ECE 571 – Advanced Microprocessor-Based Design Lecture 12

Vince Weaver

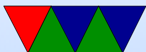
http://web.eece.maine.edu/~vweaver

vincent.weaver@maine.edu

6 October 2022

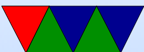
Announcements

- Don't forget HW#5
- HW#6 will be posted (prefetchers)
- Useful reading: "When prefetching works, when it doesn't and why" paper by Lee et al.
- Remember Fall Break is Tuesday



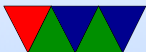
Prefetching

- 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?



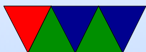
Prefetching Concerns

- 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?



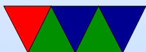
Limits of Prefetching

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



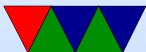
Implementation Issues

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



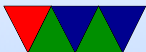
Software Prefetching

- 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



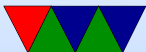
HW Prefetch Strategies – icache

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



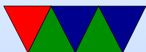
HW Prefetch Strategies – dcache

- 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)

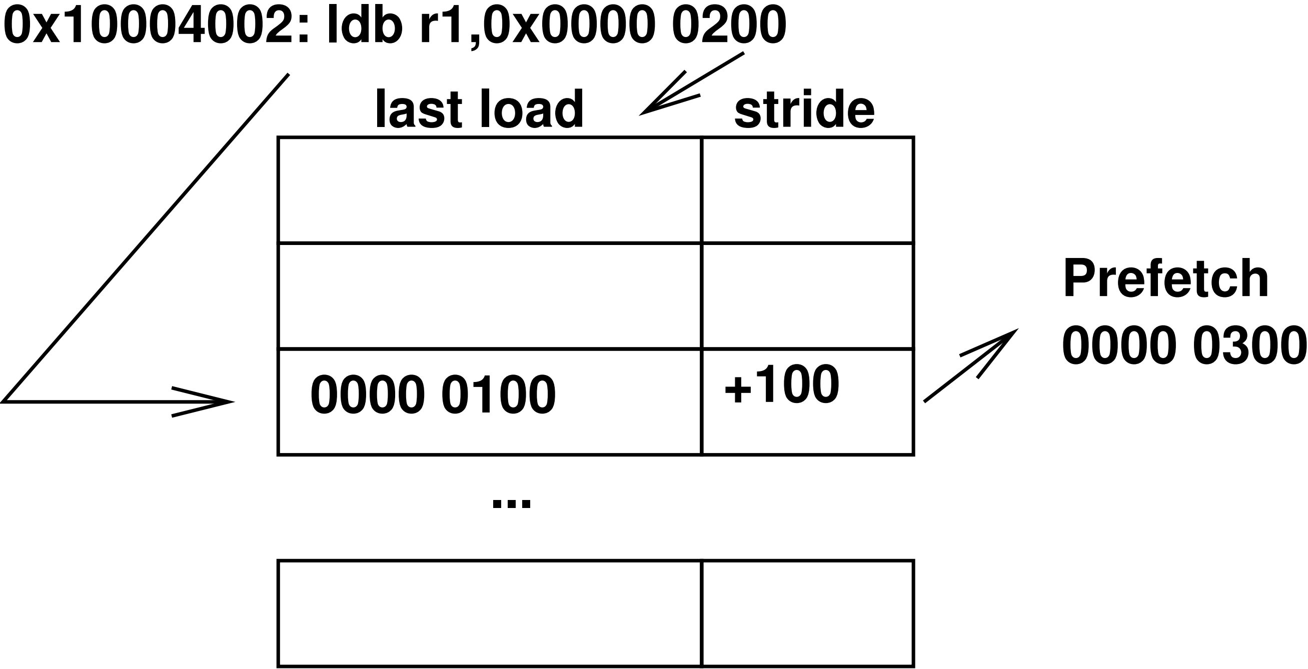


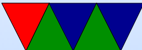
Hardware Prefetching – Stride Prefetching

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



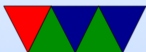
Stride Predictor





Hardware Prefetching – Correlation/Content-Directed Prefetching

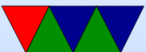
- How to handle things like pointer chasing / linked lists?
- Correlation – records sequence of misses, then when traversing again prefetches in that order
- Content directed – recognize pointers and pre-fetch what they point to



