ECE 571 – Advanced Microprocessor-Based Design Lecture 11

Vince Weaver

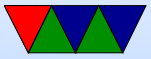
http://www.eece.maine.edu/~vweaver

vincent.weaver@maine.edu

23 February 2016

Announcements

- HW#5 Posted
- If you downloaded very early on, note that l1d-stores no longer works and I removed that question from the assignment.

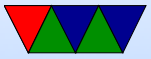


HW#4 Review

1. We are validating the claim that "one in five instructions is a branch". This is Haswell.

   (a) bzip2: 19.2B instructions, 2.8B branches, 2.5B cond branches
   roughly 15%, so roughly one in seven instructions a branch

   (b) equake_l: 1438B instructions, 123B branches, 91B cond branches
   roughly 8%, so more like one in 12



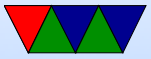
Is this expected? fp code often has longer stretches of codes, big loops, where integer codes traditionally have lots of small loops and if/else statements.

2. Branch miss rate on Haswell.

(a) bzip2: 2.8B branches, 207M branch misses, 7% miss rate

(b) equake_l: 120B branches, 587M misses, 0.5% miss rate

Is this expected? fp code usually has lots of big loops, integer code smaller loops, if/else.



3. Speculative Execution, Haswell

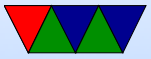
(a) bzip2: 22B uops retired, 30B uops executed = 27% not retired

(b) equake_l: 1800B uops retired 3200B uops executed = 43% not retired

That's a large number. Not sure why higher for fp, depends on other architectural things (also could be bugs in counters?)

4. bzip ratio, ARM64

(a) 20B instructions, 3B predicted, 256M mispredicted



16% branches

Why not branches events? ARM Cortex A57 not have full event support until Linux 4.4, wheras NVidia ships ancient 3.10 kernel.

5. bzip miss rate, ARM64

(a) 8%

6. (a) bzip vs equake ratios: fp has fewer branches
(b) bzip branch ratio x86 vs ARM64: about same. Note total instructions (19B vs 20B, ARM64 is RISC slightly less dense). Closer than I'd think though

(c) equake predict better, loops easier.

(d) arm64 mildly worse at branch prediction? improved over earlier versions would make interesting project

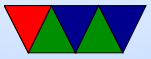
(e) speculative: equake much worse

(f) 50% miss rate

Code of mine: 500000 are random branches; 250699 of those were taken (50%)

```
if (( ((random()>>2)^(ra
    goto label_false
```

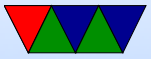
rand() is just implemented by a multiply and add



(pseudo rand)

in theory rand() and random() use the same algorithm,

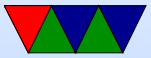
but rand() has branches last time I checked?



Prefetching

Try to avoid cache misses by bringing values into the cache before they are needed.

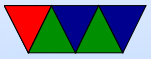
Caches with large blocksize already bring in extra data in advance, but can we do more?



Prefetching Concerns

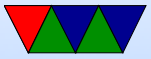
- When?
  We want to bring in data before we need it, but not too early or it wastes space in the cache.

- Where? What part of cache? Dedicated buffer?



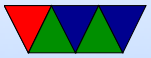
Limits of Prefetching

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



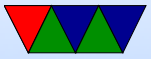
Implementation Issues

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



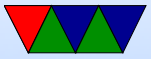
Software Prefetching

- 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



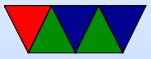
Hardware Prefetching – icache

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



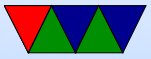
Hardware Prefetching – dcache

- 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)

