A method for automatically generating analogue benchmark suites using low-level hardware metrics

Simon McIntosh-Smith
University of Bristol, UK
simonm@cs.bris.ac.uk

Owen Thomas
Red Oak Consulting
owen@redoakconsulting.co.uk

Introduction

- There are challenges in constructing and maintaining benchmark suites, especially when the original codes are complex or classified
  - Thus may not be shared directly with vendors or partners
- Can we ***automatically*** construct benchmarks that closely model the low-level behavior of target workloads?

Methodology – I

- Select a set of open-source benchmarks that exhibit a reasonably wide range of behavior in terms of low-level hardware metrics:
  - Cycles per instruction
  - Cache misses (instruction, data, multiple levels)
  - DRAM accesses
  - Data Translation Lookaside Buffer Misses (page faults)
  - Branch instruction rate
  - Mispredicted branches rate

- These metrics were selected as the most important in terms of their effect on the observed performance in a modern CPU architecture

Methodology – II

- Characterise target codes[1];
- Characterise potential analogue benchmarks[1];
- Create benchmark analogues;
- Assemble parameterised benchmark suite.

- [1] Multiple copies in parallel, one per core

Characterisation tools

- System Tap
  - Standard Linux tool
  - Records kernel-level stats about memory use, I/O etc
- Oprofile
  - Standard Linux tool
  - Records low-level hardware metrics (cycles per instruction, branch misprediction rate etc)
- Sampling issues have to be managed

Target benchmark characterization on Nehalem

Occurrences per cycle

0.25
0.20
0.15
0.10
0.05
0.00

toy
macduff
linalg
Bithand
sol33
test_bitperm
iotest

Branches
Mispredicted
LLCmisses
L2_LINES_IN
DTLBmisses
L1Imisses
L1Din

Nehalem hardware counters from a twelve hour target job mix sample run

Nehalem cycles per instruction

macduff	toy	linalg	Bithand	sol33	test_bitperm	iotest
1.03	7.62	0.43	0.43	1.39	0.51	0.80

9.0
8.0
7.0
6.0
5.0
4.0
3.0
2.0
1.0
0.0

Characterising targets

- Do we run them:
  - On one core at a time,
  - On all the cores at the same time, or
  - In a random mix?
- Observed the impact of running each target in parallel with itself
- Some counter sampling rate issues (too few and too many are both problems).

Target characterization on Nehalem

Runtime in seconds

400
350
300
250
200
150
100
50
0

1 2 3 4 5 6 7 8

Degree of parallelism

toy 9M
toy 6M
macduff 25 50
toy 3M
sol33
test_bitperm
Bithand 1 22
macduff 24 40
linalg 10000000 100
Bithand 1 21
Bithand 1 20

Target characterization on Magny-Cours

Runtime in seconds

1,000
800
600
400
200
0

1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24

Degree of parallelism

toy 9M
toy 6M
toy 3M
macduff 25 50
sol33
test_bitperm
Bithand 1 22
macduff 24 40
linalg 10000000 100
Bithand 1 21
Bithand 1 20