Using 2-bit Counters

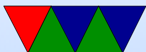
- Use 2-bit counter to see if load causing lots of misses, if so automatically treat as streaming load (Rivers)

- Partitioned cache: cache stack, heap, etc, (or little big huge) separately (Lee and Tyson)



SW Prefetch notes from paper

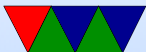
- *When Prefetching Works, When it Doesn't, and Why* by Lee, Kim, and Vuduc (ACM TACO 2012)
- Experiment on some SPEC CPU 2006 benchmarks, some helped, some hurt, some same
- Times SW Prefetch works well
  - Large number of streams (more than available tables)
  - Short streams (takes while to train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



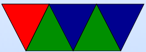
L3)

  - 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

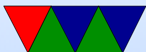


tougher ones and less to learn from



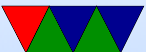
Cortex A9 Prefetch

- 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,



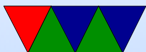
a DSB (barrier) or a PLD instruction executes, or the program does not hit in the prefetched lines.

- PLD requests always take precedence



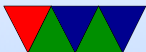
Quick Look at Haswell Prefetch

- 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



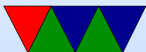
Investigating Prefetching Using Hardware Performance Counters

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



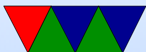
Quick Look at Core2 Prefetch

- 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



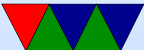
x86 SW Prefetch Instructions (AMD)

- `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



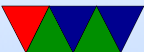
Core2

- `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



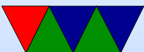
AMD (Istanbul and Later)

- `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



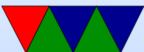
Atom

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



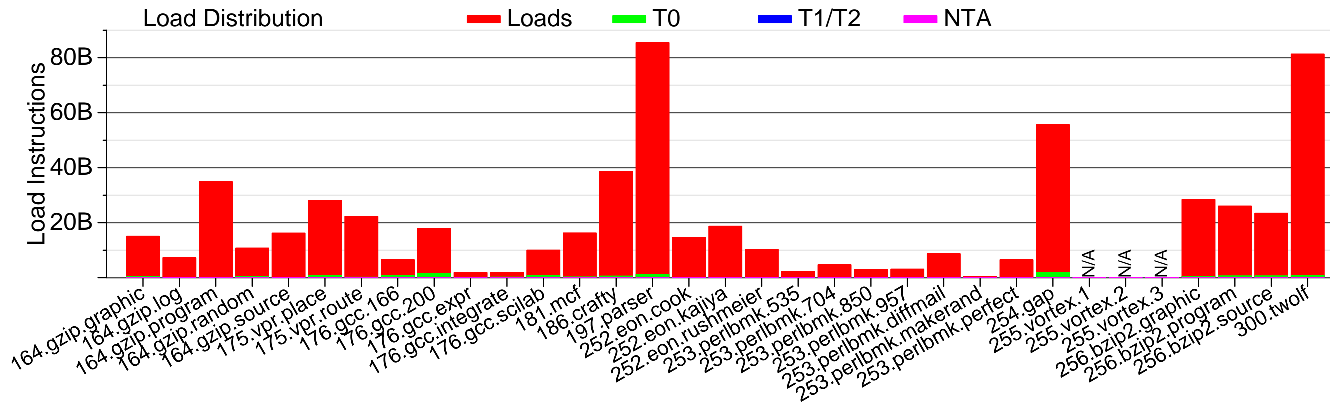
Does anyone use SW Prefetch?