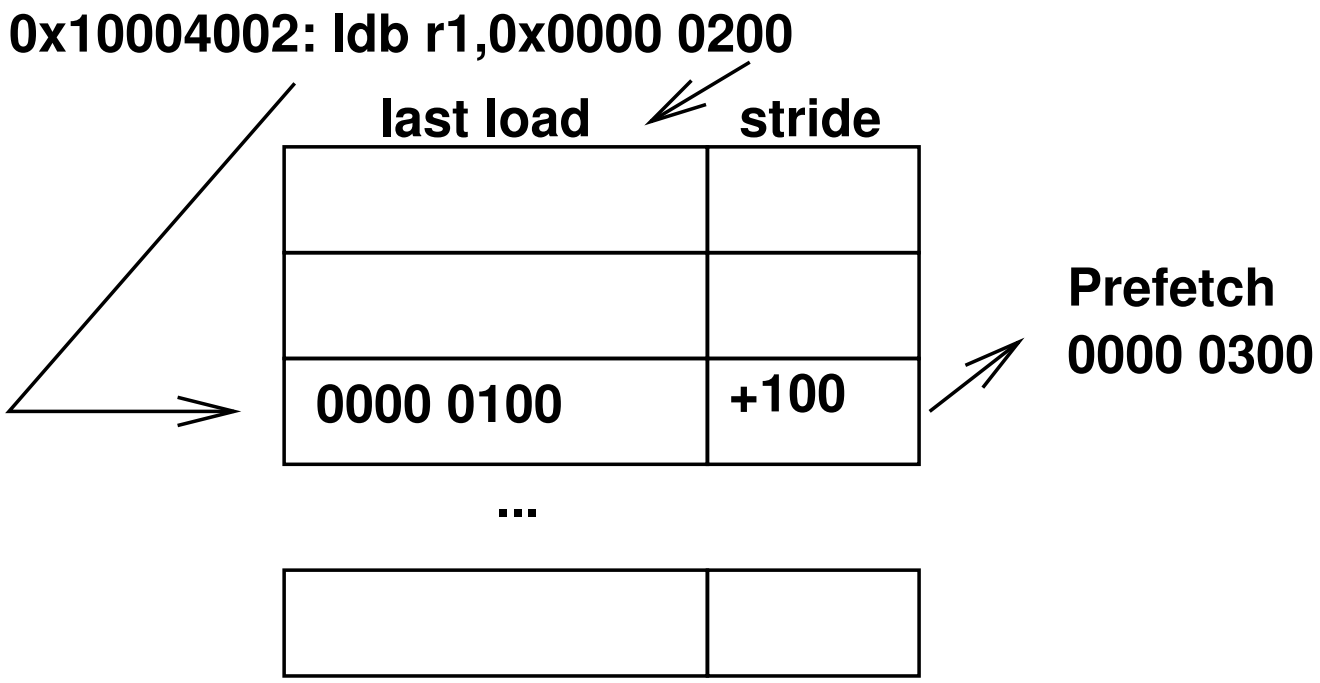


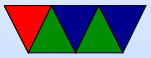
Hardware Prefetching – Stride Prefetching

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



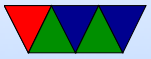
Stride Predictor





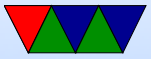
Hardware Prefetching – Correlation/Content-Directed Prefetching

- 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



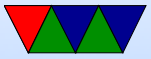
Using 2-bit Counters

- 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)



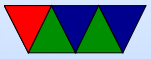
Cortex A9 Prefetch

- 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,

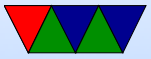


a DSB (barrier) or a PLD instruction executes, or the program does not hit in the prefetched lines.

- PLD requests always take precedence

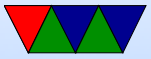


Investigating Prefetching Using Hardware Performance Counters



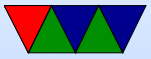
Quick Look at Core2 HW Prefetch

- 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



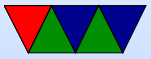
x86 SW Prefetch Instructions (AMD)

- `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



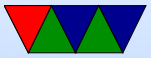
Core2

- `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



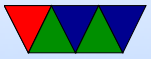
AMD (Istanbul and Later)