Analogue benchmarks

Analogue benchmark examples

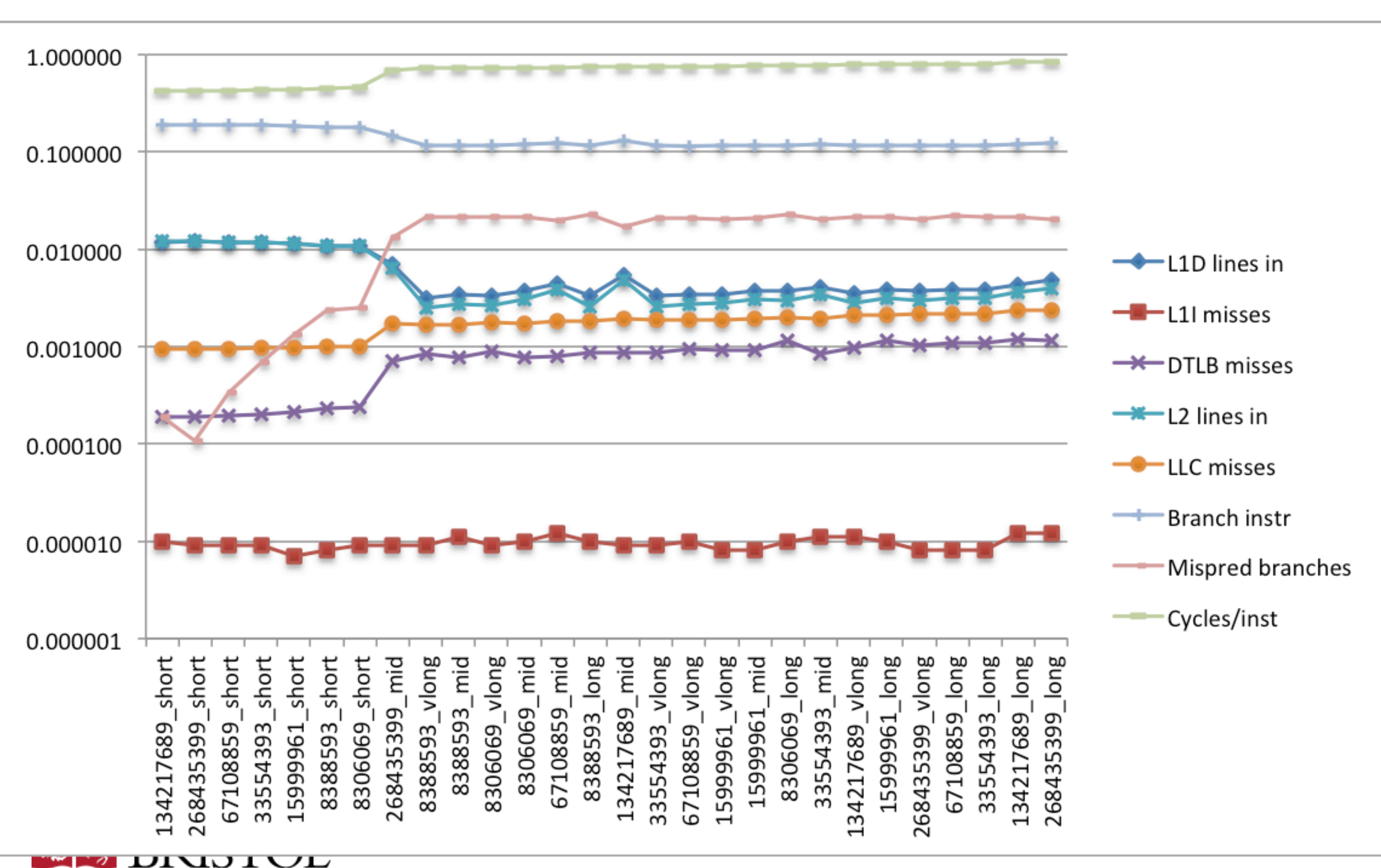
- Fhourstones
  - Something about Fhourstones
- Gups
  - Something about Gups
- Single_core
  - Something about single_core

