Performance Engineering with Profiling Tools

Reid Kleckner

John Dong

Agenda

- Theory/Background: Profiling Tools
- 2 Interactive Walkthroughs:
  - Matrix Multiply
    - Simple cache ratio measurements using the profiler
  - Branchless Sorting
    - Optimizing instruction-level parallelism / pipelining
    - Real example of how the 6.172 staff used the profiler

Theory

- "*Premature optimization is the root of all evil*"- Knuth
- Should focus on optimizing **hotspots**
- Project 1: Worked with small programs with easy-to-spot hotspots
- Real world codebases much bigger: Reading all the code is a waste of time (for optimizing)
- **Profiling**: Identifies where your code is slow

What is the bottleneck?

- Could be:
  - **CPU**
  - **Memory**
  - Network
  - Disk
  - SQL DB
  - User Input (probably not this class)
- Solution depends heavily on the problem
- Today: Focus on CPU and Memory

Profiling Tools

In order to do..	You can use…
Manual Instrumentation	`printf`, (or fancy variants thereof)
Static Instrumentation	`gprof`
Dynamic Instrumentation	`callgrind`, `cachegrind`, `DTrace`
Performance Counters	`oprofile`, `perf`
Heap Profiling	`massif`, `google-perftools`
Other tools exist for Network, Disk IO, Software-specific, … TODAY: `perf`	

Event Sampling

- Basic Idea:
  - Keep a list of where "interesting events" (cycle, branch miss, etc) happen
- Actual Implementation:
  - Keep a counter for each event
  - When a counter reaches threshold, fire interrupt
  - Interrupt handler: Record execution context
- A tool (`perf`) turns data into useful reports

Intel Performance Counters

- CPU Feature: Counters for hundreds of events
  - Performance: Cache misses, branch misses, instructions per cycle, …
  - CPU sleep states, power consumption, etc (not interesting for this class)
- Today & Project 2.1: We’ll cover the most useful CPU counters for this class
- **Intel® 64 and IA-32 Architectures Software Developer's Manual:** Appendix A lists all counters
  - http://www.intel.com/products/processor/manuals/index.htm