- `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



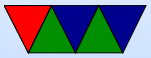
Atom

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



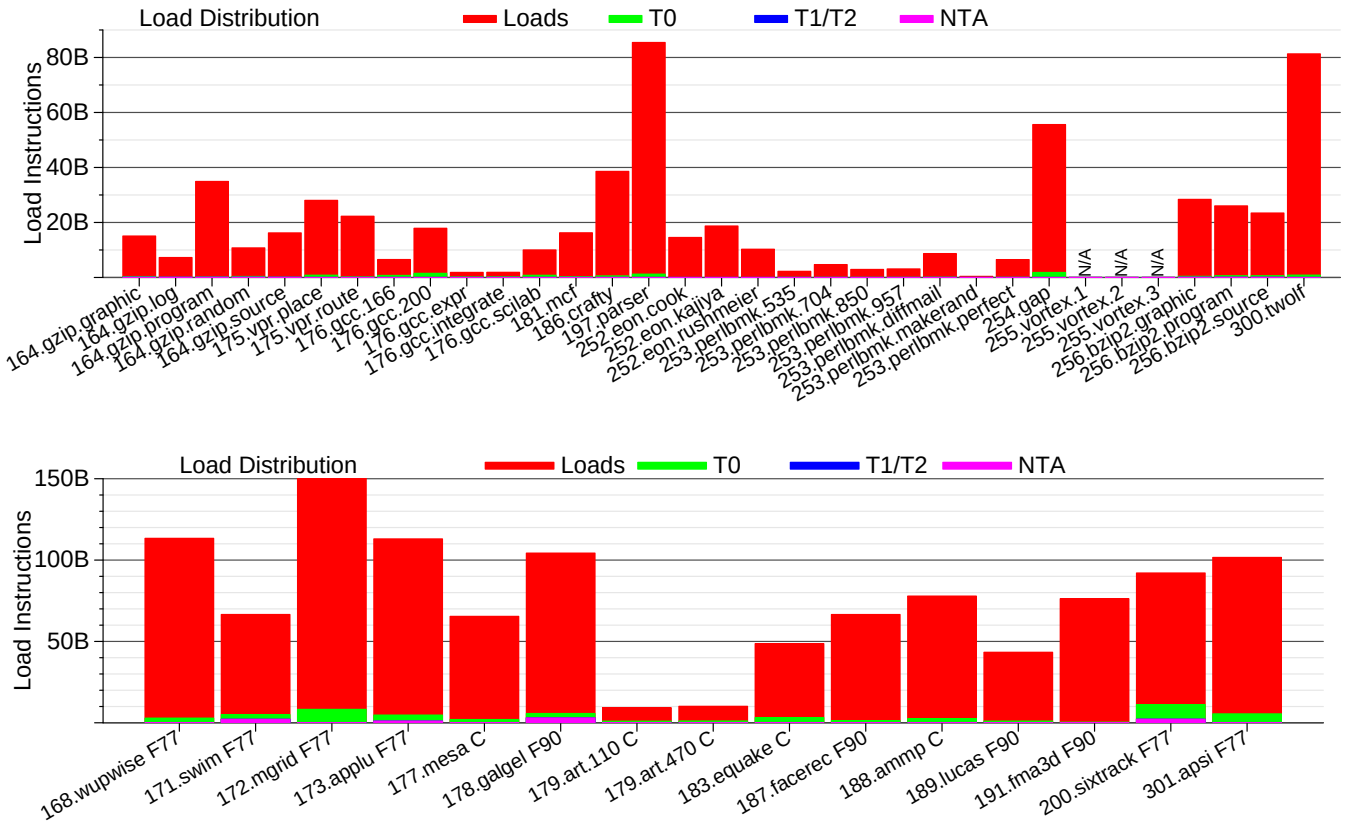
Does anyone use SW Prefetch?

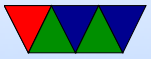
- gcc by default disables SW prefetch unless you specify `-fprefetch-loop-arrays`
- icc disables unless you specify `-xsse4.2 -op-prefetch=4`
- glibc has hand-coded SW prefetch in `memcpy()`
- Prefetch can hurt behavior:
  - Can throw out good cache lines,
  - Can bring lines in too soon,
  - Can interfere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



