ECE 571 – Advanced Microprocessor-Based Design Lecture 11

Vince Weaver

http://web.eece.maine.edu/~vweaver

vincent.weaver@maine.edu

8 October 2019

Announcements

- HW#5 was posted, Caches
- Useful reading: "When prefetching works, when it doesn't and why" paper by Lee et al.

HW#4 (brpred) review – Measurements

- Bzip2 on Haswell-EP
  instr=19,2001M, branches=2,852M,conditional=2,561M
  branch:instr 15% (1:6),conditional 13.6% (1:7)%
  If way too low, entering command wrong (no file error low counts)
- equake_l on Haswell-EP
  instr=1,744B,branches=208B,conditional=178B
  branch:instr 12% (1:8), conditional 10% (1:10)
- Branch miss rate Haswell-EP bzip2 = 7.28%, equake_l

= 0.31%

- Speculative execution Haswell bzip2: roughly 74% retired, equake_l: roughly 54% retired
- AMD EPYC branch miss rate: bzip2 = 7.2% equake_l = 0.37%
- ARM64 bzip2 branch ratio:
  instr=20,141M, branches=3,344M, 17% (1:6)
  note on ARM does predicted include only conditional?
- ARM64 branch miss rate: bzip2=7.8%

HW#4 (brpred) review – Questions

(a) Why BR% ratio differ?

Compiler being stupid? These are SPEC benchmarks so you can bet that these benchmarks are being optimized as completely as possible.

Floating point vs Integer code. Floating point, like equake, tends to have lots of regular loops over big blocks of calculations. Integer code like compilers and compression reads user data and makes decisions, so many more if/then loops on irregular data.

How could you determine the cause?

(b) BR ratio on bzip Haswell/arm64?
actually about the same considering arm64 is more typically RISC and x86 CISC. ARM32 is 1:16 (why? probably conditional execution. How could you tell?)

(c) Miss rate differ? FP vs Int program.
Loops easier to predict.

(d) Different branch predictors.

(e) Pi worse (17.6%) Lower end CPU?

Note, ARM64 *completely* different than ARM32
Smaller structures? 32-bit code? Fewer branches (Conditional execution) so ones left are harder? Cortex-A53
What could we do? Run the 32-bit version on Jetson and see?

(f) Retired instructions. bzip2: roughly 74% retired, equake_l: roughly 54% retired So in theory bzip2 is more power efficient

(g) 50% benchmark. See below

HW#4 brpred hardware

- Cortex A-53
  - single entry Branch Target Instruction Cache (BTIC)
  - 256-entry Branch Target Address Cache (BTAC) to predict the target address of indirect branches.
  - The branch predictor is global, uses branch history registers, a 3072-entry pattern history prediction table
  - 8-entry return stack to accelerate returns from procedure calls
- Cortex-A57

- 2-level dynamic predictor with Branch Target Buffer (BTB)
- Static branch predictor.
- Indirect predictor.
- Return stack.

- Haswell
  - It's a secret (even Agner Fogg doesn't know)

HW#4 Writing a program to give 50%

- What kind of benchmark?
- Random number generation.
- How many branches in random()? Divide-based pseudo-number gen?
- Results below on ivybridge

```
branch-mul (pseudo-random, 2 branches per loop)
5000138 4999862
        20,123,251      branches                  #  228.926 M/sec
         5,004,391      branch-misses             #   24.87% of all branches
branch-rand (rand(), 17 branches per loop)
170,143,753      branches                  #  726.443 M/sec
        10,358,447      branch-misses             #    6.09% of all branches
branch-random (random(), 15 branches per loop)
      150,139,161      branches                  #  719.563 M/sec
        10,622,205      branch-misses             #    7.07% of all branches
```

Prefetching

- Cold misses can be common.
- Try to avoid cache misses by bringing values into the cache before they are needed.
- Caches with large blocksize already bring in extra data in advance, but can we do more?

Prefetching Concerns

- When?
  We want to bring in data before we need it, but not too early or it wastes space in the cache.

- Where? What part of cache? Dedicated buffer?

Limits of Prefetching

- May kick data out of cache that is useful
- Costs energy, especially if we do not use the data

Implementation Issues

- Which cache level to bring into? (register, L1, L2)
- Faulting, what happens if invalid address
- Non-catchable areas (MTRR, PAT).
  Bad to prefetch mem-mapped registers!