Characterising analogues

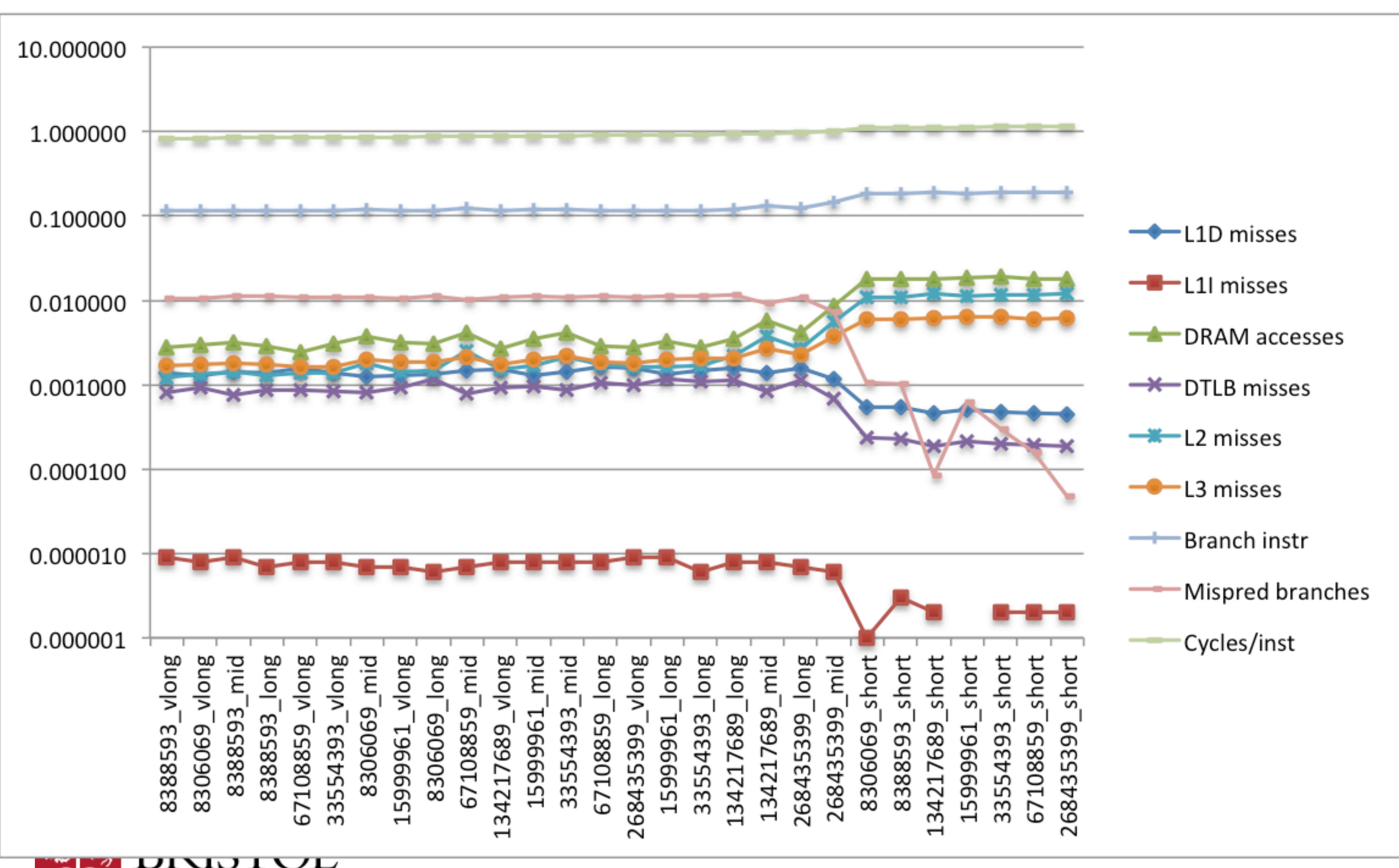
For each target platform T

- For each analogue benchmark A
  - For each parameterisation P
    - $A_T(P)$ - run A on T with parameters P
    - Record low-level hardware metrics for $A_T(P)$