SW Prefetch Distribution

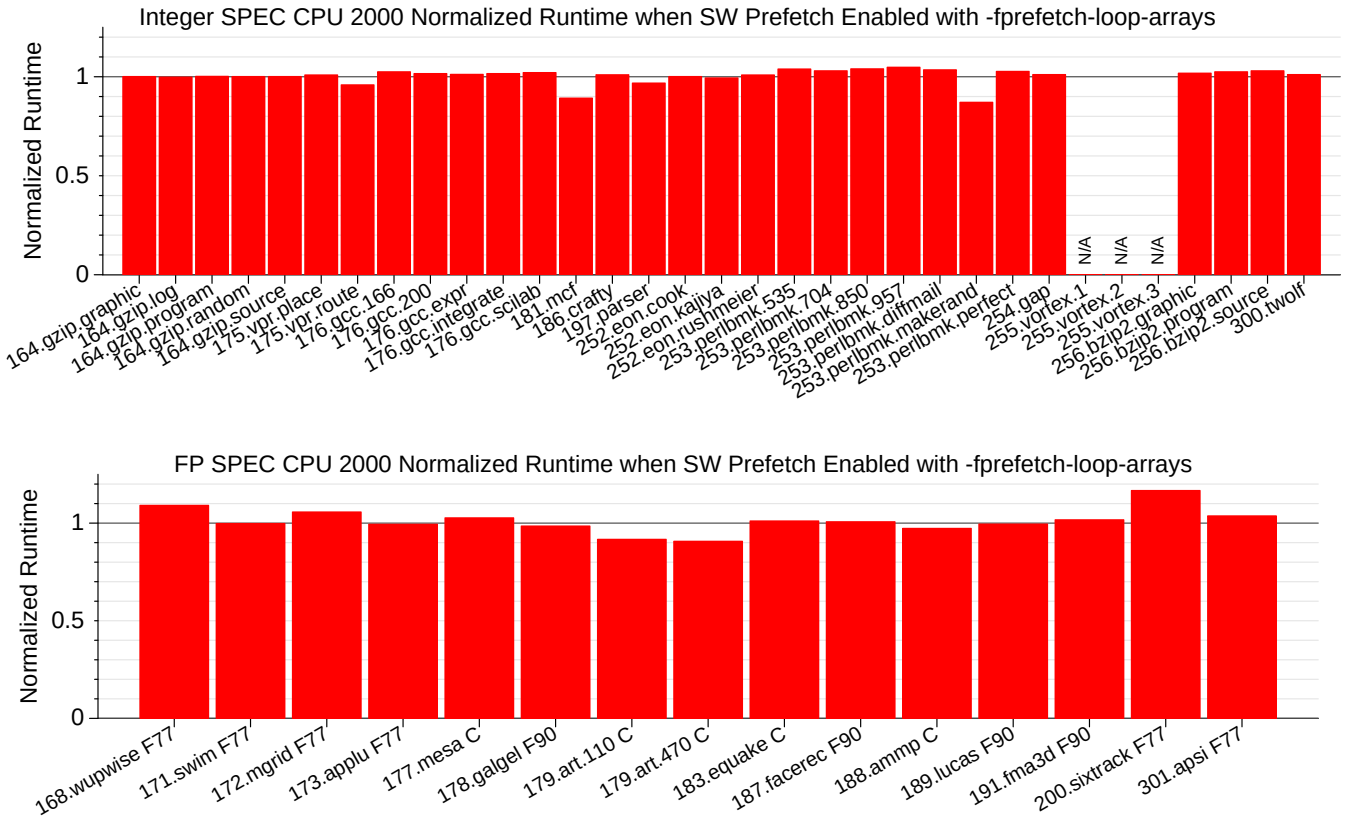
SPEC CPU 2000, Core2, gcc `-fprefetch-loop-arrays`

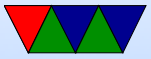




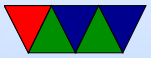
Normalized SW Prefetch Runtime

on Core2 (Smaller is Better)