Software Prefetching

- ARM has PLD instruction
- PREFETCHW for write (3dnow, Alpha) cache protocol
- Prefetch, evict next (make it LRU) Alpha
- Prefetch a stream (Altivec)
- Prefetch0, 1, 2 to all cache levels (x86 SSE)
  Prefecthnta, non-temporal

Hardware Prefetching – icache

- Bring in two cache lines
- Branch predictor can provide hints, targets
- Bring in both targets of a branch

Hardware Prefetching – dcache

- Bring in next line – on miss bring in N and N+1 (or more?)

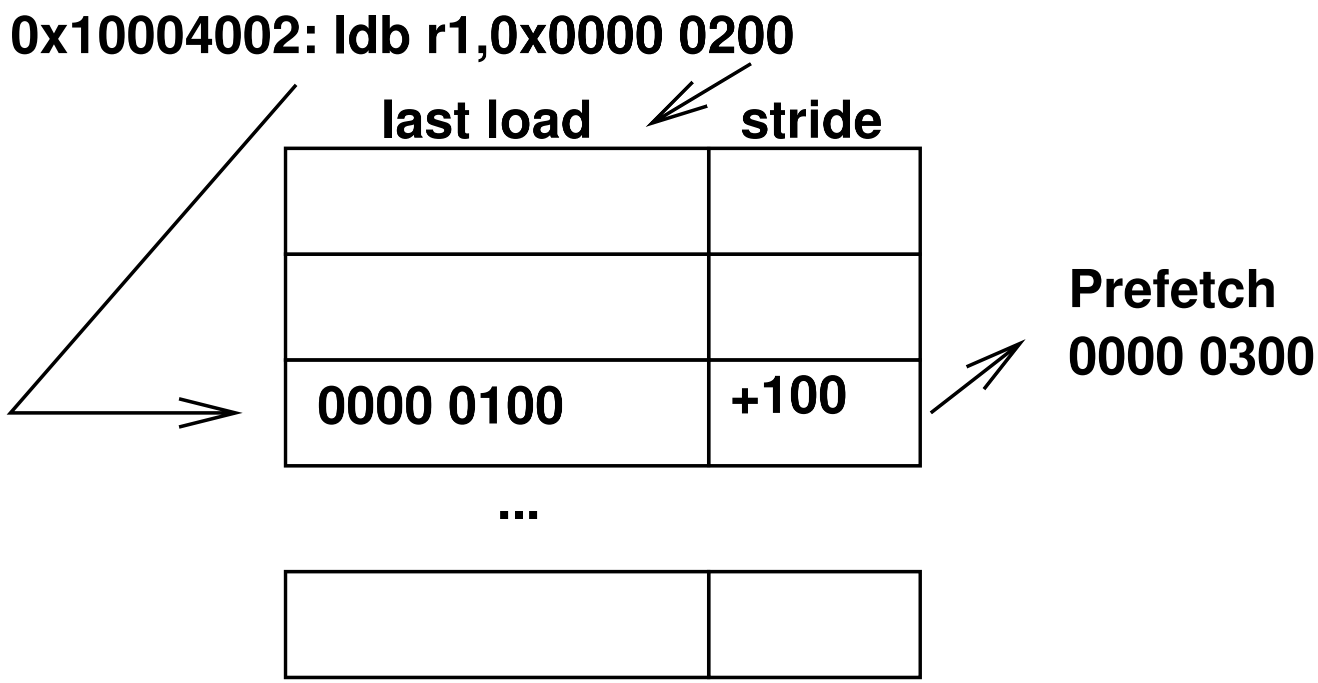
- Demand – bring in on miss (every other access a miss with linear access)
  Tagged – bring in N+1 on first access to cache line (no misses with linear access)

Hardware Prefetching – Stride Prefetching

- Stride predictors – like branch predictor, but with load addresses, keep track of stride

- Separate stream buffer?