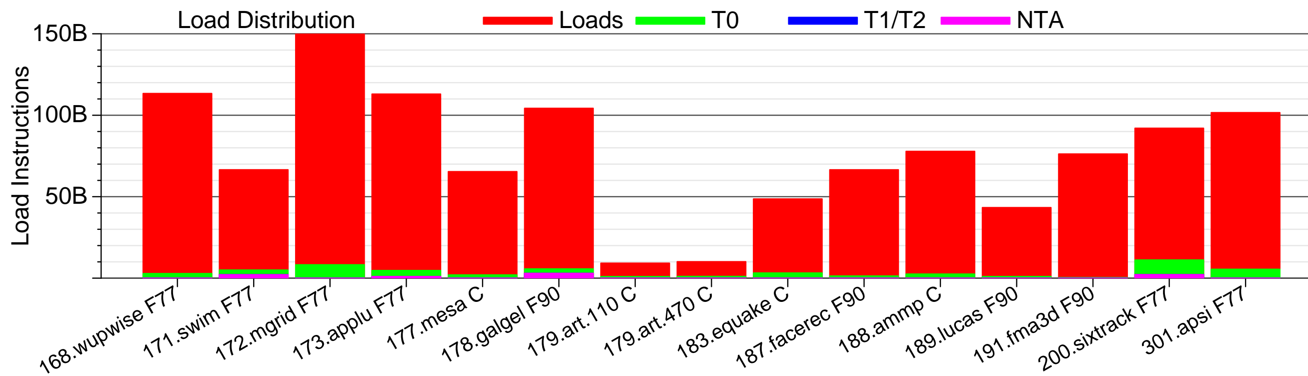
- gcc by default disables SW prefetch unless you specify `-fprefetch-loop-arrays`
- icc disables unless you specify `-xsse4.2 -op-prefetch=4`
- glibc has hand-coded SW prefetch in `memcpy()`
- Prefetch can hurt behavior:
  - Can throw out good cache lines,
  - Can bring lines in too soon,
  - Can interfere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

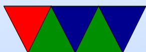


SW Prefetch Distribution

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

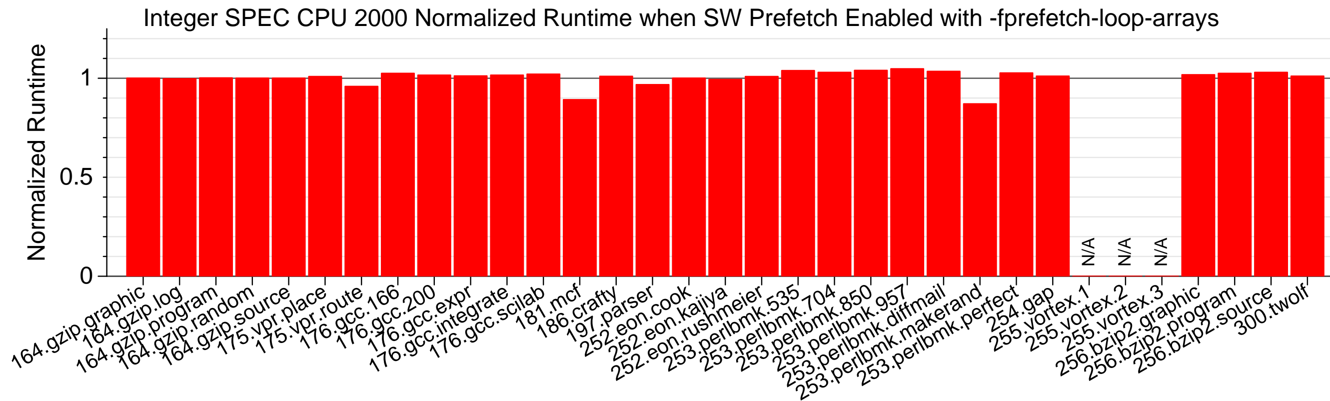


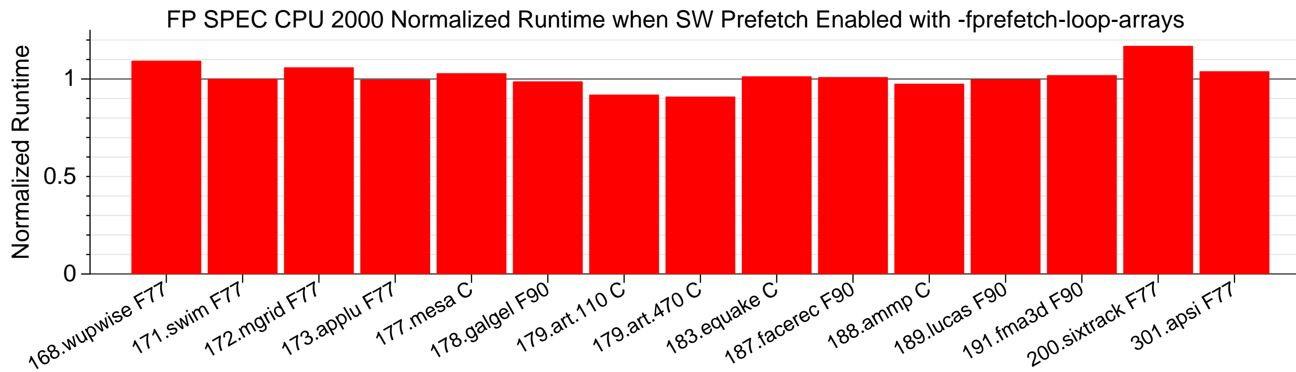


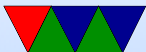


Normalized SW Prefetch Runtime

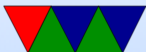
on Core2 (Smaller is Better)





