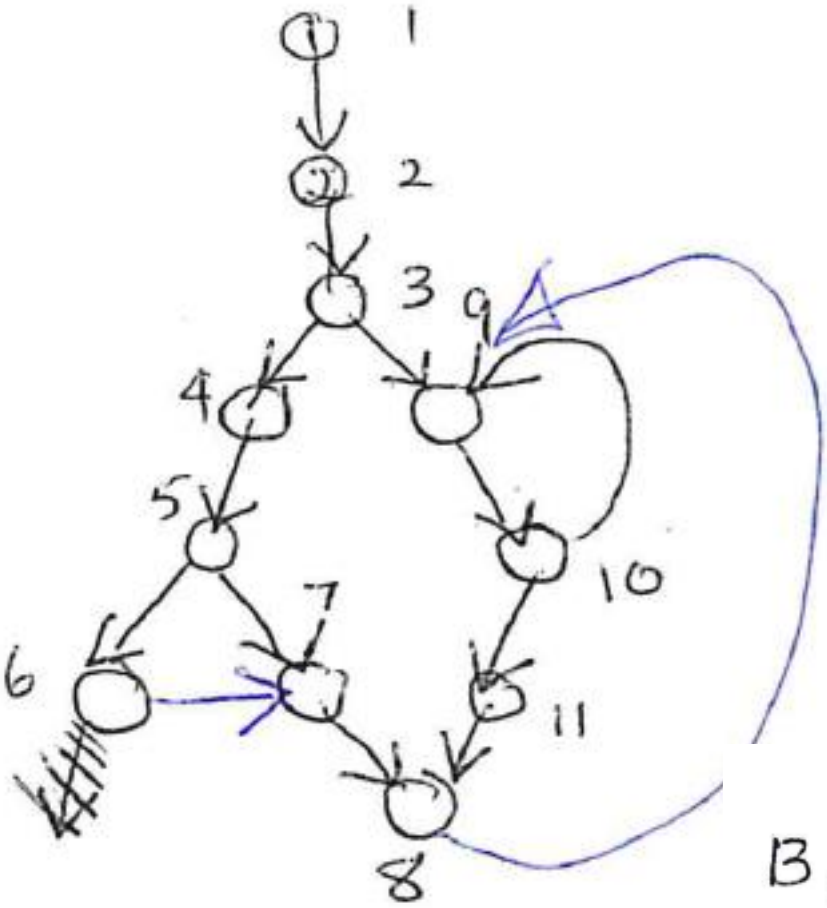
```
L1:   I1
L2:   I2
L3:   I3
      I4
      If cond true go to L3
L6:   I6
L7:   Loop L1
```

Answer: 3 basic blocks

B1 = {I1, I2} ; I3 is an entry so cannot be in this BB

B2 = {I3, I4,I5}

B3 = {I6, I7}



Assume each node is an instruction.
Each arc shows flow of program control.
Find basic blocks in this program graph

$B_1 = \{1, 2, 3\}$.

$B_2 = \{4, 5\}$

$B_3 = \{6\}$

$B_4 = \{7\}$

$B_5 = \{8\}$

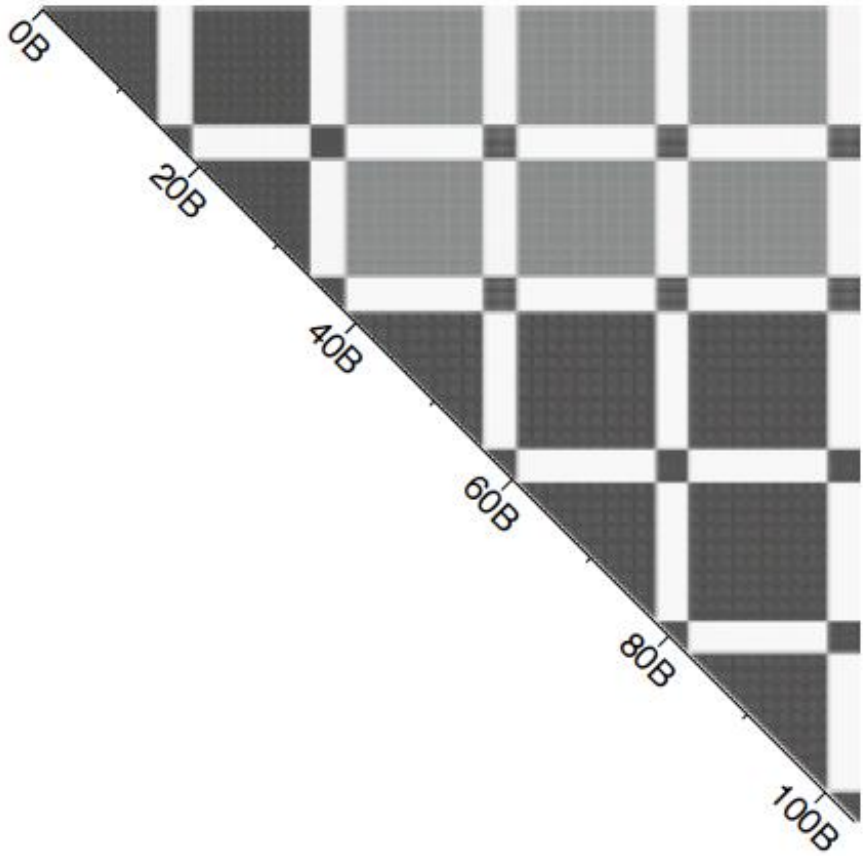
$B_6 = \{9, 10\}$

$B_7 = \{11\}$

Static Basic Blocks vs Dynamic

1. Can static and dynamic basic blocks be different?
2. What we did on previous page was static
3. Many profiling tools just identify dynamic basic blocks
4. How can dynamic basic blocks be different in the following example?

```
L1:   I1
L2:   I2
L3:   I3
      I4
      If cond true go to L3
L6:   I6
L7:   Loop L1
```



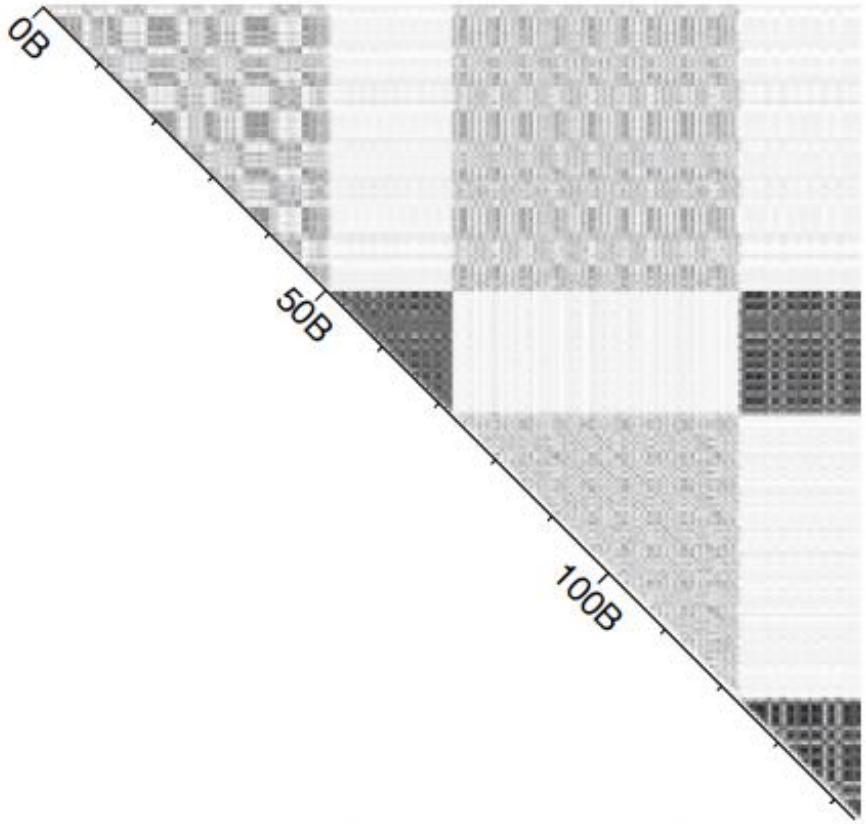


Figure 1: Basic block similarity matrix for the programs `gzip-graphic` (shown left) and `bzip-graphic` (shown right). The diagonal of the matrix represents the program's execution to completion with units in billions of instructions. The darker the points, the more similar the intervals are (the Manhattan distance is closer to 0), and the lighter the points the more different they are (the Manhattan distance is closer to 2).