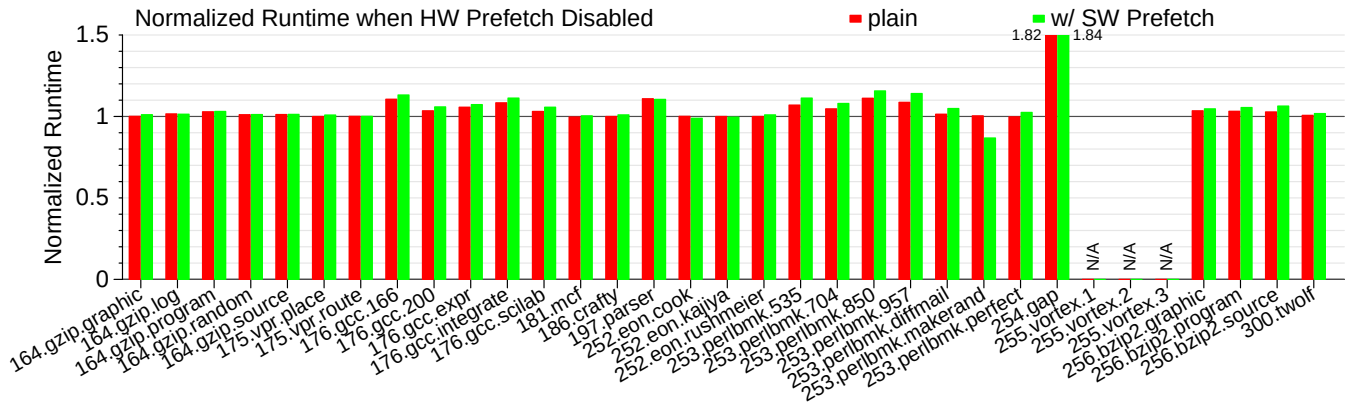


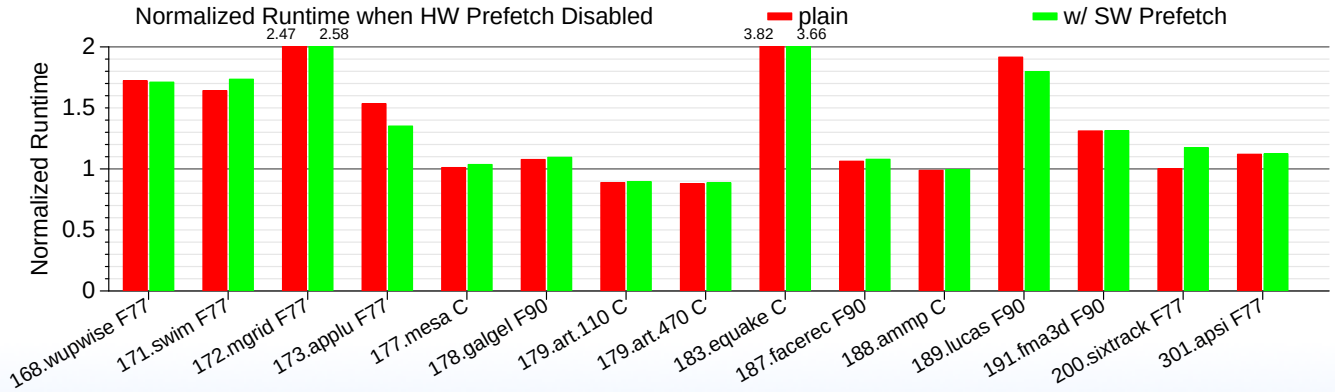
The HW Prefetcher on Core2 can be Disabled