Stride Predictor



Hardware Prefetching – Correlation/Content-Directed Prefetching

- How to handle things like pointer chasing / linked lists?
- Correlation – records sequence of misses, then when traversing again prefetches in that order
- Content directed – recognize pointers and pre-fetch what they point to

Using 2-bit Counters

- Use 2-bit counter to see if load causing lots of misses, if so automatically treat as streaming load (Rivers)
- Partitioned cache: cache stack, heap, etc, (or little big huge) separately (Lee and Tyson)

SW Prefetch notes from paper

- *When Prefetching Works, When it Doesn't, and Why* by Lee, Kim, and Vuduc (ACM TACO 2012)
- Experiment on some SPEC CPU 2006 benchmarks, some helped, some hurt, some same
- Times SW Prefetch works well
  - Large number of streams (more than available tables)
  - Short streams (takes while to train up HW prefetch)
  - Irregular memory access
  - Hint to bring into L1 (HW often only prefetches to

L3)

  - Loop bounds, SW less likely to go off end of arrays at end of loops
- Times SW works poorly
  - Increases instruction count (both insns, but also a sw prefetch might have extra calcs to construct address)
  - Static behavior, cannot adapt to phase behavior
  - Code changes might be needed (unrolling, etc) to give more calculations between loads
- SW and HW might be antagonistic
  - SW might predict all easy prefetches, leaving HW with

tougher ones and less to learn from

Cortex A9 Prefetch

- PLD – prefetch instruction
  has dedicated instruction unit
- Optional hardware prefetcher. (Disabled on pandaboard)
- Can prefetch 8 data streams, detects ascending and descending with stride of up to 8 cache lines
- Keeps prefetching as long as causing hits
- Stops if: crosses a 4kB page boundary, changes context,

a DSB (barrier) or a PLD instruction executes, or the program does not hit in the prefetched lines.

- PLD requests always take precedence

Quick Look at Haswell Prefetch

- https://software.intel.com/en-us/articles/disclosure-of-hw-prefetcher-control-on-some-intel-proce
- 4 prefetches, can independently disable
- L2 hardware prefetcher – fetch data or code into L2
- L2 adjacent cache line prefetcher – bring in 2 cache lines (128B)
- DCU prefetcher – fetch into L1-D cache
- DCU IP prefetcher – use load history to predict what to bring in

Investigating Prefetching Using Hardware Performance Counters

These are notes from some experiments I ran around the 2010 time frame

Quick Look at Core2 Prefetch

- Instruction prefetcher
- L1 Data Cache Unit Prefetcher (streaming).
  Ascending data accesses prefetch next line
- L1 Instruction Pointer Strided Prefetcher.
  Looks for strided access from particular load instructions.
  Forward or Backward up to 2k apart
- L2 Data Prefetch Logic.
  Fetches to L2 based on the L1 DCU

x86 SW Prefetch Instructions (AMD)

- `PREFETCHNTA` – SSE1, non temporal (use once)
- `PREFETCHT0` – SSE1, prefetch to all levels
- `PREFETCHT1` – SSE1, prefetch to L2 + higher
- `PREFETCHT2` – SSE1, prefetch to L3 + higher
- `PREFETCH` – AMD 3DNOW! prefetch to L1
- `PREFETCHW` – AMD 3DNOW! prefetch for write

Core2

- SSE_PRE_EXEC:NTA – counts NTA
- SSE_PRE_EXEC:L1 – counts T0 (fxsave+2, fxrstor+5)
- SSE_PRE_EXEC:L2 – counts T1/T2
- Problem: Only 2 counters available on Core2

AMD (Istanbul and Later)

- `PREFETCH_INSTRUCTIONS_DISPATCHED:NTA`
- `PREFETCH_INSTRUCTIONS_DISPATCHED:LOAD`
- `PREFETCH_INSTRUCTIONS_DISPATCHED:STORE`
- These events appear to be speculative, and won't count SW prefetches that conflict with HW prefetches

Atom

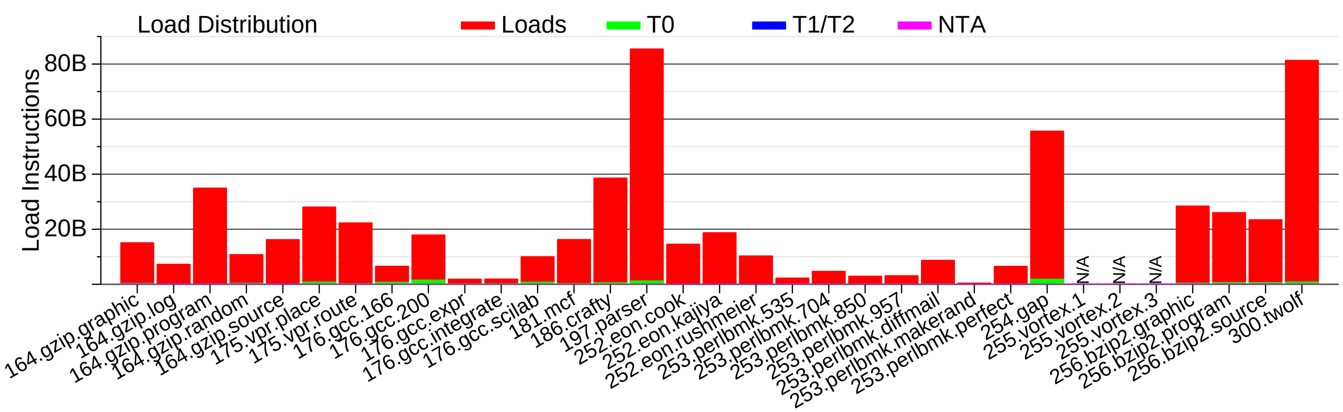
- PREFETCH:PREFETCHNTA
- PREFETCH:PREFETCHT0
- PREFETCH:SW_L2
- These events will count SW prefetches, but numbers counted vary in complex ways