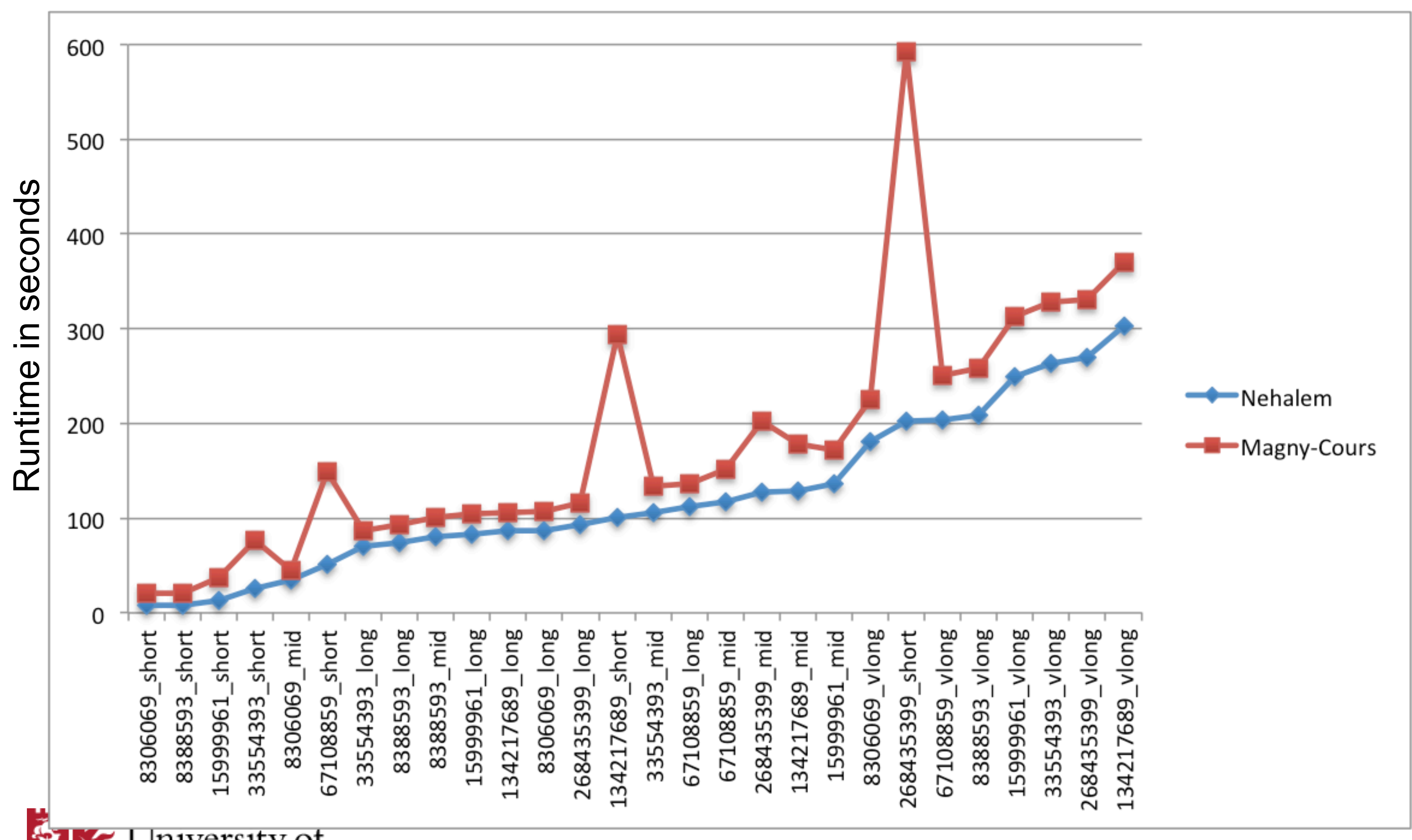
Fhourstones on Nehalem



Fhourstones on Magny-Cours



Fhourstones runtimes



University of
BRISTOL

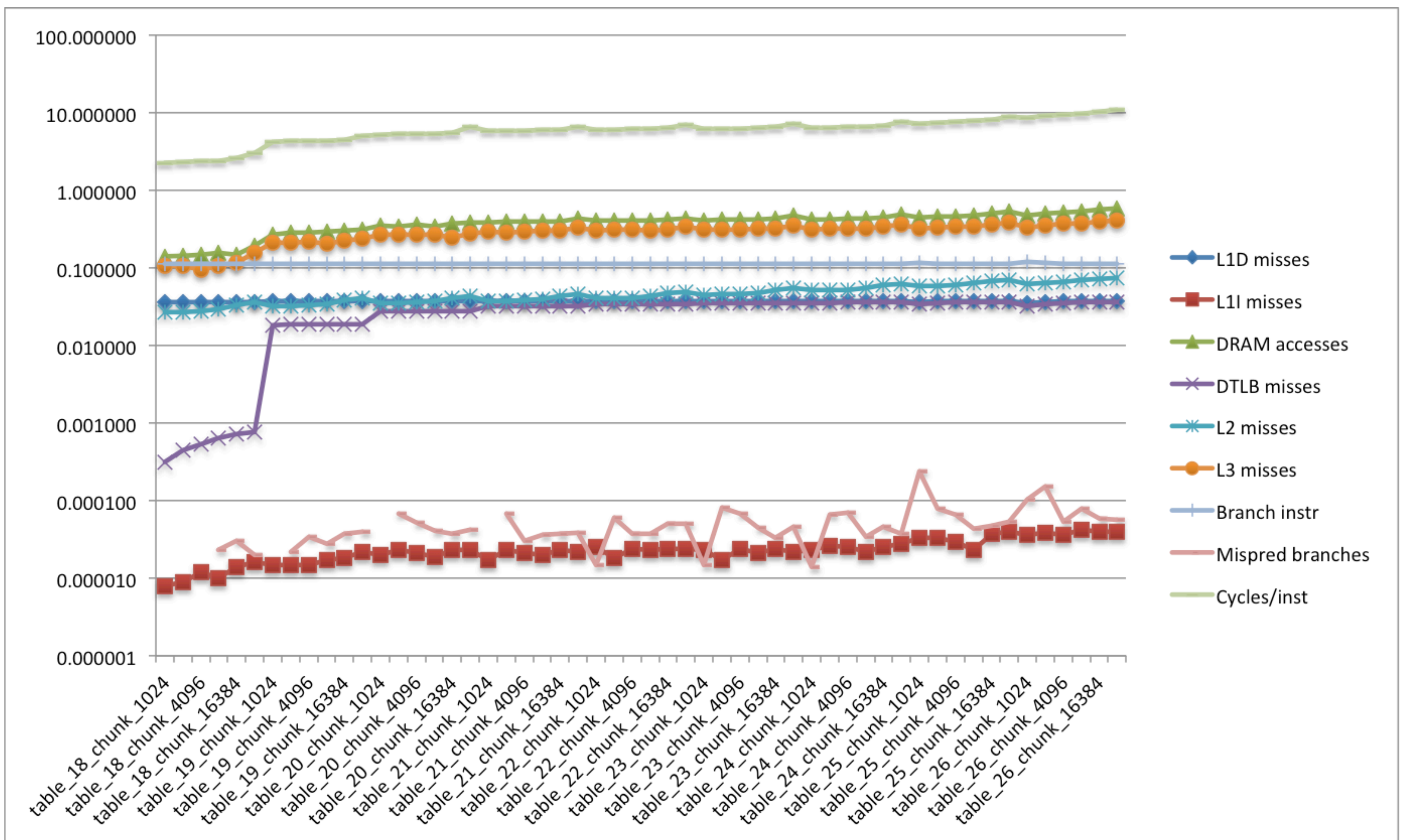
Gups on Nehalem

10.000000
1.000000
0.100000
0.010000
0.001000
0.000100
0.000010

table_18_chunk_1024
table_18_chunk_4096
table_18_chunk_16384
table_19_chunk_1024
table_19_chunk_4096
table_19_chunk_16384
table_20_chunk_1024
table_20_chunk_4096
table_20_chunk_16384
table_21_chunk_1024
table_21_chunk_4096
table_21_chunk_16384
table_22_chunk_1024
table_22_chunk_4096
table_22_chunk_16384
table_23_chunk_1024
table_23_chunk_4096
table_23_chunk_16384
table_24_chunk_1024
table_24_chunk_4096
table_24_chunk_16384
table_25_chunk_1024
table_25_chunk_4096
table_25_chunk_16384
table_26_chunk_1024
table_26_chunk_4096
table_26_chunk_16384
table_27_chunk_65536
table_28_chunk_65536

L1D lines in
L1I misses
DTLB misses
L2 lines in
LLC misses
Branch instr
Mispred branches
Cycles/inst