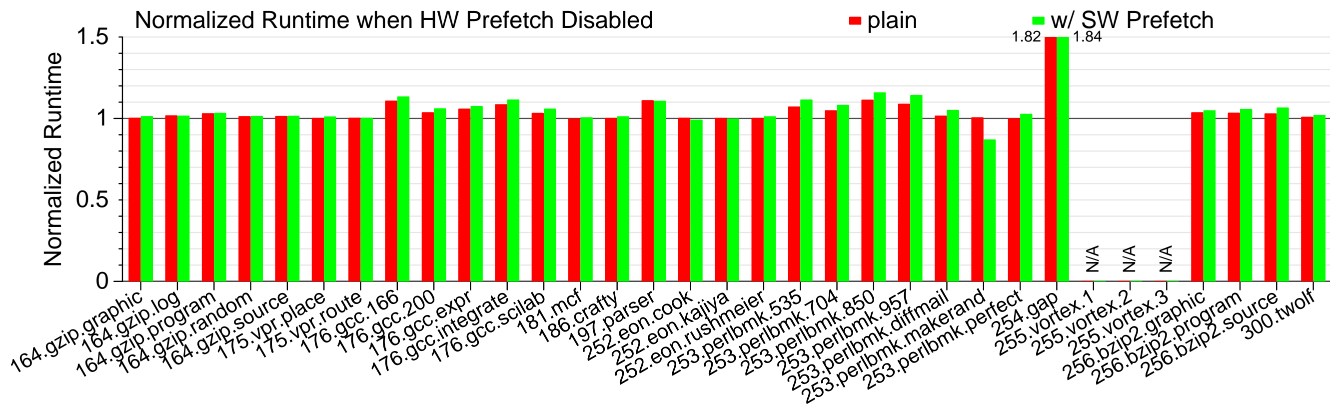


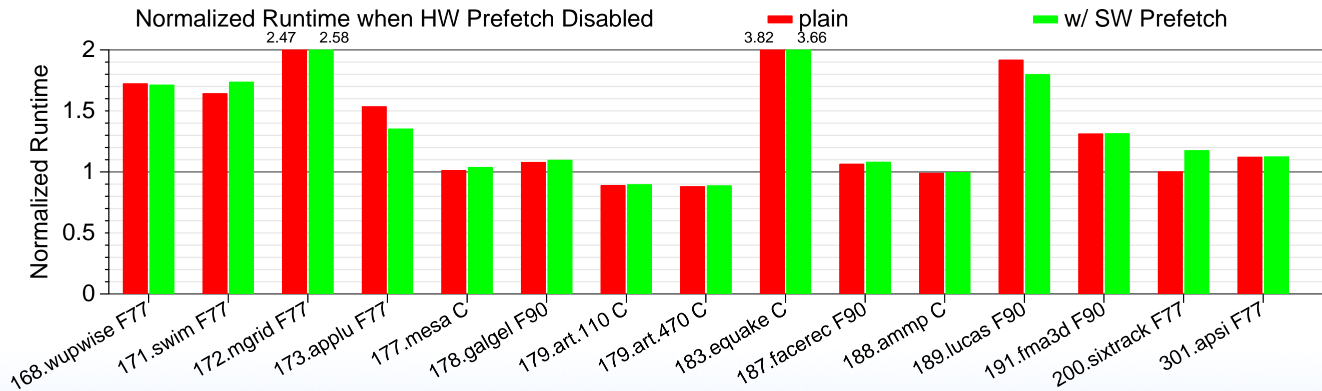
The HW Prefetcher on Core2 can be Disabled