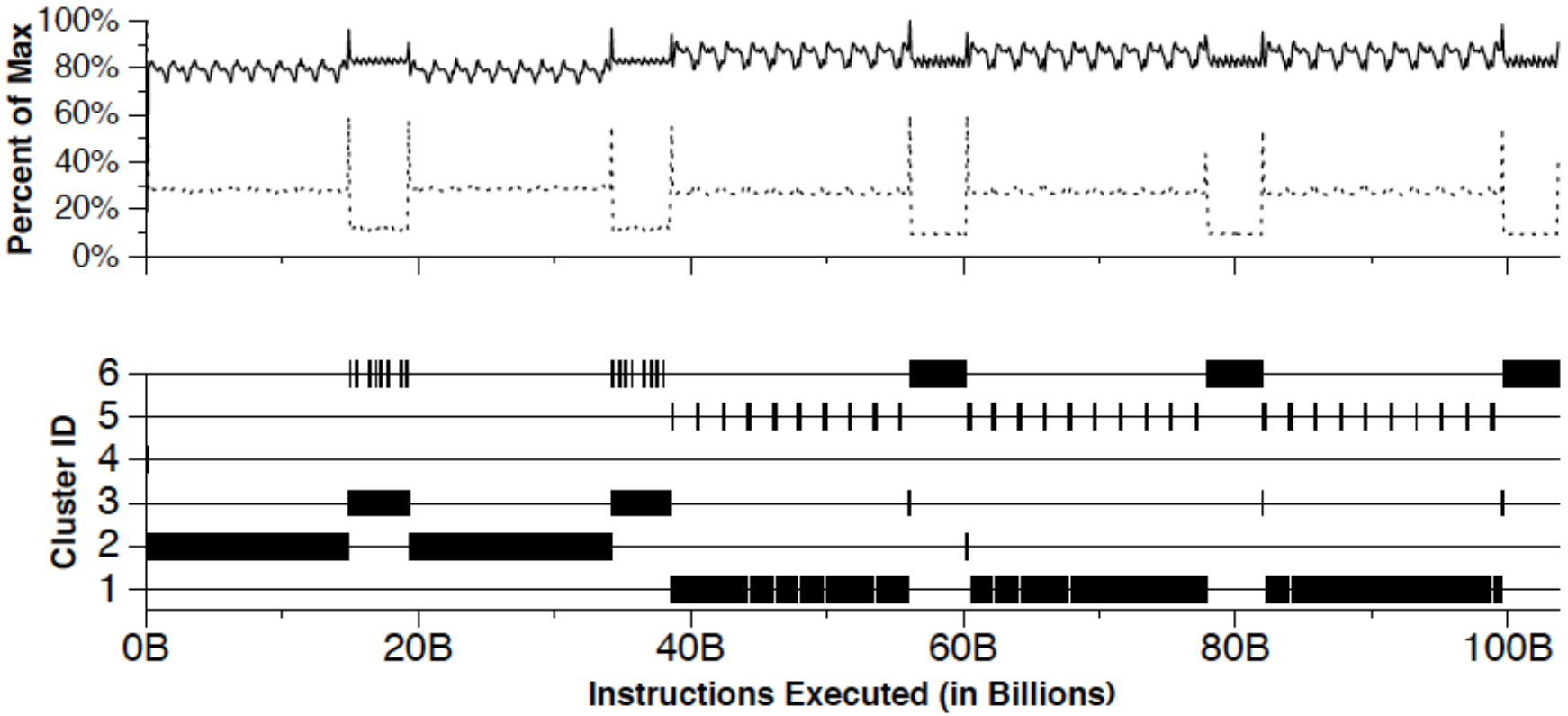


Figure 2: (top graph) Time varying graph for `gzip-graphic`. The average IPC (drawn with solid line) and L1 data cache miss rate (drawn with dotted line) are plotted for every interval (100 million instructions of execution) showing how these metrics vary over the program's execution. The x-axis represents the execution of the program over time, and the y-axis the percent of max value the metric had during execution. The results are non-accumulative.

Figure 3: (bottom graph) Cluster graph for `gzip-graphic`. The full run of the execution is partitioned into a set of 6 clusters. The x-axis is in instructions executed, and the graph shows for each interval of execution (every 100 million instructions), which cluster the interval was placed into.

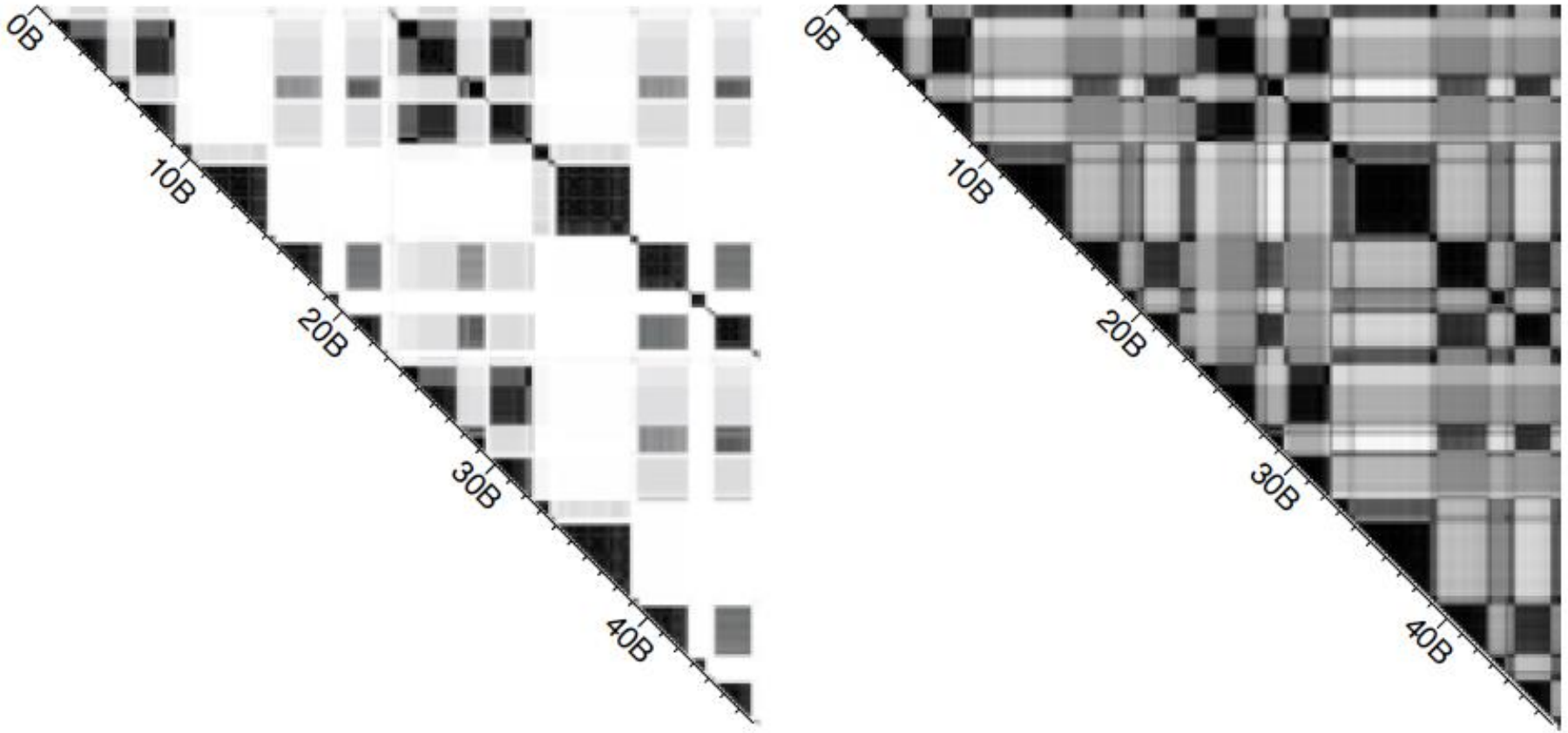


Figure 4: The original basic block similarity matrix for the program gcc (shown left), and the similarity matrix for gcc-166 drawn from projected data (on right). The figure on the left use the original basic block vectors (each of which has over 100,000 dimensions) and uses the Manhattan distance as a method of difference taking. The figure on the right uses projected data (down to 15 dimensions) and uses the Euclidean distance for difference taking.