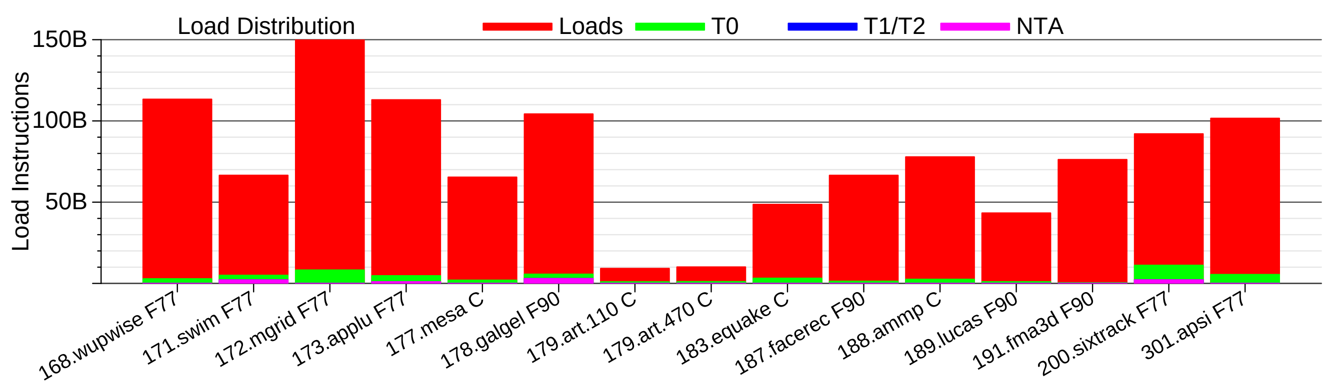
Does anyone use SW Prefetch?

- gcc by default disables SW prefetch unless you specify `-fprefetch-loop-arrays`
- icc disables unless you specify `-xsse4.2 -op-prefetch=4`
- glibc has hand-coded SW prefetch in `memcpy()`
- Prefetch can hurt behavior:
  - Can throw out good cache lines,
  - Can bring lines in too soon,
  - Can interfere with the HW prefetcher

SW Prefetch Distribution

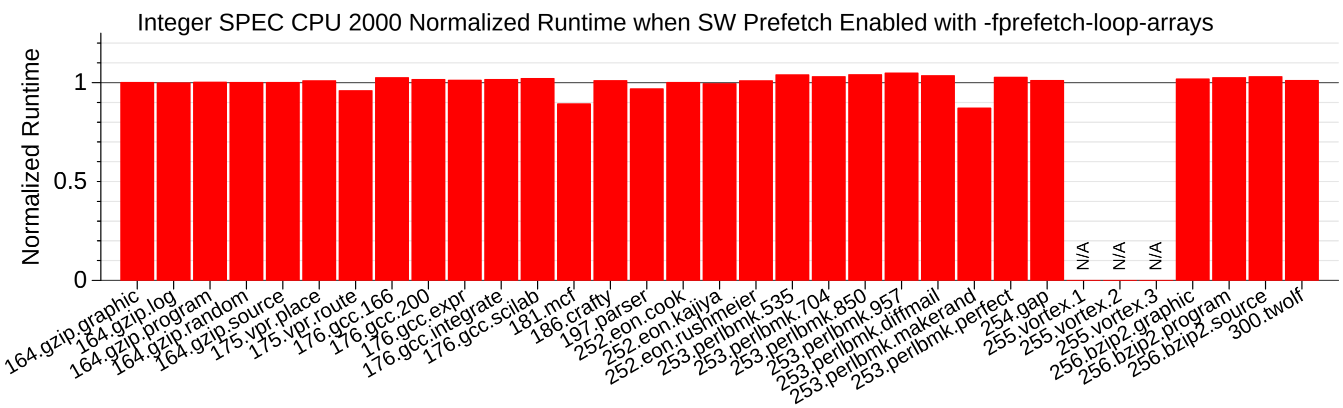
SPEC CPU 2000, Core2, gcc `-fprefetch-loop-arrays`