Linux:
Performance Counter Subsystem

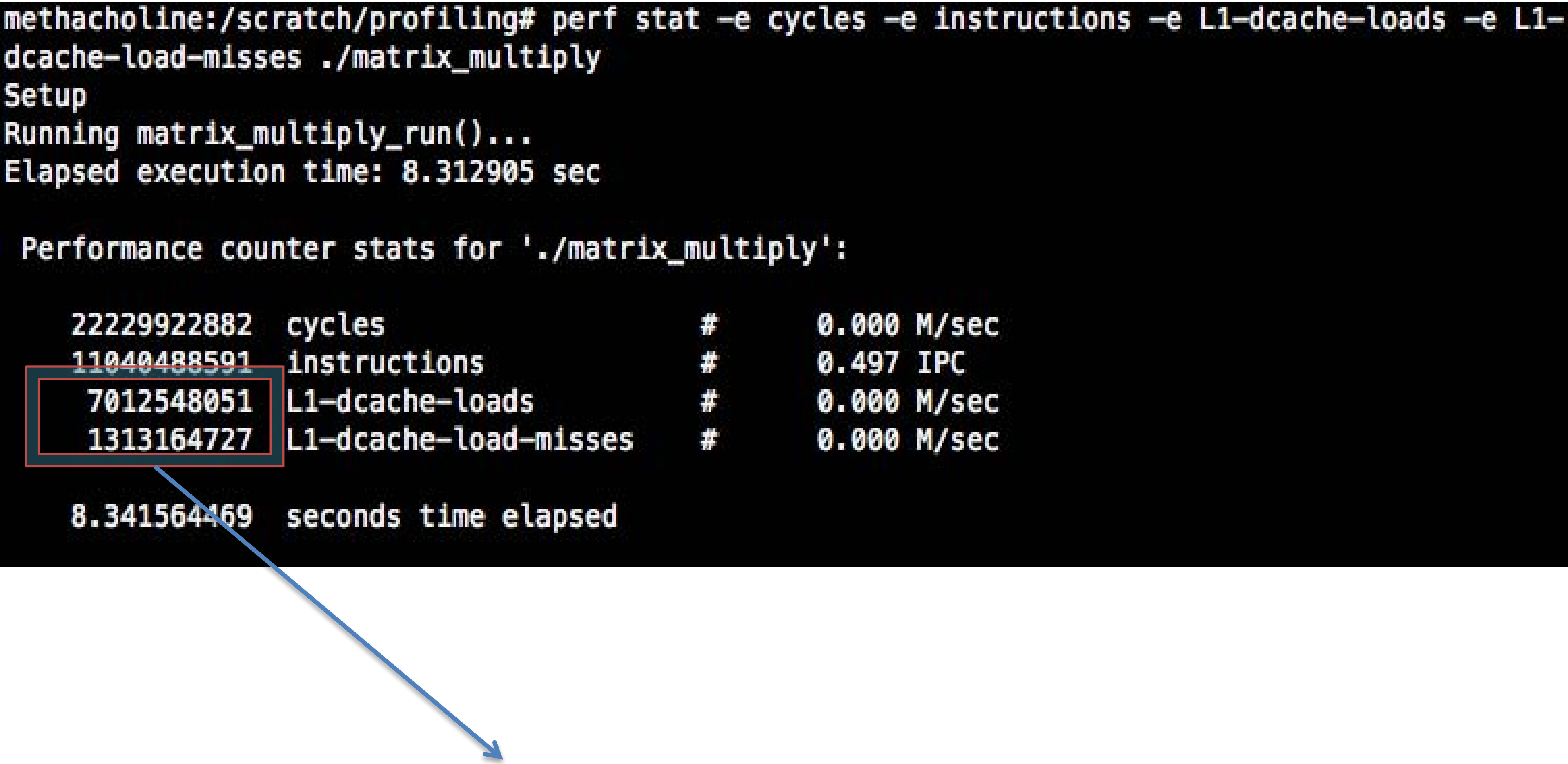
- New event sampling tool (2.6.31 and above)
  - Older tools: oprofile, perfmon
- Can monitor software and hardware events
  - Show all predefined events: `perf list`
  - Define your own performance counters…

- On your machine: `perf`in **linux-tools**

https://perf.wiki.kernel.org/

Demo 1: Matrix Multiply

```
intmatrix_multiply_run(const matrix* A, const matrix* B, matrix* C)
{
inti, j, k;
for (i = 0; i< A->rows; i++) {
for (j = 0; j< B->cols; j++) {
for (k = 0; k< A->cols; k++) {
        C->values[i][j] +=
 A->values[i][k] * B->values[k][j];
      }
    }
  }
}
```