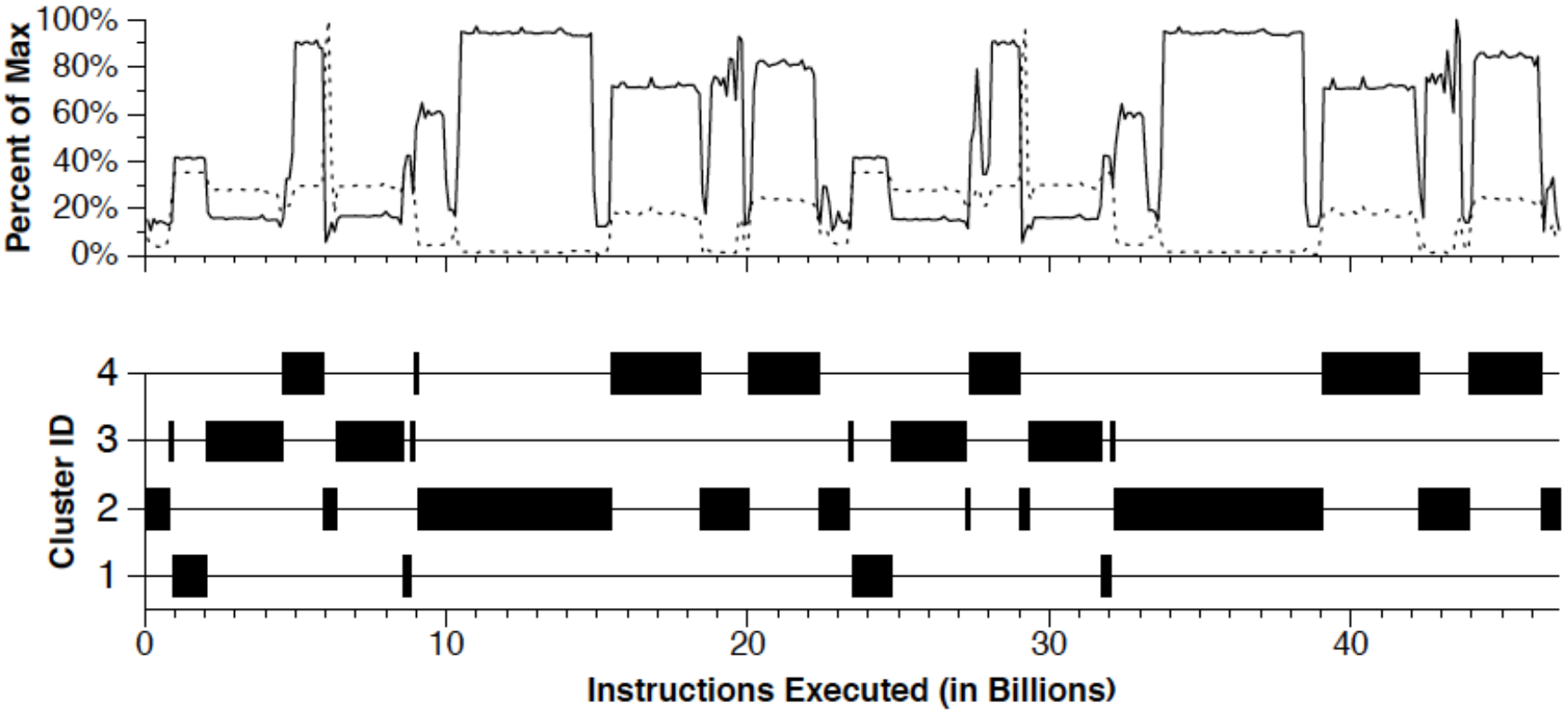


Figure 5: (top graph) Time varying graph for gcc-166. The average IPC (drawn with solid line) and L1 data cache miss rate (drawn with dotted line) are plotted for every interval (100 million instructions of execution) showing how these metrics vary over the program's execution. The x-axis represents the execution of the program over time, and the y-axis the percent of max value the metric had during execution. The results are non-accumulative.

Figure 6: (bottom graph) Cluster graph for gcc-166. The full run of the execution is partitioned into a set of 4 clusters. The x-axis is in instructions executed, and the graph shows for each interval of execution (every 100 million instructions), which cluster the interval was placed into.