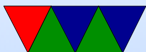


Runtime with HW Prefetcher Disabled

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

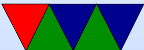






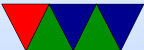
PAPI_PRF_SW Revisited

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

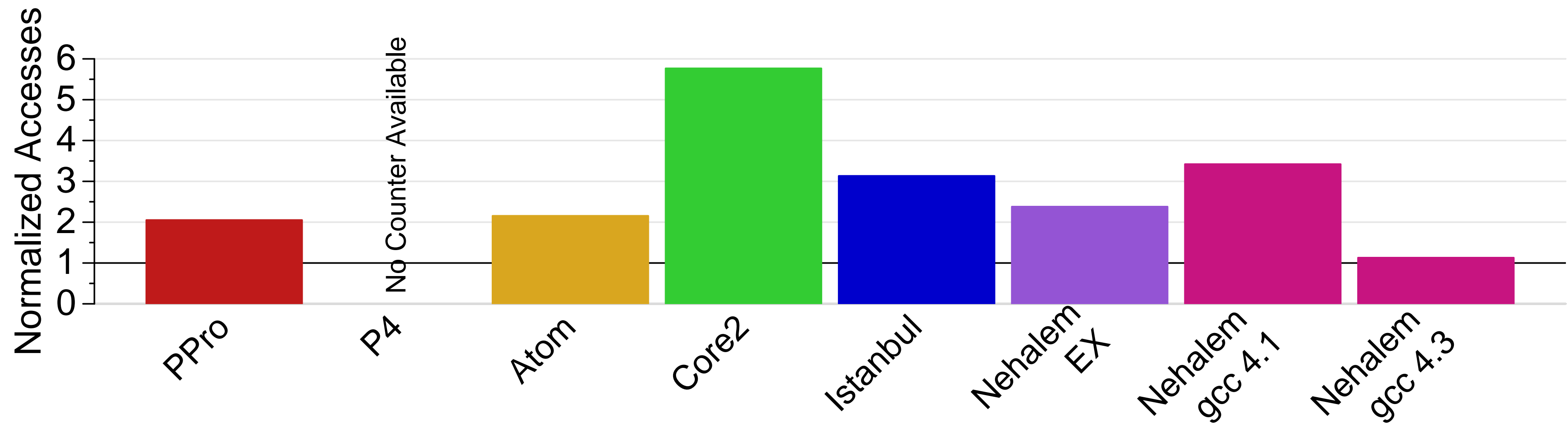


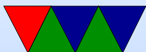
L1 Data Cache Accesses

```
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





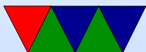
PAPI_L1_DCA

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>
```

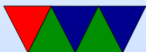
Unexpected Code

```
* 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>
```

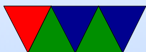


L1 Data Cache Misses

- 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

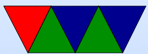


Normalized Misses

1.25
1.00
0.75
0.50
0.25
0.00

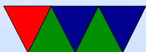
No Counter Available

PPro
P4
Atom
Core2
Core2 NoPrefetch
Istanbul
Nehalem
Nehalem EX



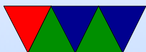
L1D Sources of Divergences

- 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



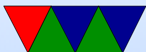
PAPI_L2_TCM – Forward/Reverse/Random

Normalized L2 Misses

1.75
1.50
1.25
1.00
0.75
0.50
0.25
0.00

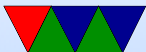
No Counter Available

PPro
P4
Atom
Core2
Core2 NoPrefetch
Istanbul
Nehalem
Nehalem EX



L2 Sources of Divergences

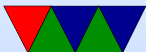
- Hardware Prefetching
- PAPI Measurement Noise
- Operating System Activity
- Non-LRU Cache Replacement
- Cache Coherency Traffic



Cache Performance Measurement

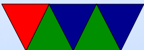
Matrix-Matrix multiply is the typical example.

Despite being a big deal in HPC, MMM happens in embedded world too.



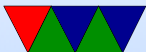
Naive Matrix-Matrix Multiply 1

```
#define MATRIX_SIZE 512
static double a[MATRIX_SIZE][MATRIX_SIZE];
static double b[MATRIX_SIZE][MATRIX_SIZE];
static double c[MATRIX_SIZE][MATRIX_SIZE];

for(i=0;i<MATRIX_SIZE;i++) {
    for(j=0;j<MATRIX_SIZE;j++) {
        for(k=0;k<MATRIX_SIZE;k++) {
            c[i][j]+=a[i][k]*b[k][j];
        }
    }
}
```



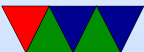
Naive Matrix-Matrix Multiply 1 – what's the issue?