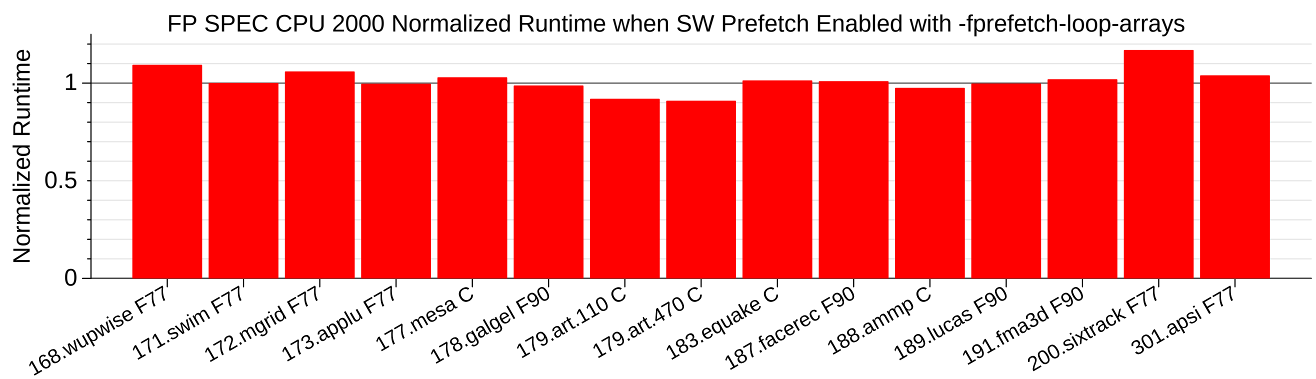




Normalized SW Prefetch Runtime

on Core2 (Smaller is Better)