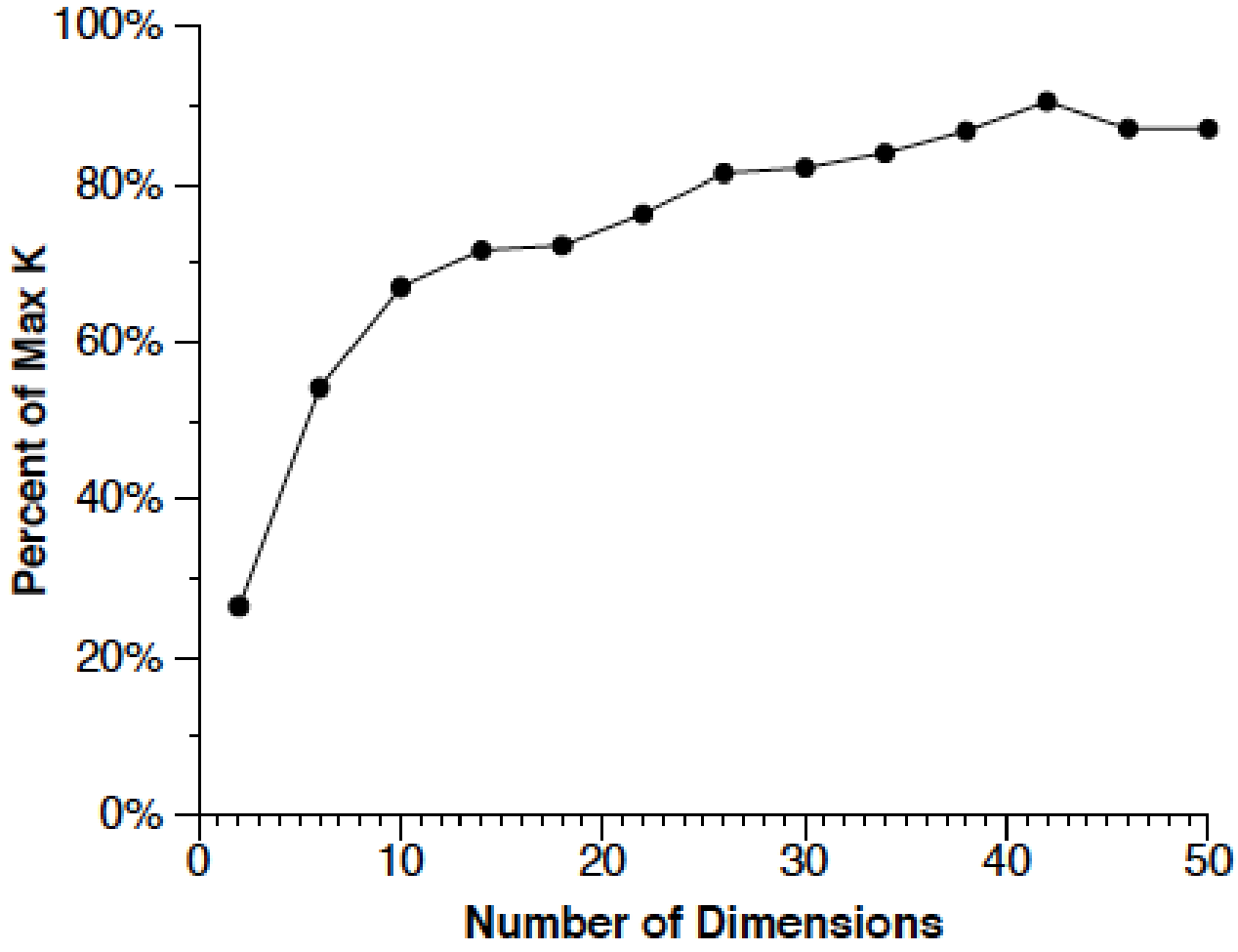
Phase Finding Algorithm

1. Profile the basic blocks executed in each program to generate the basic block vectors for every 100 million instructions of execution.

2. Reduce the dimension of the BBV data to 15 dimensions using random linear projection.

3. Try the k-means clustering algorithm on the low-dimensional data for k values 1 to 10. Each run of k-means produces a clustering, which is a partition of the data into k different clusters.

4. For each clustering ($k = 1 \ldots 10$), score the fit of the clustering using the BIC. Choose the clustering with the smallest k, such that it's score is at least 90% as good as the best score.


100%
80%
60%
40%
20%
0%
Percent of Max K
0
10
20
30
40
50
Number of Dimensions

Bayesian Information Criterion – A penalized likelihood

$$BIC(D,k) = l(D|k) - \frac{p_j}{2}\log(R)$$

where $l(D|k)$ is the likelihood, R is the number of points in the data, and p_j is the number of parameters to estimate,

$$l(D|k) = \sum_{i=1}^{k} -\frac{R_i}{2}\log(2\pi) - \frac{R_i d}{2}\log(\sigma^2) - \frac{R_i - 1}{2} + R_i \log(R_i/R)$$

where R_i is the number of points in the ith cluster, and σ^2 is the average variance of the Euclidean distance from each point to its cluster center.

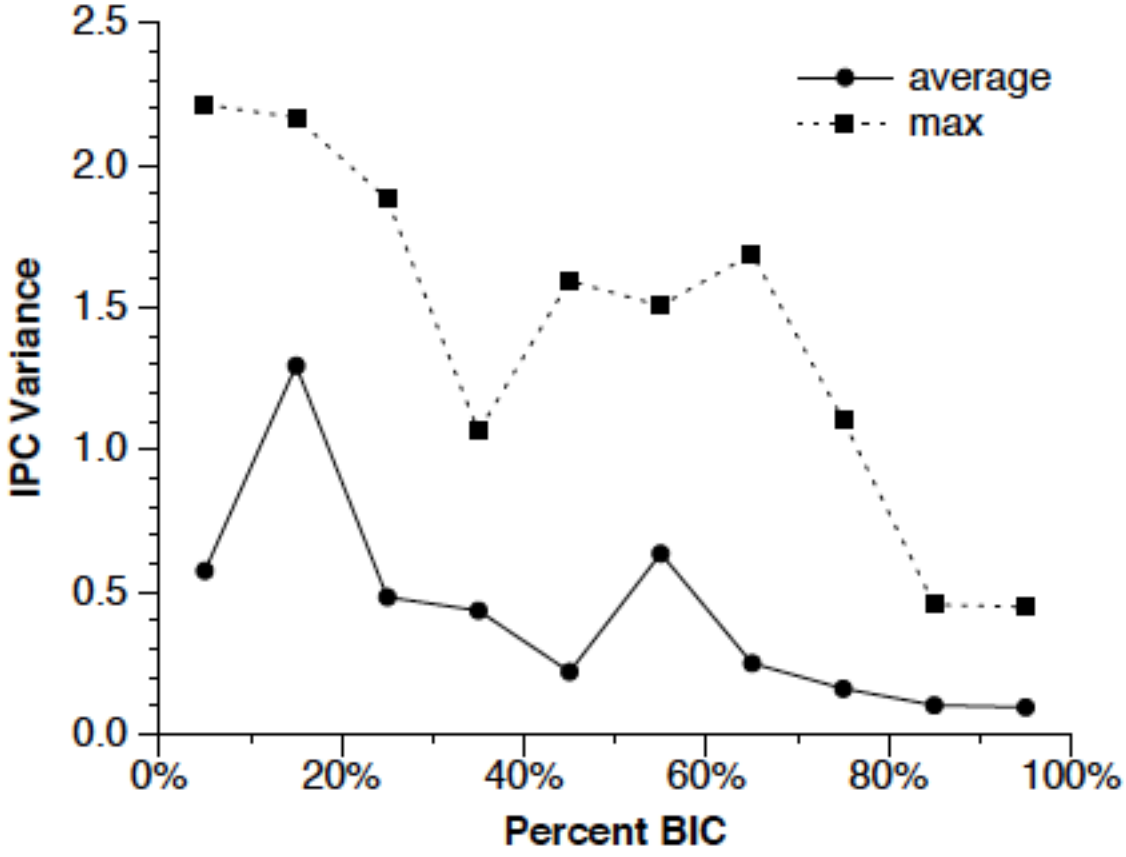


Figure 8: Plot of average IPC variance and max IPC variance versus the BIC. These results indicate that for our data, a clustering found to have a BIC score greater than 80% will have, on average, and IPC variance of less than 0.2.