Gups on Magny-Cours



BRISTOL

Gups runtimes

Runtime in seconds

200
180
160
140
120
100
80
60
40
20
0

table_18_chunk_1024
table_20_chunk_1024
table_21_chunk_1024
table_22_chunk_1024
table_20_chunk_2048
table_18_chunk_4096
table_21_chunk_2048
table_22_chunk_2048
table_23_chunk_2048
table_24_chunk_2048
table_18_chunk_8192
table_26_chunk_2048
table_22_chunk_4096
table_20_chunk_8192
table_25_chunk_4096
table_26_chunk_4096
table_19_chunk_16384
table_23_chunk_8192
table_24_chunk_8192
table_18_chunk_32768
table_21_chunk_16384
table_22_chunk_16384
table_20_chunk_32768
table_25_chunk_16384
table_21_chunk_32768
table_23_chunk_32768
table_25_chunk_32768

Nehalem
Magny-Cours

Effects of parallelism on sweeps

Runtime in seconds

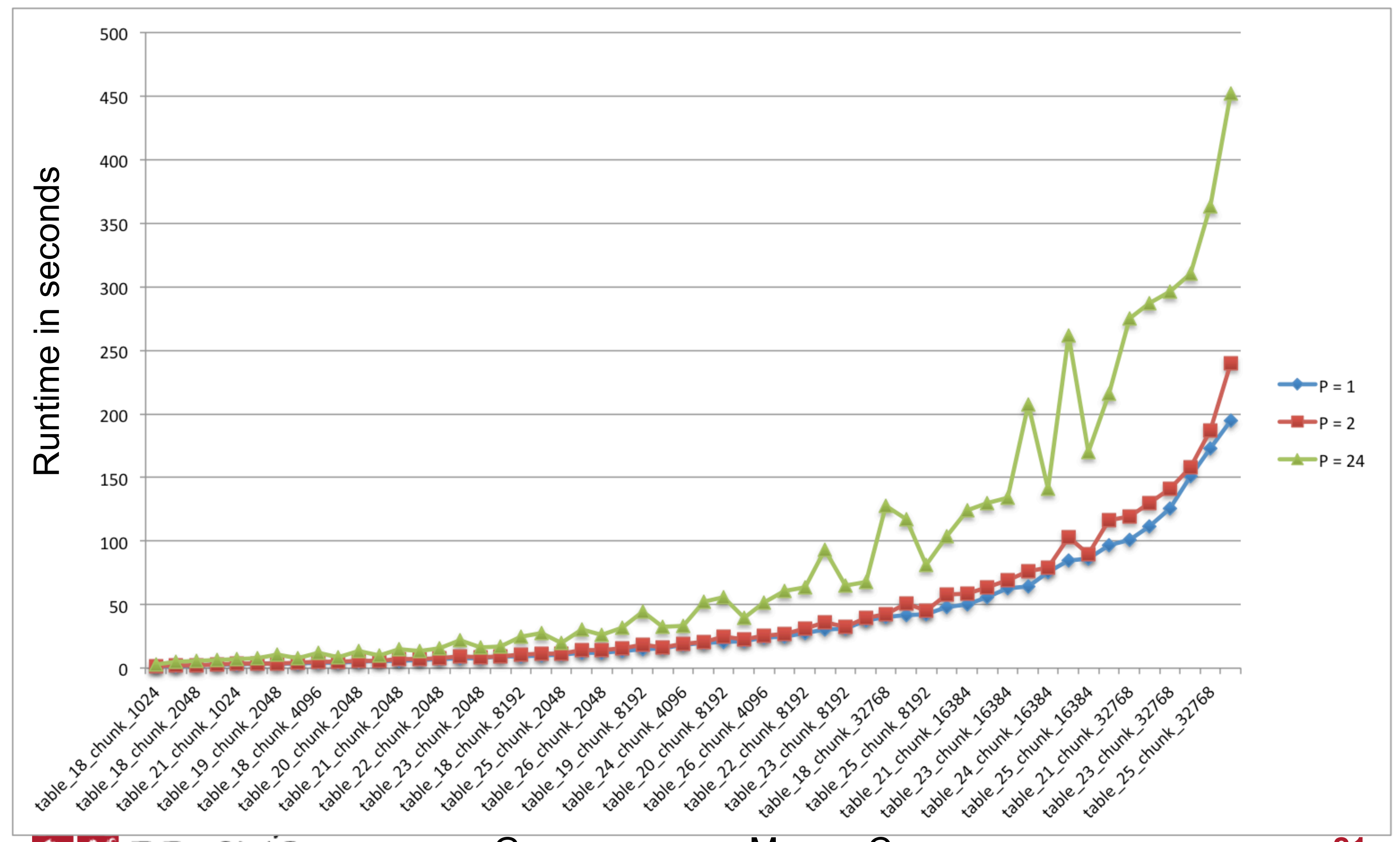
400
350
300
250
200
150
100
50
0

8306069_short
8388593_short
15999961_short
33554393_short
8306069_mid
67108859_short
33554393_long
8388593_long
8388593_mid
15999961_long
134217689_long
8306069_long
268435399_long
134217689_short
33554393_mid
67108859_long
67108859_mid
268435399_mid
134217689_mid
15999961_mid
8306069_vlong
268435399_short
67108859_vlong
8388593_vlong
15999961_vlong
33554393_vlong
268435399_vlong
134217689_vlong