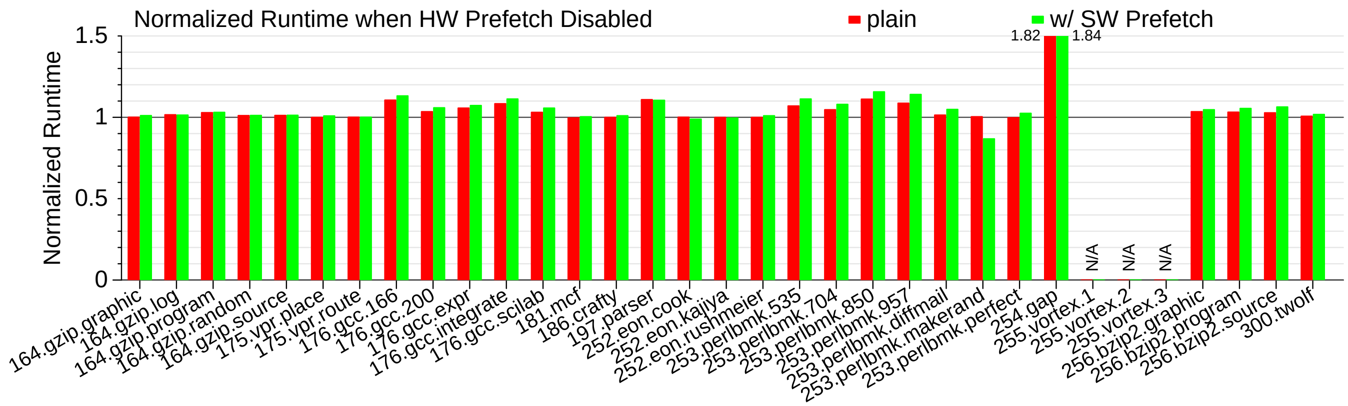


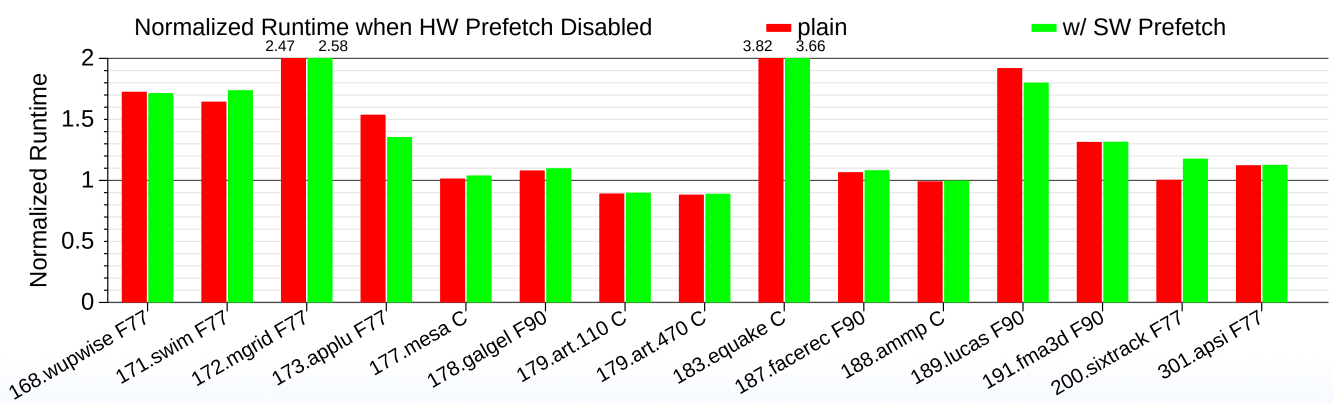


The HW Prefetcher on Core2 can be Disabled

Runtime with HW Prefetcher Disabled

Normalized against Runtime with HW Prefetcher Enabled
on Core2 (Smaller is Better)





PAPI_PRF_SW Revisited

- Can multiple machines count SW Prefetches?
  Yes.
- Does the behavior of the events match expectations?
  Not always.
- Would people use the preset?
  Maybe.

L1 Data Cache Accesses

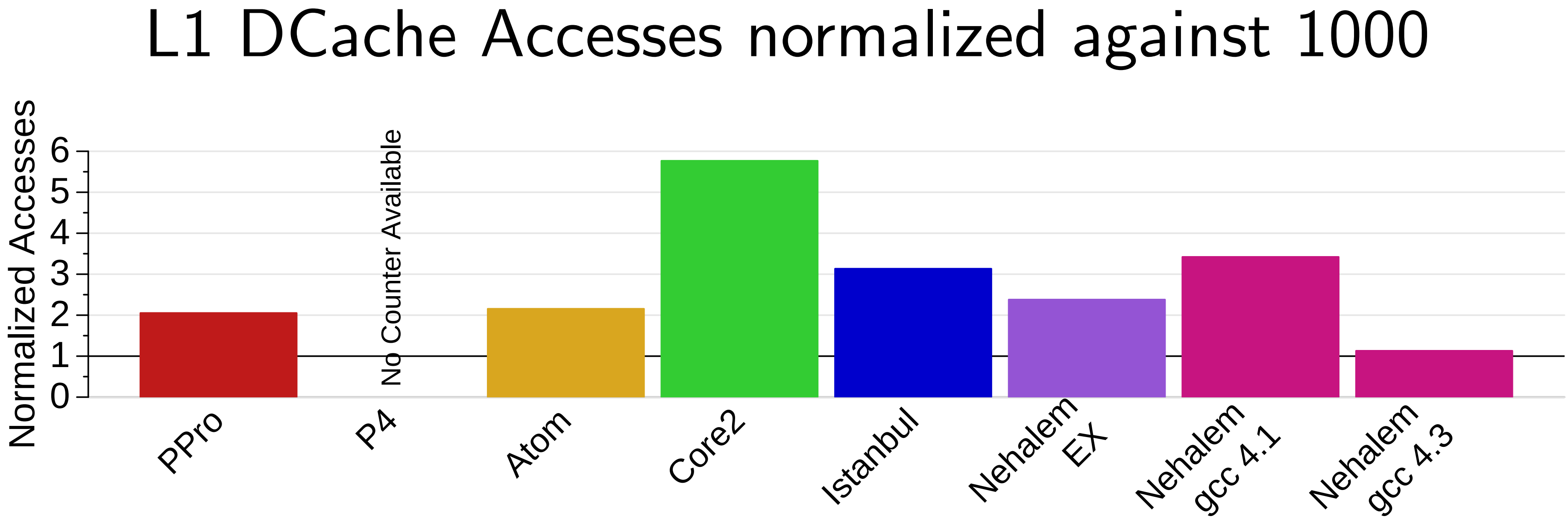
```
float array[1000],sum = 0.0;

PAPI_start_counters(events,1);

for(int i=0; i<1000; i++) {
   sum += array[i];
}

PAPI_stop_counters(counts,1);
```