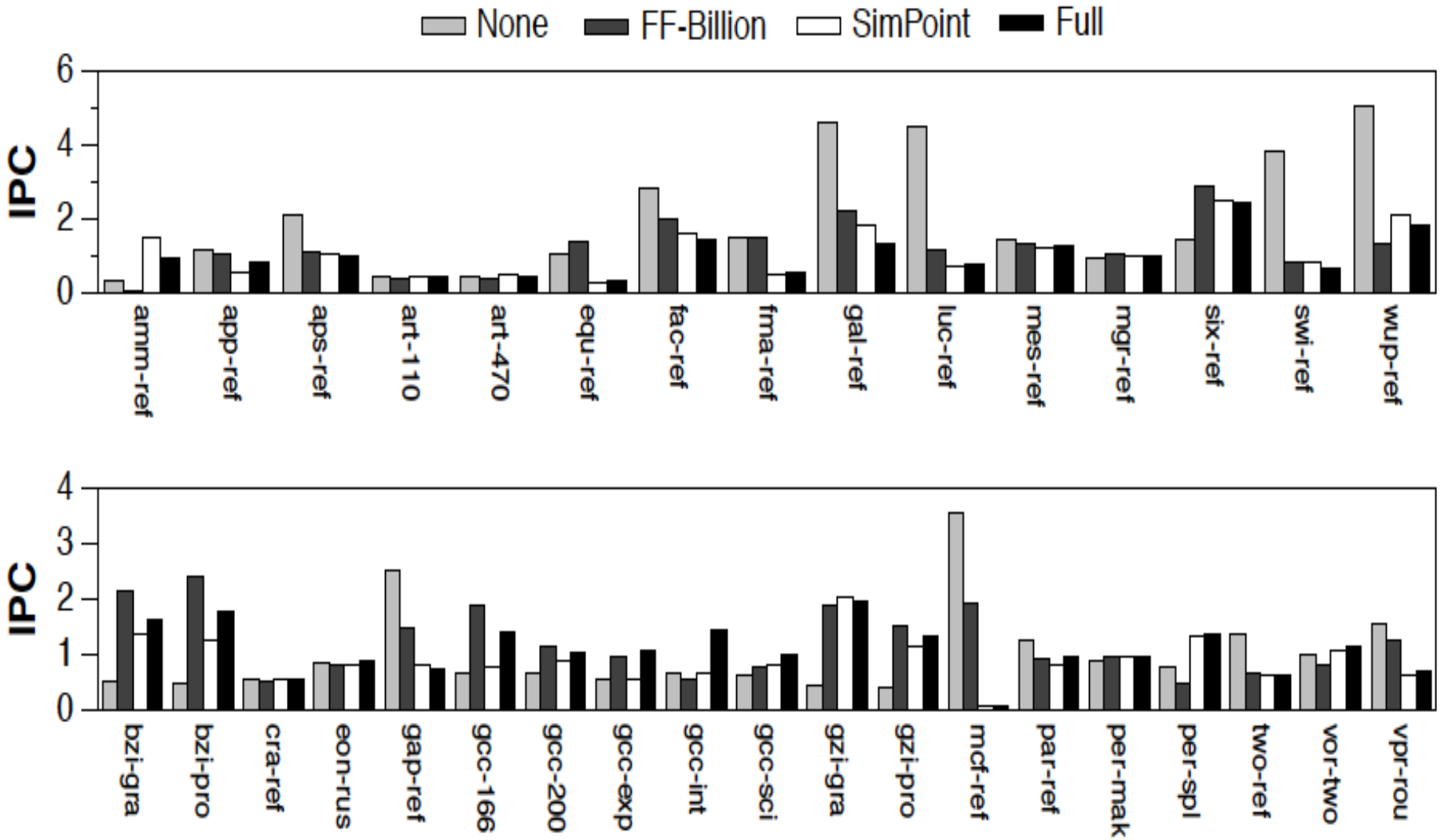


Figure 9: Simulation results starting simulation at the start of the program (none), blindly fast forwarding 1 billion instructions, using a single simulation point, and the IPC of the full execution of the program.

name	Len	Init	SP	PC	Proc Name	Multiple SimPoints				
ammp	3265	24	109	026834	mm_fv_update.	3026(13.8) 1607(12.6)	1774(31) 2437(4.9)	595(15.3) 3112(11.5)	1068(1.3) 2480(2.2)	2128(7.4)
applu	2238	3	2180	018520	buts_	624(22.1) 1507(14.5)	1625(22.5)	1956(18.8)	2234(6.6)	1380(15.5)
apsi	3479	3	3409	0380ac	dctdxf_	2107(5.6)	2863(14)	1007(70.7)	896(7.7)	1618(2)
art-110	417	75	341	00fbb0	match	82(42.9)	255(41.2)	50(15.8)		
art-470	450	83	366	00f5d0	match	300(36.2)	46(14.7)	236(49.1)		
bzip2-graphic	1435	4	719	012a5c	spec_putc	168(11.7) 519(11.6)	1042(3.7) 872(8.2)	430(7.5) 195(5.6)	762(16.2) 148(2)	106(15.3) 1435(18.2)
bzip2-program	1249	4	459	00ddd0	sortIt	140(11) 1005(7)	468(12.3) 94(6.9)	78(6.2) 606(14)	990(16) 859(14.6)	445(7.4) 341(4.7)
bzip2-source	1088	4	978	00d774	qSort3	395(16) 177(34.7)	511(4.3)	64(29.1)	488(7.3)	530(8.6)
crafty	1918	462	775	021730	SwapXray	123(25)	510(19.7)	664(22.7)	1123(32.5)	
eon-rushmeier	578	140	404	04e1b4	viewingHit	260(6.6)	238(23.7)	337(20.9)	435(35.6)	216(13.1)
equake	1315	35	813	012410	phi0	874(12.2) 62(11.6)	1292(36.7)	463(12.2)	336(24.1)	3(3.2)
facerec	2682	356	376	02d1f4	graphroutines_lo.	1976(60.1)	1528(2.5)	1935(3.9)	1398(29.2)	348(4.3)
fma3d	2683	192	2542	0e3140	scatter_element.	112(7) 509(13)	209(0.6)	842(68.4)	1600(11)	47(0.1)
galgel	4093	3	2492	02db00	syshtn_	3511(5.5) 2181(29)	2081(11) 2161(3.3)	3466(11.2) 1017(5.5)	516(31.6)	2141(2.7)

name	Len	Init	SP	PC	Proc Name
ammp	3265	24	109	026834	mm_fv_update.
applu	2238	3	2180	018520	buts_
apsi	3479	3	3409	0380ac	dctdxf_
art-110	417	75	341	00fbb0	match
art-470	450	83	366	00f5d0	match
bzip2-graphic	1435	4	719	012a5c	spec_putc
bzip2-program	1249	4	459	00ddd0	sortIt
bzip2-source	1088	4	978	00d774	qSort3
crafty	1918	462	775	021730	SwapXray
eon-rushmeier	578	140	404	04e1b4	viewingHit
equake	1315	35	813	012410	phi0

Multiple SimPoints				
3026(13.8) 1607(12.6)	1774(31) 2437(4.9)	595(15.3) 3112(11.5)	1068(1.3) 2480(2.2)	2128(7.4)
624(22.1) 1507(14.5)	1625(22.5)	1956(18.8)	2234(6.6)	1380(15.5)
2107(5.6)	2863(14)	1007(70.7)	896(7.7)	1618(2)
82(42.9)	255(41.2)	50(15.8)		
300(36.2)	46(14.7)	236(49.1)		
168(11.7) 519(11.6)	1042(3.7) 872(8.2)	430(7.5) 195(5.6)	762(16.2) 148(2)	106(15.3) 1435(18.2)
140(11) 1005(7)	468(12.3) 94(6.9)	78(6.2) 606(14)	990(16) 859(14.6)	445(7.4) 341(4.7)
395(16) 177(34.7)	511(4.3)	64(29.1)	488(7.3)	530(8.6)
123(25)	510(19.7)	664(22.7)	1123(32.5)	
260(6.6)	238(23.7)	337(20.9)	435(35.6)	216(13.1)
874(12.2) 62(11.6)	1292(36.7)	463(12.2)	336(24.1)	3(3.2)
1976(60.1)	1528(2.5)	1935(3.9)	1398(29.2)	348(4.3)