P = 1
P = 2
P = 8

Fhourstones sweep on Nehalem

Effects of parallelism on sweeps



Gups sweep on Magny-Cours

Effects of parallelism on sweeps

Runtime in seconds

40
40
39
39
38
38
37
37
36
36

array_512_ncrypt_3
array_512_ncrypt_12
array_1024_ncrypt_3
array_1024_ncrypt_12
array_2048_ncrypt_3
array_2048_ncrypt_12
array_4096_ncrypt_3
array_4096_ncrypt_12
array_8192_ncrypt_3
array_8192_ncrypt_12
array_16384_ncrypt_3
array_16384_ncrypt_12
array_32768_ncrypt_3
array_32768_ncrypt_12
array_65536_ncrypt_3
array_65536_ncrypt_12
array_131072_ncrypt_3
array_131072_ncrypt_12
array_262144_ncrypt_3
array_262144_ncrypt_12
array_524288_ncrypt_3
array_524288_ncrypt_12
array_1048576_ncrypt_3
array_1048576_ncrypt_12
array_2097152_ncrypt_3
array_2097152_ncrypt_12
array_4194304_ncrypt_3
array_4194304_ncrypt_12
array_8388608_ncrypt_3
array_8388608_ncrypt_12
array_16777216_ncrypt_3
array_16777216_ncrypt_12

P = 1
P = 2
P = 24

Single_core sweep on Magny-Cours

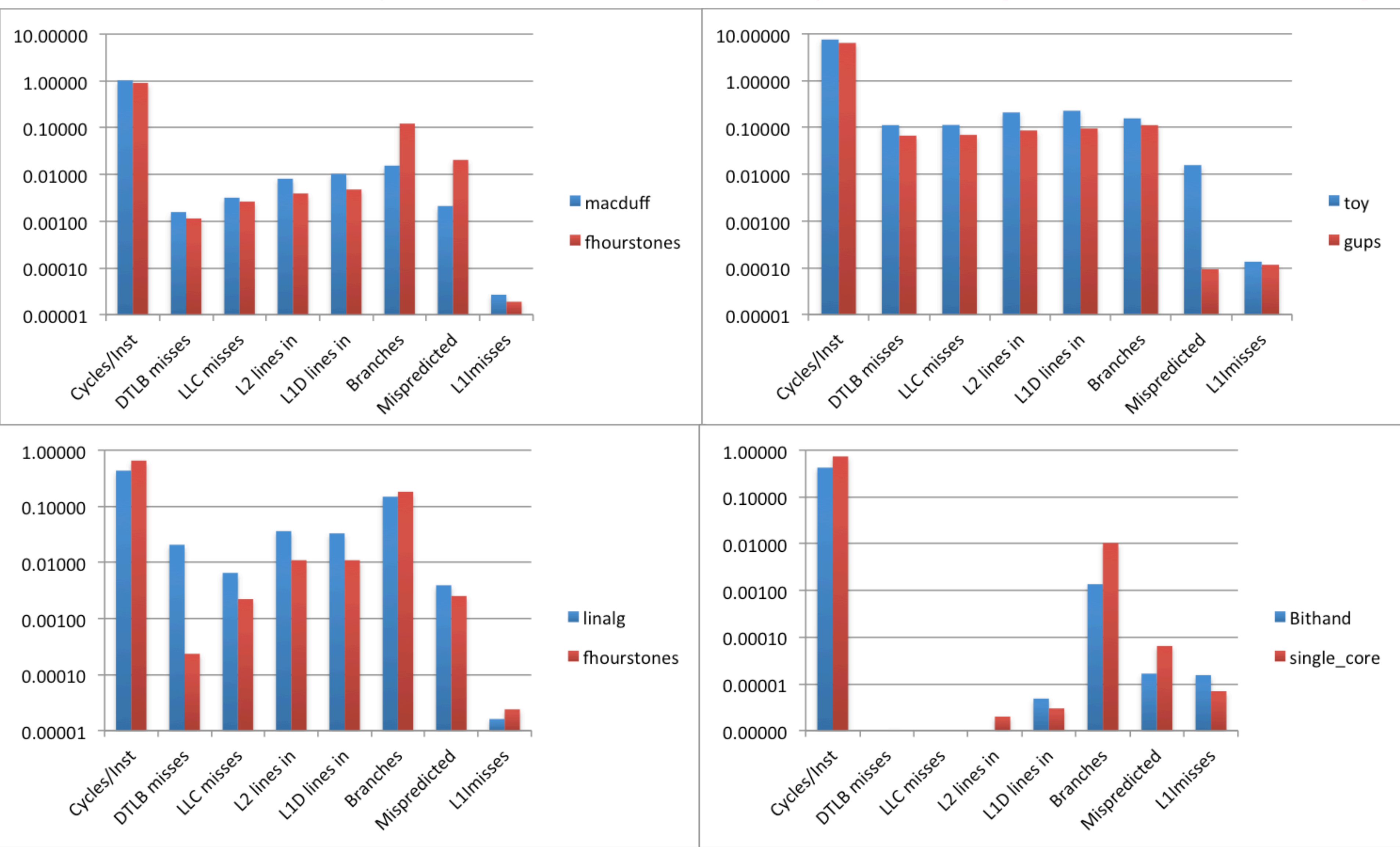
Assembling analogue benchmark suites

Exploration of analogue benchmark suite methodologies

There are a number of different ways we can construct an analogue job mix that models a target job mix:

1. Manually select one analogue to represent one target
2. Automatic point-wise construction of a composite analogue job mix
3. Residual refinement of a composite analogue job mix

Method 1: hand selection of analogues for targets (Nehalem)



Method 1 results

Percentage of total run-time

	Nehalem target jobmix	Nehalem analogue jobmix	Magny-Cours target jobmix	Magny-Cours analogue jobmix
bithand	2.3	2.5	1.8	1.4
linalg	5.0	4.7	4.6	5.2
toy+sol	9.3	7.8	13.3	9.4
macduff	72.3	81.2	72.8	77.7

The derived benchmark suite was assessed by comparing the breakdown of the percentage of CPU time for each analogue over a nine hour extended run of the analogue benchmark job mix.

Method 1 results summary

- Across the whole target and analogue job mixes, the differences in the runtimes represent an 8.2% overestimate on Nehalem and a 1.3% overestimate on Magny-Cours.
- In summary, the parameterization was successful, delivering analogue benchmark job mixes that estimate performance to within 10% overall of the target benchmark job mix.

Method 2

Automatic point-wise construction of a composite analogue job mix

- In this approach we took an aggregate view of the target job mix, using the Oprofile metrics averaged across all the target benchmarks across all the platforms. This gives a set of thirty metrics which describe the aggregate ‘shape’ of the target job mix across all platforms.
- We then took the sweep dataset previously constructed by running each analogue benchmark with a range of different input parameters across all the platforms, resulting in 141 data points, each with the same thirty metrics as the aggregate for the target job mix.
- An analogue job mix was then constructed by creating a pool of 100 job slots which were incrementally tested against all of the 141 analogues to determine whether a better least-squares fit would result from the replacement of the previous job by the new one. An iteration involved testing each of the job slots on turn.
- The stopping criteria were that are either the number of iterations exceeded 1000 or the difference in the least squares error between iterations falls below 0.01.

Method 2 results

Least-squares error

Iteration	1	2	3	4	5	6	7
Least-squares error	23.52	11.93	11.61	11.56	11.49	11.42	11.39

The analogue job mix constructed using this algorithm converged quickly, taking just seven passes to meet the stopping criteria. The convergence of the least-squares error is shown above.

Method 2 results (Nehalem)

10.00000
1.00000
0.10000
0.01000
0.00100
0.00010
0.00001

Test1 analogue jobmix average
Target jobmix average

Cycles/Inst
DTLB misses
LLC misses
L2 lines in
L1D lines in
Branches
Mispredicted
L1I misses

Method 2 results (Nehalem)

Cycles/Inst	DTLB misses	LLC misses	L2 lines in	L1D lines in	Branches	Mispredicted	L1I misses
3.2%	3.2%	3.2%	3.2%	10.9%	-26.0%	98.6%	39.6%

100%
80%
60%
40%
20%
0%
-20%
-40%

Method 2 results (Nehalem)

- The method 2 analogue job mix has been constructed in a way that makes it behave very closely to that of the target job mix, as measured by the Oprofile hardware metrics.
- All of the most important metrics show errors of less than 4% between the target and analogue job mixes.
- Larger errors are limited to the less important metrics that have lower sampling rates.
- With this approach the performance across the whole analogue job mix is indicated by the cycles per instruction metric.

Method 3

Residual refinement of a composite analogue job mix

- An aggregate view of the target job mix is taken using the Oprofile metrics summarised across all the target benchmarks and across all the platforms.
- We then attempt to iteratively build a representative workload using as our measure of fitness the least squares difference between the average metric scores of the new job mix with those of the target job mix.
- This method differs from method 2 in that job slots are filled incrementally. Thus the algorithm starts from an empty job mix and incrementally looks for analogues which reduce the residual error.
- 10.3.3. The error for the first 100 slots is show in Figure 44 below. The x axis represents the job slot.

Method 3 results

The least squares difference rapidly converges to around 11.5 and by the time we have used 5000 job slots this has only dropped to 11.4. Thus by the time we have around 100 jobs we have a good analogue job mix that should closely match the behavior of the target job mix.