PAPI_L1_DCA



L1 DCache Accesses normalized against 1000
Normalized Accesses
0
1
2
3
4
5
6
No Counter Available
PPro
P4
Atom
Core2
Istanbul
Nehalem EX
Nehalem gcc 4.1
Nehalem gcc 4.3

PAPI_L1_DCA

Expected Code

```
* 4020d8:      f3 0f 58 00              addss  (%rax),%xmm0
  4020dc:      48 83 c0 04              add    $0x4,%rax
  4020e0:      48 39 d0                 cmp    %rdx,%rax
  4020e3:      75 f3                    jne    4020d8 <main+0x328>
```

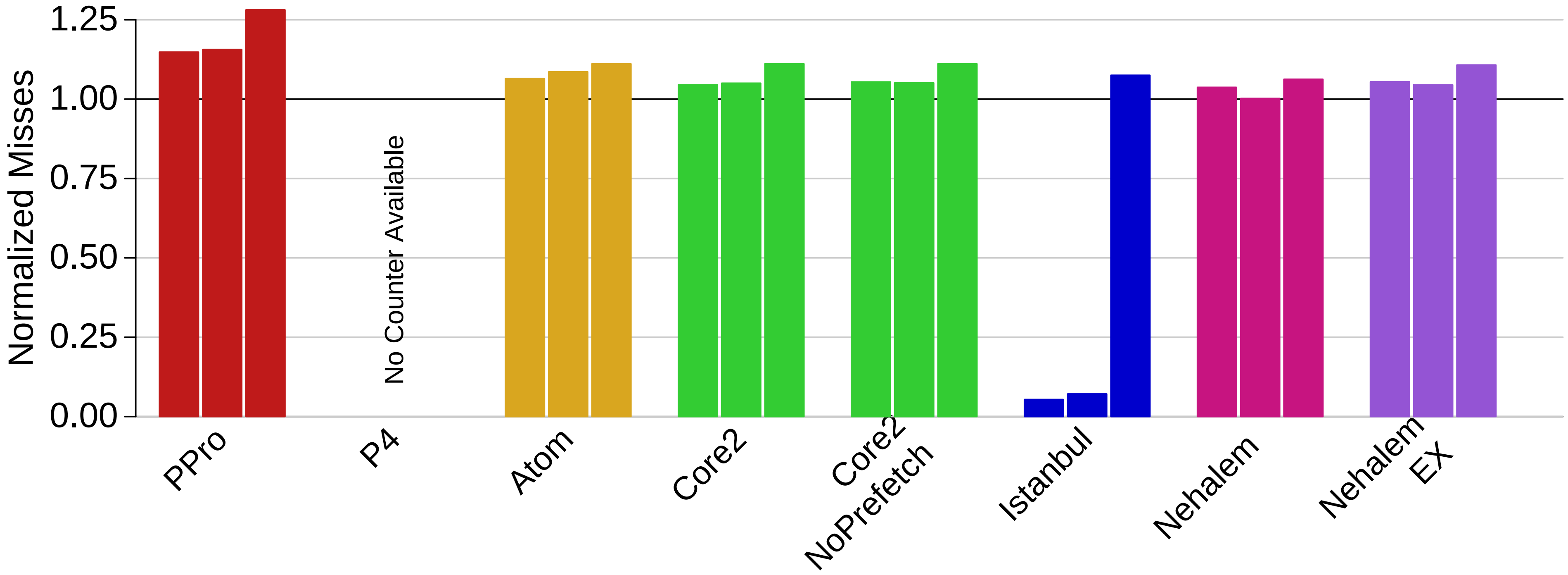
Unexpected Code

```
* 401e18:      f3 0f 10 44 24 0c        movss  0xc(%rsp),%xmm0
* 401e1e:      f3 0f 58 04 82           addss  (%rdx,%rax,4),%xmm0
  401e23:      48 83 c0 01              add    $0x1,%rax
  401e27:      48 3d e8 03 00 00        cmp    $0x3e8,%rax
* 401e2d:      f3 0f 11 44 24 0c        movss  %xmm0,0xc(%rsp)
  401e33:      75 e3                    jne    401e18 <main+0x398>
```

L1 Data Cache Misses

- Allocate array as big as L1 DCache
- Walk through the array byte-by-byte
- Count misses with `PAPI_L1_DCM` event
- If 32B line size, if linear walk through memory, first time will have 1/32 miss rate or 3.125%. Second time through (if fit in cache) should be 0%.