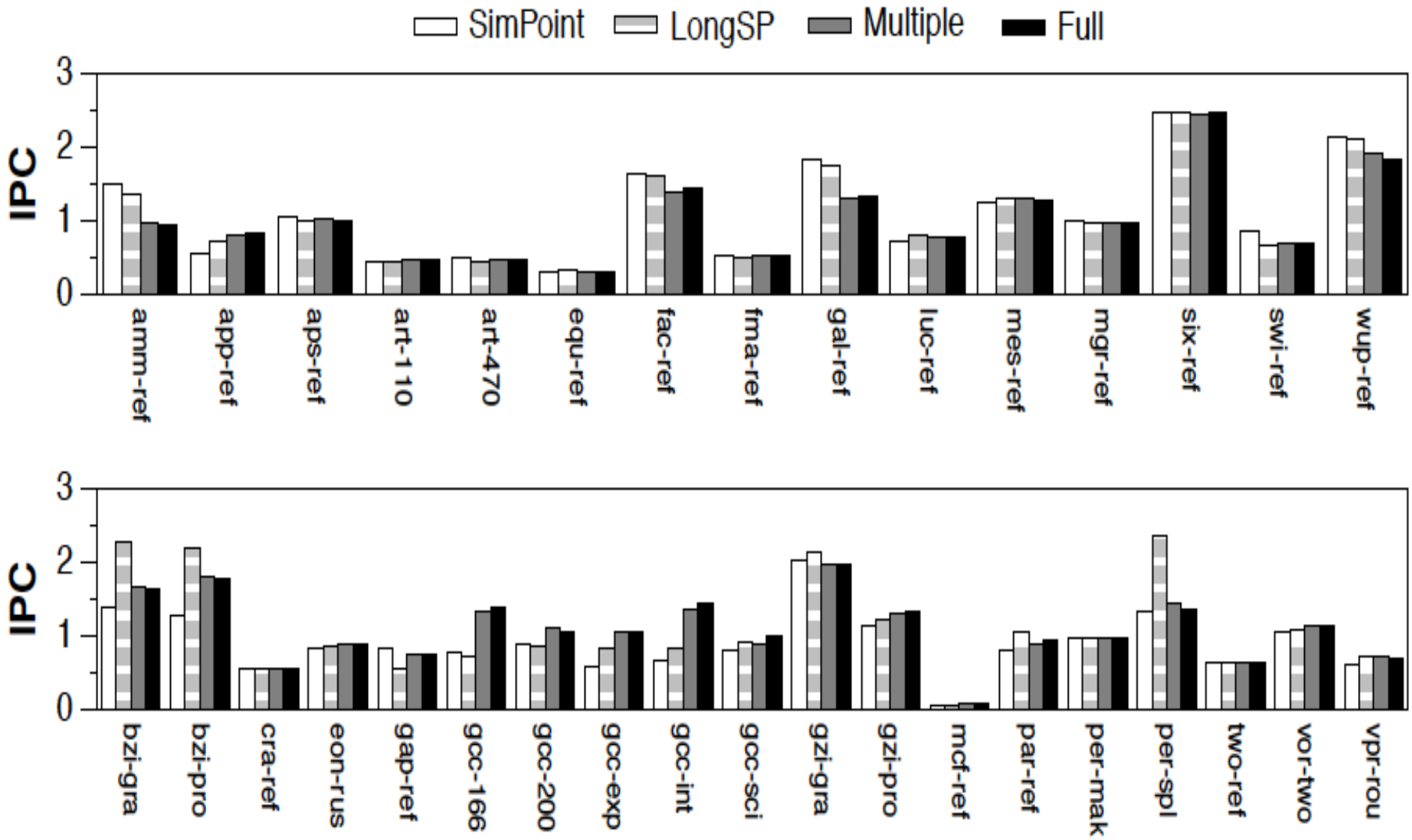


Figure 10: Multiple simulation point results. Simulation results are shown for using a single simulation point simulating for 100 million instructions, LongSP chooses a single simulation point simulating for the same length of execution as the multiple point simulation, simulation using multiple simulation points, and the full execution of the program.

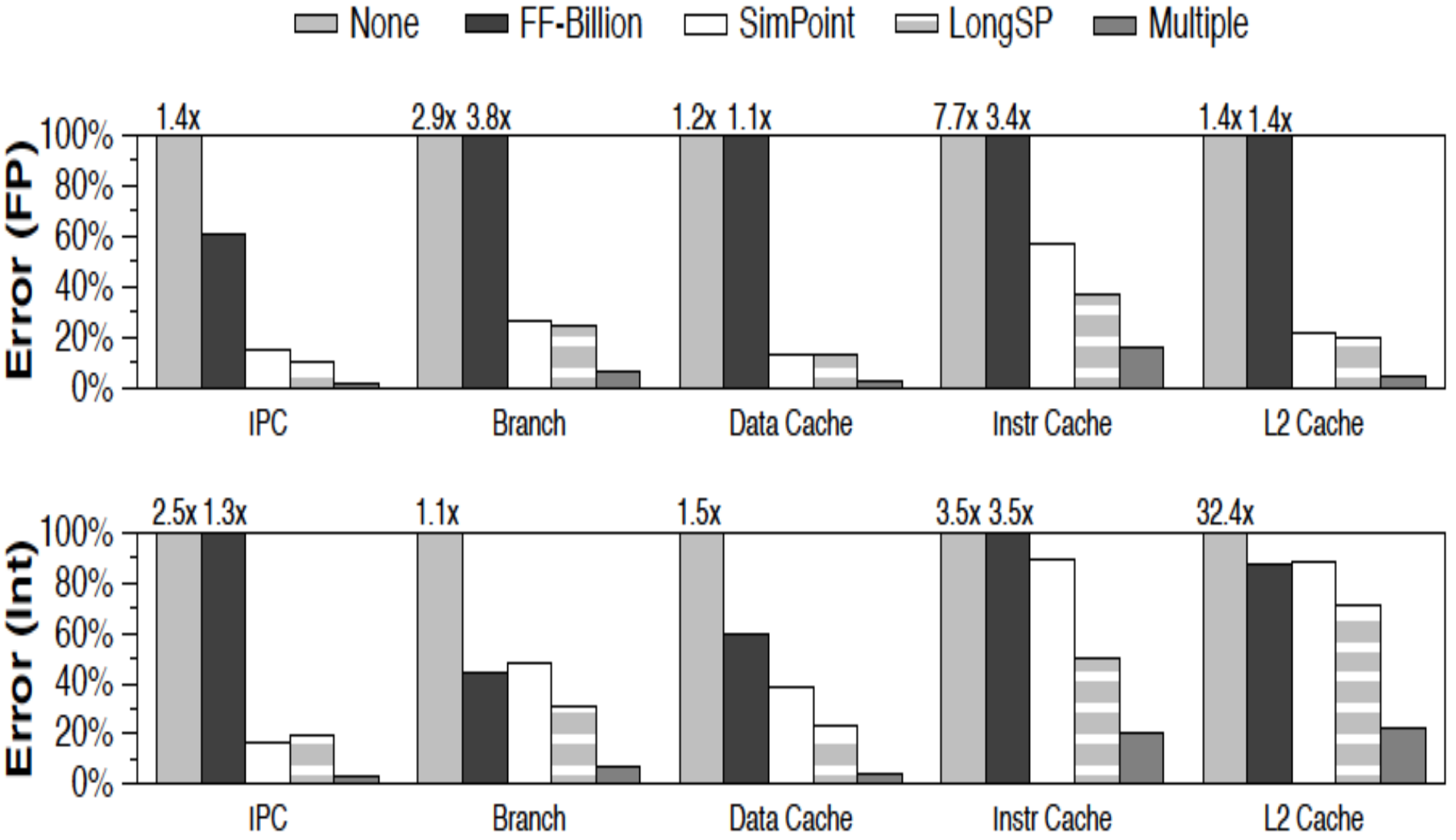


Figure 11: Average error results for the SPEC 2000 floating point (top) and integer (bottom) benchmarks for IPC, branch misprediction, instruction, data and unified L2 cache miss rates.

[18] T. Sherwood and B. Calder. Time varying behavior of programs. Technical Report UCSD-CS99-630, UC San Diego, August 1999.

[19] T. Sherwood, E. Perelman, and B. Calder. Basic block distribution analysis to find periodic behavior and simulation points in applications. In *International Conference on Parallel Architectures and Compilation Techniques*, September 2001.

[20] T. Sherwood, S. Sair, and B. Calder. Phase tracking and prediction. Technical Report CS2002-0710, UC San Diego, June 2002.

Other Work from Same authors

[18] T. Sherwood and B. Calder. Time varying behavior of programs. Technical Report UCSD-CS99-630, UC San Diego, August 1999.

[19] T. Sherwood, E. Perelman, and B. Calder. Basic block distribution analysis to find periodic behavior and simulation points in applications. In *International Conference on Parallel Architectures and Compilation Techniques*, September 2001.

[20] T. Sherwood, S. Sair, and B. Calder. Phase tracking and prediction. Technical Report CS2002-0710, UC San Diego, June 2002.

Speed of Simulators

SimpleScalar [3], one of the faster cycle-level simulators, can simulate around 400 million instructions per hour. Unfortunately many of the new SPEC 2000 programs execute for 300 billion instructions or more. At 400 million instructions per hour this will take approximately 1 month of CPU time.