Method 3 results (Nehalem)

10.00000
1.00000
0.10000
0.01000
0.00100
0.00010
0.00001

Test2 analogue jobmix average
Target jobmix average

Cycles/Inst | DTLB misses | LLC misses | L2 lines in | L1D lines in | Branches | Mispredicted | L1I misses

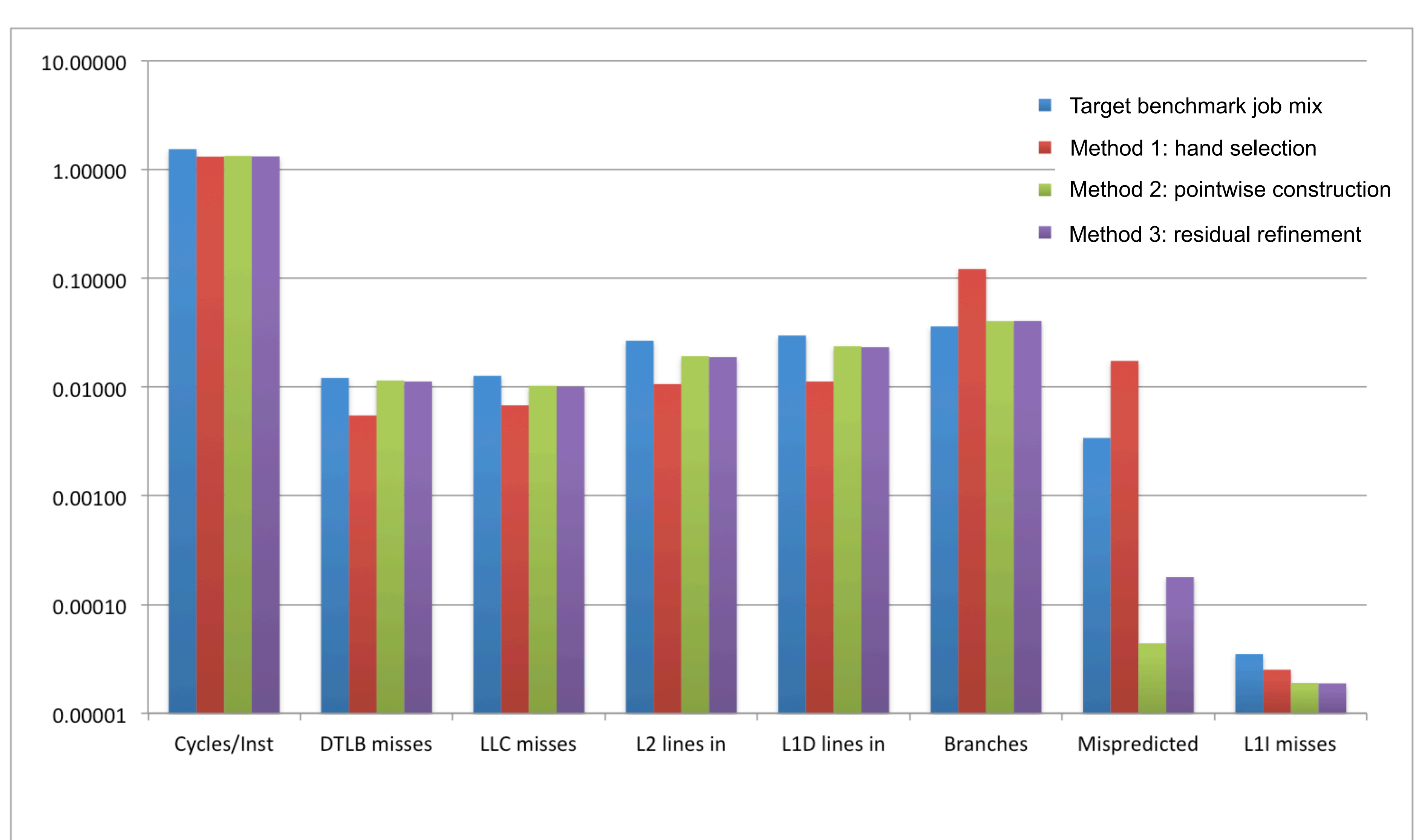
Method 3 results (Nehalem)

Cycles/Inst	DTLB misses	LLC misses	L2 lines in	L1D lines in	Branches	Mispredicted	L1I misses
4.2%	-3.4%	11.2%	20.9%	12.7%	-26.3%	94.1%	39.9%

Method 3 results (Nehalem)

- One can see from these results that this new analogue job mix also behaves in a manner very close to that of the target job mix, as measured by the Oprofile metrics.
- The first two most important metrics show errors of less than 5% between the target and analogue job mixes.
- Again, less important metrics with lower sampling rates show larger errors.
- Method 3 did not prove as accurate as method 2, with larger errors in general.
- As with the previous new approach, the performance across the whole analogue job mix is indicated by the cycles per instruction metric.

Comparison of the three methods



Conclusions

- The results show that these approaches are useful, producing analogue benchmark job mixes that are within 10-15% of the runtime of the target benchmark job mix across a range of different platforms.
- We found that the optimal mix of analogue benchmarks varied by platform, suggesting caution is needed when interpreting the results when using the analogue benchmark job mixes.
- Results also showed that we lacked an analogue that exhibited significant branch mispredictions compared to the target benchmarks
- These techniques can be useful in abstracting away from classified codes.