- Branch Misses?
- TLB Misses?
- ICache Misses?
- DCache Misses?
- L2 Cache Misses?



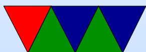
Naive Matrix-Matrix Multiply 1 – perf results

```
Matrix multiply sum: s=1016404871450364.375000
Performance counter stats for './matrix_multiply':
         2,783.31 msec task-clock                #    0.999 CPUs utilized
              326      context-switches          #  117.127 /sec
                0      cpu-migrations            #    0.000 /sec
            1,104      page-faults               #  396.651 /sec
    8,707,341,986      cycles                    #    3.128 GHz
    8,633,400,413      instructions              #    0.99  insn per cycl
    1,080,685,729      branches                  #  388.274 M/sec
        1,084,737      branch-misses             #    0.10% of all branch

      2.785585085 seconds time elapsed
      2.772288000 seconds user
      0.012001000 seconds sys
```



Naive Matrix-Matrix Multiply 1 – Cache Stalls

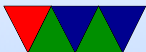
```
 perf stat -e cycles,cycle_activity.stalls_l1d_pending,cycle_activity.stal
Matrix multiply sum: s=1016404871450364.375000

 Performance counter stats for './matrix_multiply':

     8,733,443,210      cycles
     2,512,840,181      cycle_activity.stalls_l1d_pending
     4,380,477,182      cycle_activity.stalls_l2_pending
         2,167,274      icache.ifetch_stall

       2.893835652 seconds time elapsed

       2.884815000 seconds user
       0.008002000 seconds sys
```



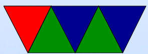
Naive Matrix-Matrix Multiply 1 – Cache results

```
perf stat -e L1-dcache-loads:u,L1-dcache-loads-misses:u  ./matrix_multiply
Matrix multiply sum: s=1016404871450364.375000

 Performance counter stats for './matrix_multiply':

     2,149,619,922      L1-dcache-loads
     1,115,775,552      L1-dcache-loads-misses    #   51.91% of all L1-dca

       2.809774785 seconds time elapsed

       2.796647000 seconds user
       0.012002000 seconds sys
```

`l2_rqsts.references:u` and `l2_lines_in.all:u` L2



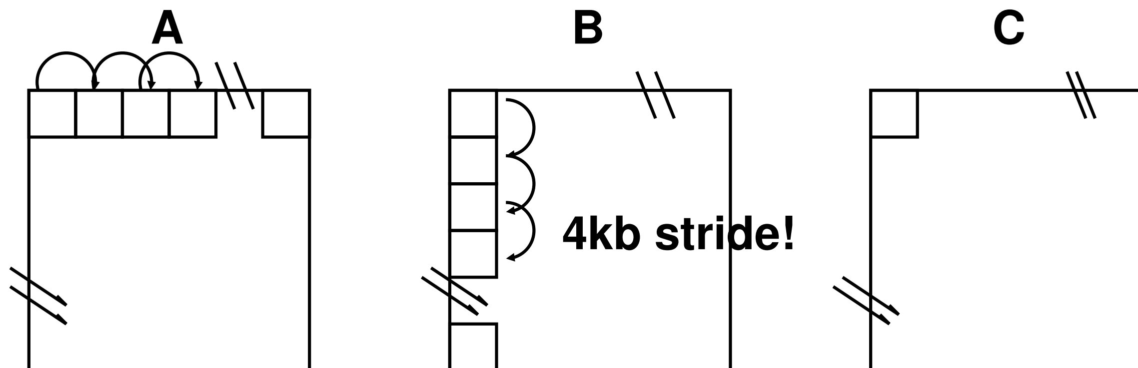
$= 1.0\text{B}/1.1\text{B}$

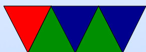


Naive Matrix-Matrix Multiply 1 – What's the Issue?