CPU 2006 programs range from 300 billion to 5 trillion instructions

Available Simulation Points

1. CPU 2000 – Alpha binaries from UCSD
2. CPU 2006 – PINPOINTS tool from Intel
3. CPU 2006 – Pin Points from UT LCA (ICCD 2006 paper, Nair and John) (x86 binaries)
4. CPU 2006 – 22 Alpha binaries – K. Ganesan (SPEC workshop 2009)
5. PARSEC - ROI (Region of Interest)
6. CPU 2006 – for SIMICS - based on Ultra SPARC binaries – being generated in LCA now

Available simpoint tools

1. PINPOINTS tool from Intel (PIN based)
2. Valgrind BBV generation tool (Open source)
3. Qemu BBV generation (Open source)

4. PinPlay – to fast forward upto the simulation point

Simulation Points for SPEC CPU 2006

Arun A. Nair, Lizy K. John
Dept. of Electrical and Computer Engineering
University of Texas at Austin
Austin, TX 78712 USA
{*nair, ljohn*}*@ece.utexas.edu*

ICCD (International Conference on Computer Design)

1998

TABLE I

NUMBER OF SIMULATION POINTS, NUMBER OF SIMULATION POINTS AMOUNTING TO 90% OF TOTAL EXECUTION AND INSTRUCTION COUNT FOR SPEC CPU 2006

Benchmark	Simulation Points	90 percentile Points	Instructions (billions)
400.perlbench-splitmail	21	12	756.9
401.bzip2-combined	17	13	371.92
403.gcc-scilab	17	9	68.57
429.mcf	14	9	464.98
445.gobmk-trevord.tst	18	13	359.52
456.hmmer-retro.hmm	17	15	2472.91
458.sjeng	16	12	2654.13
459.gemsFDTD	20	12	308.88
462.libquantum	22	15	4534.27
464.h264ref-sss_encoder_main	20	14	3289.98
471.omnetpp	9	6	787.08
473.astar-rivers.cfg	8	6	961.44

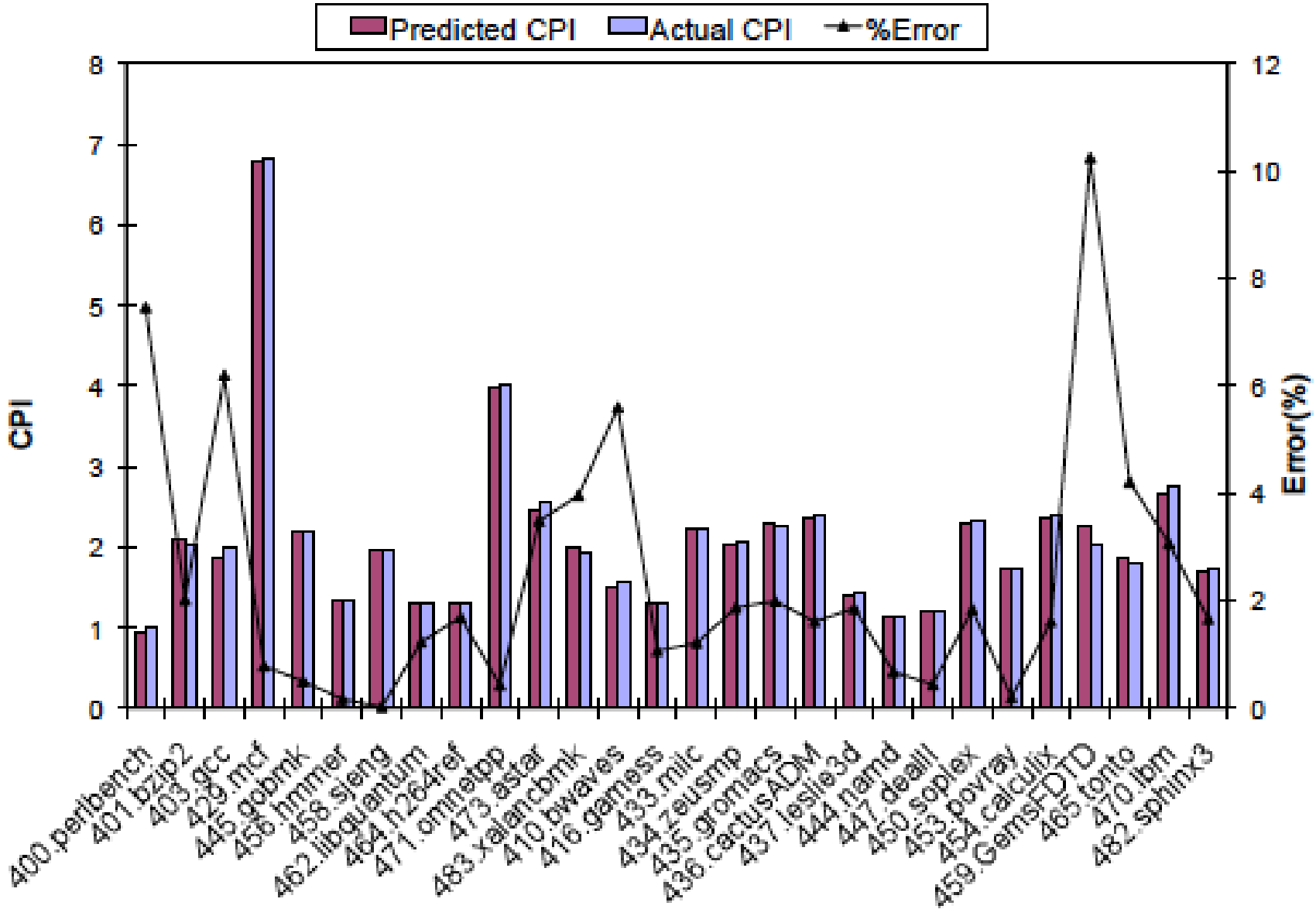
410.bwaves	22	10	2780.95
416.games-triazolium	15	11	3717.7
433.milc	23	18	1649.57
434.zeusmp	26	19	2273.56
435.gromacs	20	19	2267
436.cactusADM	21	3	3115.92
437.leslie3d	22	20	4745.74
444.namd	26	18	3293.89
447.dealII	21	14	2809.95
450.soplex-ref.mps	21	17	414.17
454.calculix	10	7	8499.78
453.povray	20	15	1287.36
465.tonto	20	15	3002.2
470.lbm	21	12	1567.55
482.sphinx	20	16	3135.75
Average	**18.75**	**13.07**	**2249.75**