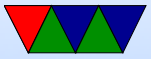


Runtime with HW Prefetcher Disabled

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

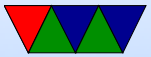






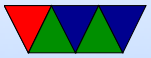
PAPI_PRF_SW Revisited

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

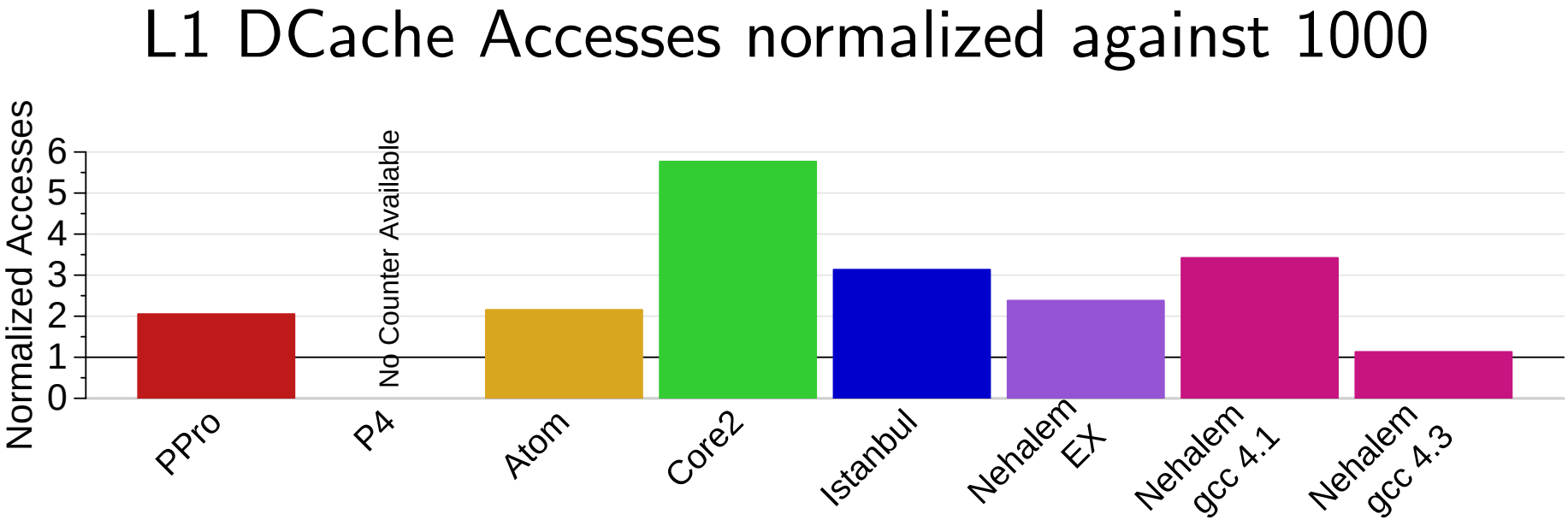


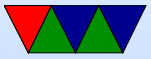
L1 Data Cache Accesses

```
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



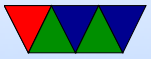
PAPI_L1_DCA

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>
```

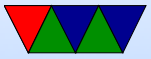
Unexpected Code

```
* 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>
```