PAPI_L1_DCM – Forward/Reverse/Random


Normalized Misses
1.25
1.00
0.75
0.50
0.25
0.00
No Counter Available
PPro
P4
Atom
Core2
Core2 NoPrefetch
Istanbul
Nehalem
Nehalem EX

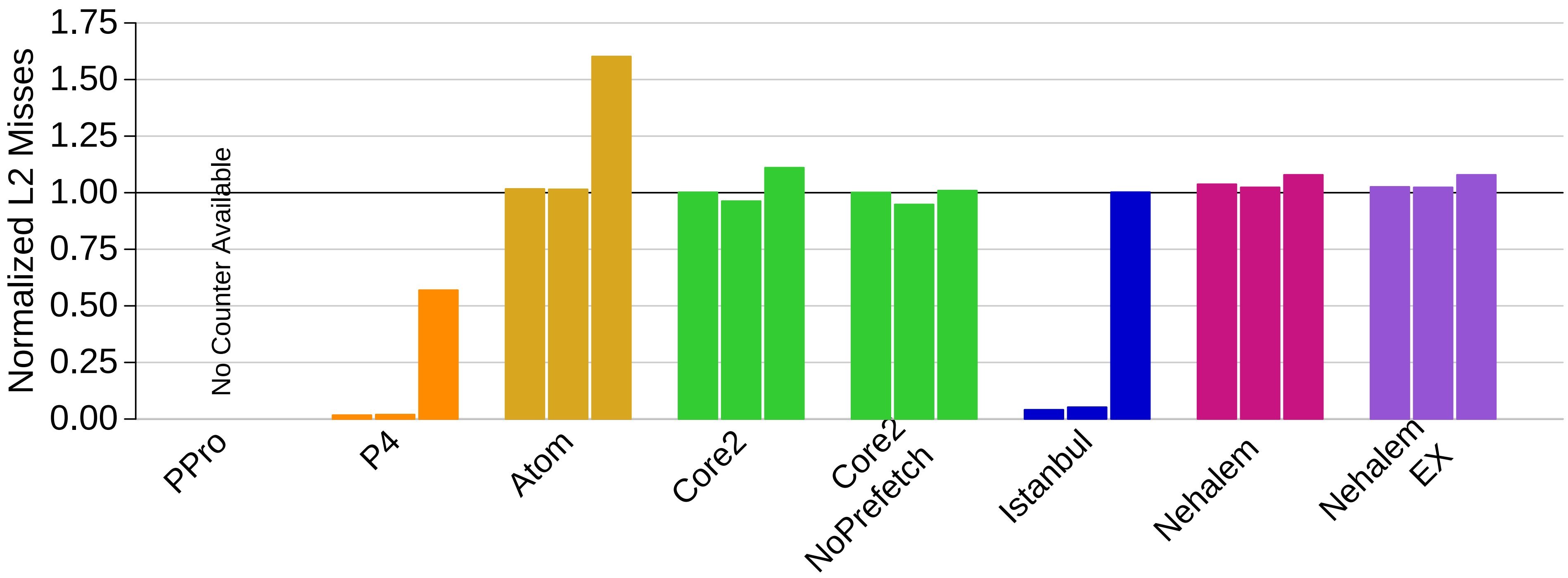
L1D Sources of Divergences

- Hardware Prefetching
- PAPI Measurement Noise
- Operating System Activity
- Non-LRU Cache Replacement

L2 Total Cache Misses

- Allocate array as big as L2 Cache
- Walk through the array byte-by-byte
- Count misses with `PAPI_L2_TCM` event

PAPI_L2_TCM – Forward/Reverse/Random



L2 Sources of Divergences

- Hardware Prefetching
- PAPI Measurement Noise
- Operating System Activity
- Non-LRU Cache Replacement
- Cache Coherency Traffic