TABLE II

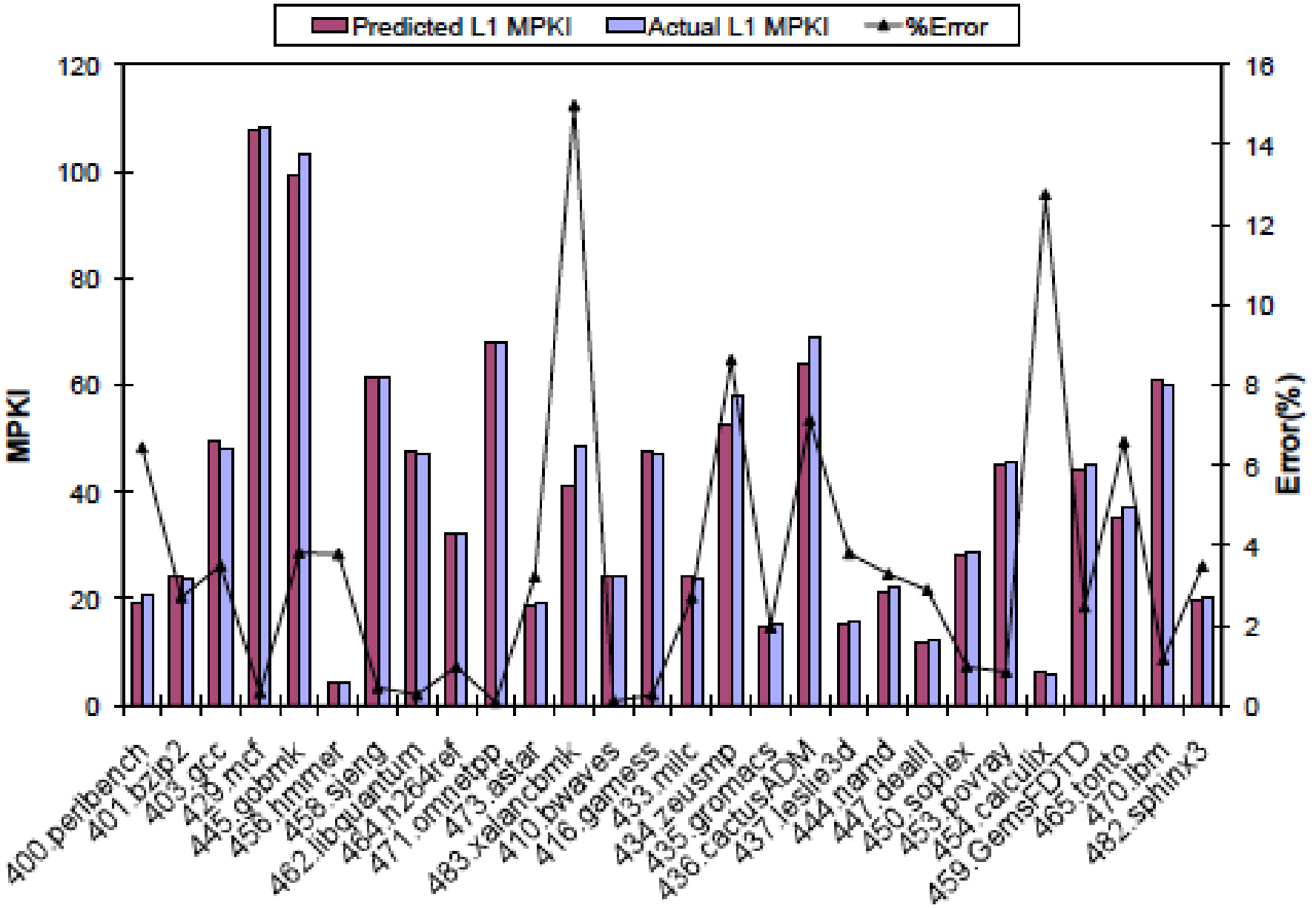
NUMBER OF SIMULATION POINTS, NUMBER OF SIMULATION POINTS AMOUNTING TO 90% OF TOTAL EXECUTION AND INSTRUCTION COUNT FOR SPEC CPU 2000

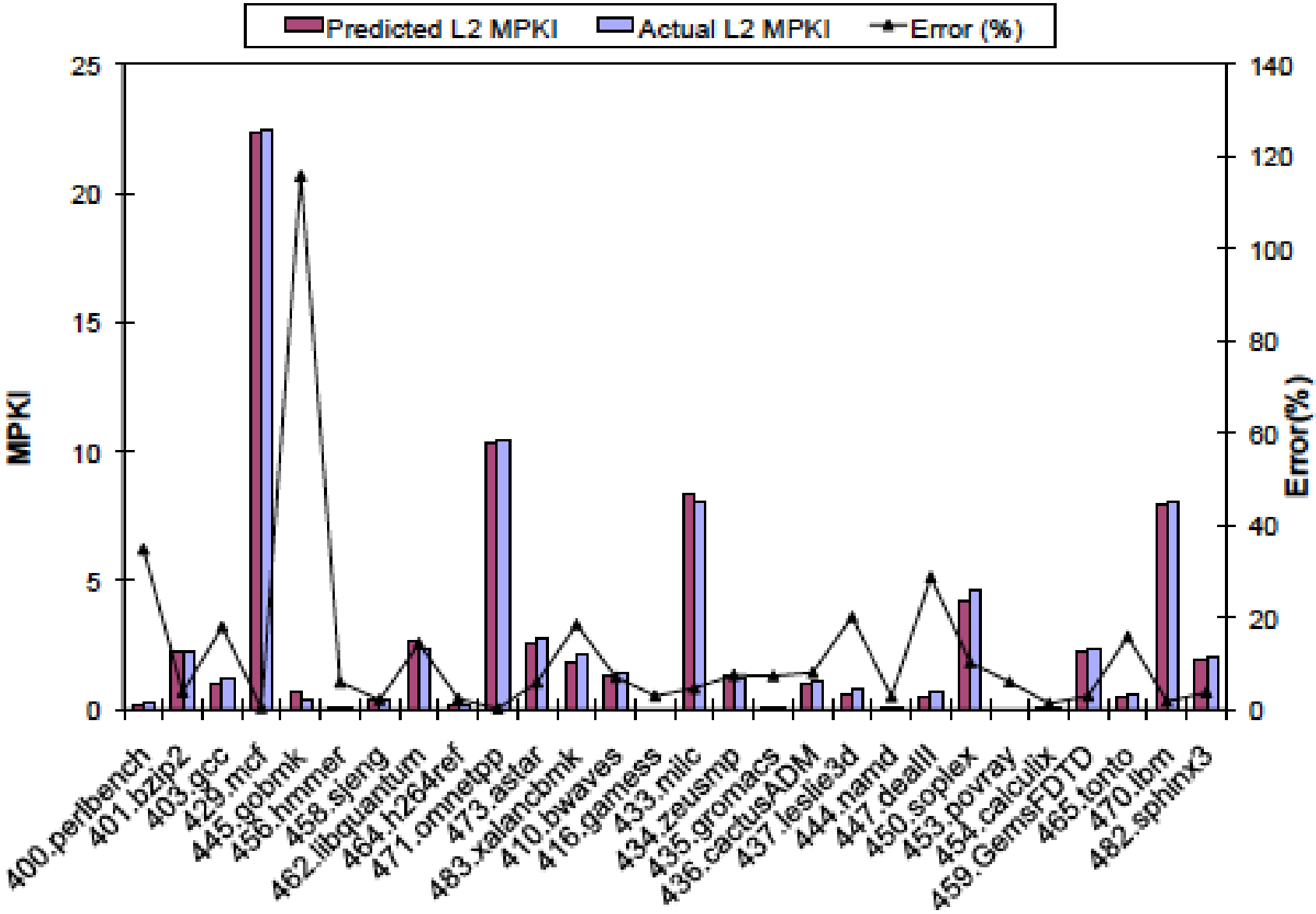
Benchmark	Simulation Points	90 percentile Points	Instructions (billions)
176.gcc-scilab	18	11	38.51
176.gcc-166	23	14	21.29
164.gzip-graphic	27	21	71.47
164.gzip-source	14	10	54.17
175.vpr-place	15	11	111.86
175.vpr-route	23	15	85.63
300.twolf	20	14	290.93
186.crafty	16	13	216.96
181.mcf	12	8	48.80
253.perlbmk	16	10	94.87
256.bzip-source	20	15	87.08
256.bzip-graphic	23	20	117.28
197.parser	13	10	281.77
254.gap	18	12	54.17
179.art-1	15	12	113.55
179.art-2	12	10	117.29

179.art-2	12	10	117.29
173.applu	25	18	528.82
188.ammp	24	13	386.60
200.sixtrack	9	5	936.54
183.equake	30	23	149.67
301.apsi	20	11	602.69
171.swim	22	20	249.89
172.mgrid	22	17	523.77
168.wupwise	23	9	490.19
177.mesa	18	13	317.34
Average	**19.12**	**13.4**	**239.65**



(a) CPI measurements for SPEC CPU 2006.


Predicted L1 MPKI
Actual L1 MPKI
%Error
MPKI
Error(%)
120
100
80
60
40
20
0
16
14
12
10
8
6
4
2
0
400.perlbench
401.bzip2
403.gcc
429.mcf
445.gobmk
458.hmmer
458.sjeng
462.libquantum
464.h264ref
471.omnetpp
473.astar
483.xalancbmk
410.bwaves
416.gamess
433.milc
434.zeusmp
435.gromacs
436.cactusADM
437.leslie3d
444.namd
447.dealII
450.soplex
453.povray
454.calculix
459.GemsFDTD
465.tonto
470.lbm
482.sphinx3



(e) L2 MPKI measurements for SPEC CPU 2006