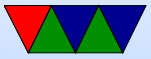


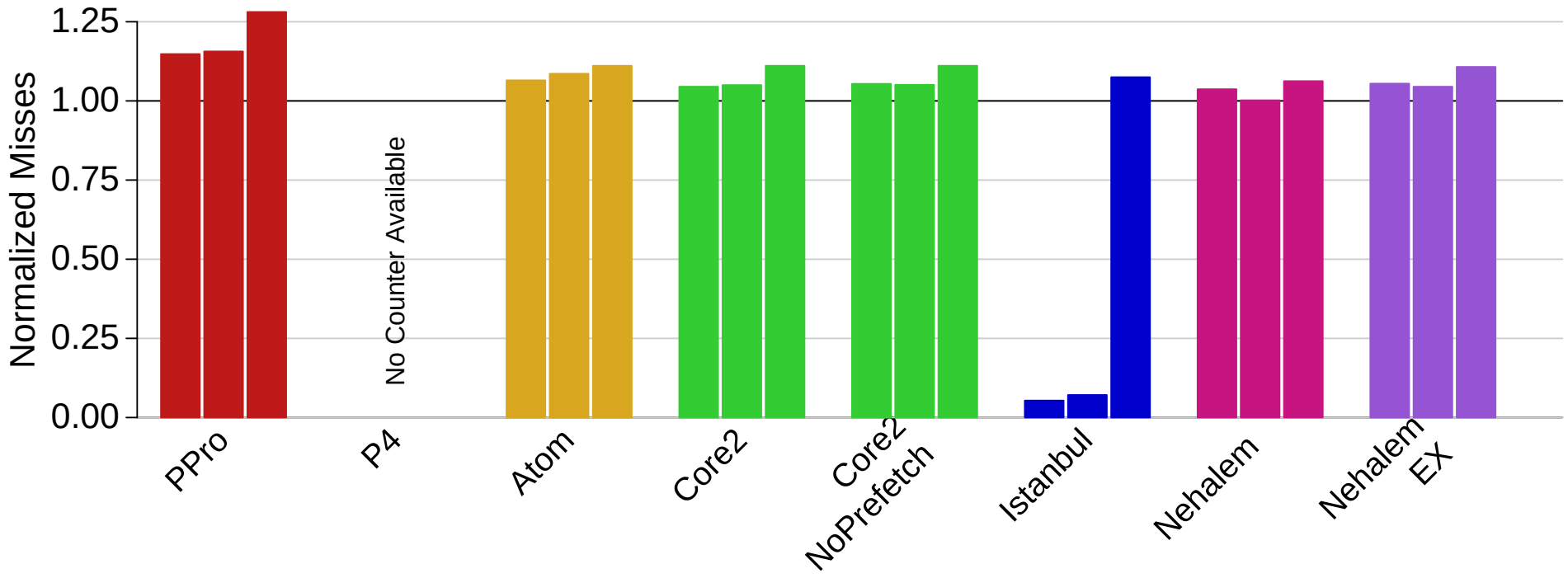
L1 Data Cache Misses

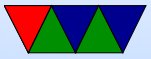
- Allocate array as big as L1 DCache
- Walk through the array byte-by-byte
- Count misses with `PAPI_L1_DCM` event
- If 32B line size, if linear walk through memory, first time will have 1/32 miss rate or 3.125%. Second time through (if fit in cache) should be 0%.



PAPI_L1_DCM – Forward/Reverse/Random

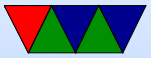



Normalized Misses
1.25
1.00
0.75
0.50
0.25
0.00
No Counter Available
PPro
P4
Atom
Core2
Core2 NoPrefetch
Istanbul
Nehalem
Nehalem EX



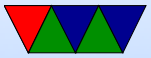
L1D Sources of Divergences

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

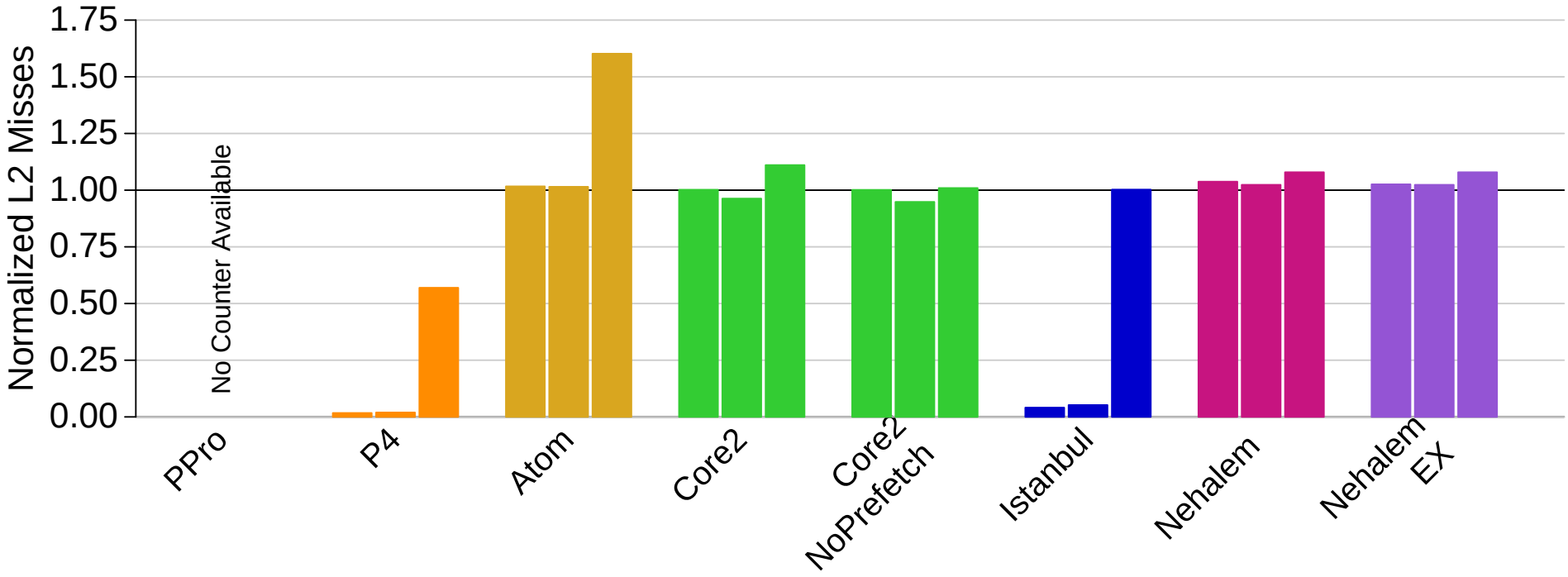


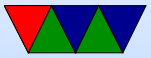
L2 Total Cache Misses

- 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