3 billion memory accesses about right (1024x1024x1024x3)

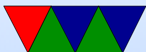
```
1,076,901,569        L1-dcache-stores
2,149,619,592        L1-dcache-loads
```





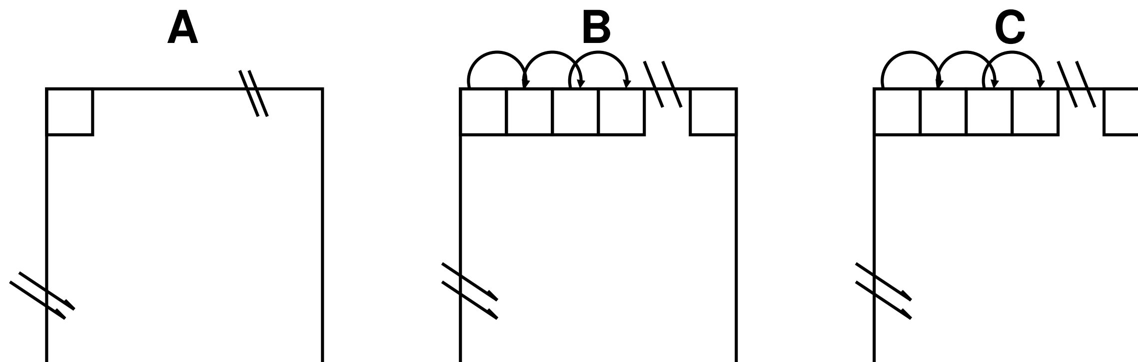
Note: what's the best?

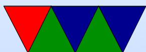
I think recently you can get $O(n^2.7)$



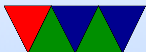
Switch the loop ordering

```
for(i=0;i<MATRIX_SIZE;i++) {
    for(k=0;k<MATRIX_SIZE;k++) {
        for(j=0;j<MATRIX_SIZE;j++) {
            c[i][j]+=a[i][k]*b[k][j];
        }
    }
}
```

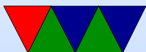




Naive Matrix-Matrix Multiply 2 – perf results

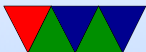


```
Matrix multiply sum: s=1016404871450368.875000

 Performance counter stats for './matrix_multiply_swapped':

            924.58 msec task-clock                #    0.999 CPUs utilized
               104      context-switches          #  112.484 /sec
                 0      cpu-migrations            #    0.000 /sec
             1,095      page-faults               #    1.184 K/sec
     2,342,399,985      cycles                    #    2.533 GHz
     8,626,089,078      instructions              #    3.68  insn per cycl
     1,078,729,068      branches                  #    1.167 G/sec
         1,068,154      branch-misses             #    0.10% of all branch

       0.925868346 seconds time elapsed
       0.917551000 seconds user
       0.008013000 seconds sys
```



Naive Matrix-Matrix Multiply 2 – DCache Stalls

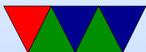
```
Matrix multiply sum: s=1016404871450368.875000

 Performance counter stats for './matrix_multiply_swapped':

     2,368,380,342      cycles
        13,677,345      cycle_activity.stalls_l1d_pending
        13,395,580      cycle_activity.stalls_l2_pending
           936,833      icache.ifetch_stall

       0.923080202 seconds time elapsed

       0.910786000 seconds user
       0.012036000 seconds sys
```



Naive Matrix-Matrix Multiply 2 – DCache results

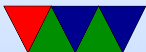
```
Matrix multiply sum: s=1016404871450368.875000

 Performance counter stats for './matrix_multiply_swapped':

     3,222,307,315      L1-dcache-loads
       138,886,482      L1-dcache-loads-misses    #    4.31% of all L1-dca

       0.923336583 seconds time elapsed

       0.910938000 seconds user
       0.012038000 seconds sys
```

$L2 = 273M/138M$



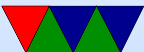
Other Ways to Optimize

- Tiling
- Parallelizing



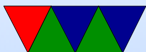
Use ATLAS/BLAS

```
cblas_dgemm(CblasRowMajor,
      CblasNoTrans,CblasNoTrans,
      1024,1024,1024,
      1.0,(const double *)a,1024,
      (const double *)b,1024,
      1.0,(double *)c,1024);
```



Matrix-Matrix Mul ATLAS – perf results

- time = 0.28s real, 0.28 user
- IPC 3.0
- Stalls = 630M total, 12M/12M/862k
- L1 Cache miss rate = 2.6%
- L2 Cache miss rate = 6M/26M



Matrix-Matrix Mul OpenBlas – perf results

- time = 0.05 real, 0.67 user
- IPC 0.5
- Stalls = 3.8B total, 85M/1B/3M
- L1 Cache miss rate = 20%
- L2 Cache miss rate = 12M/81M