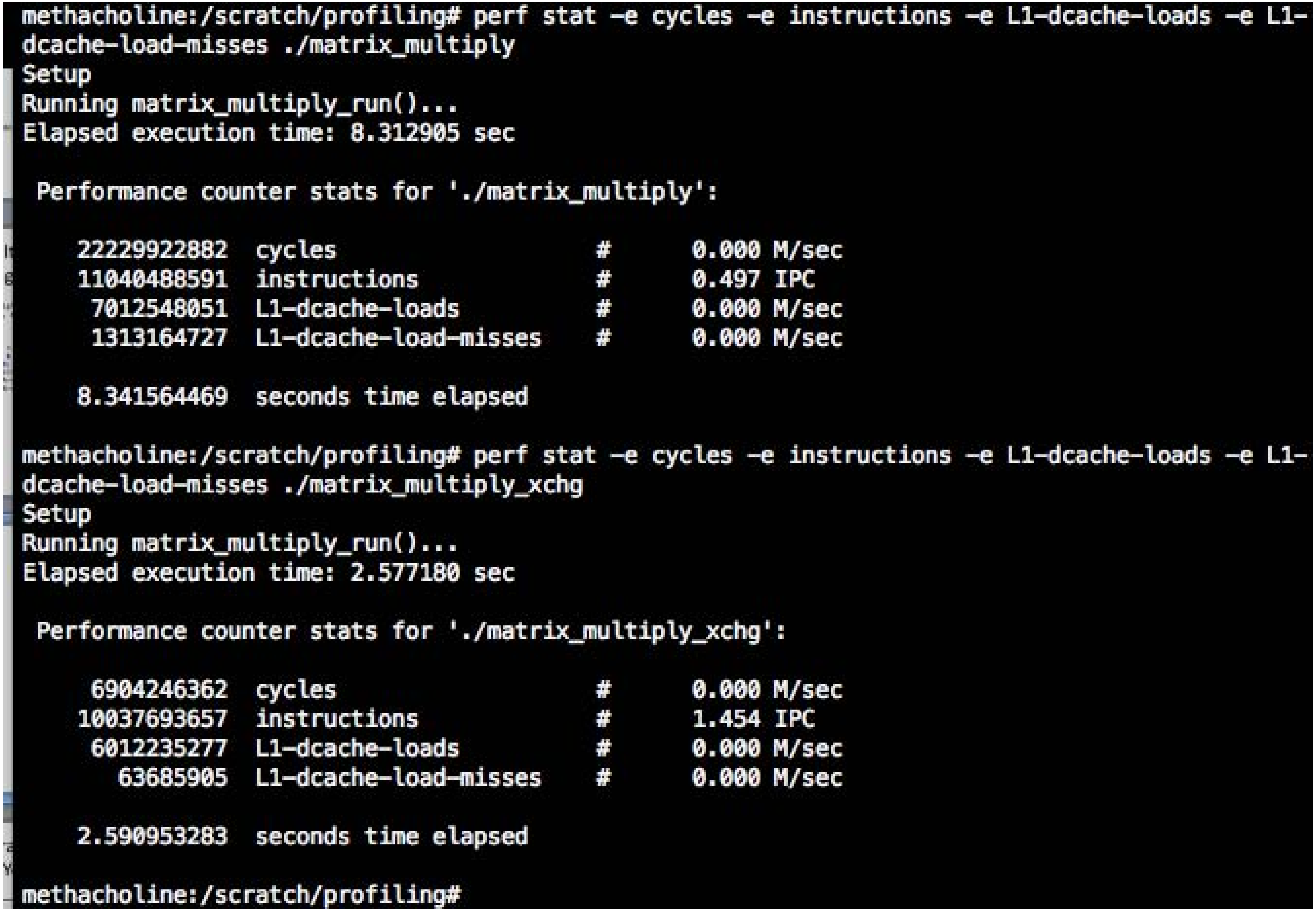
```
methacholine:/scratch/profiling# perf stat -e cycles -e instructions -e L1-dcache-loads -e L1-dcache-load-misses ./matrix_multiply
Setup
Running matrix_multiply_run()...
Elapsed execution time: 8.312905 sec

 Performance counter stats for './matrix_multiply':

    22229922882  cycles                    #      0.000 M/sec
    11040488591  instructions              #      0.497 IPC
     7012548051  L1-dcache-loads           #      0.000 M/sec
     1313164727  L1-dcache-load-misses     #      0.000 M/sec

    8.341564469  seconds time elapsed
```

Divide these two to get L1 miss rate

Demo #1: Matrix Multiply (Inner Loop Exchange)

```
intmatrix_multiply_run(const matrix* A, const
matrix* B, matrix* C)

{
inti, j, k;
for (i = 0; i< A->rows; i++) {
for (j = 0; j< B->cols; j++) {
for (k = 0; k< A->cols; k++) {
        C->values[i][j] +=
 A->values[i][k] *
B->values[k][j];
      }
    }
  }
}
```

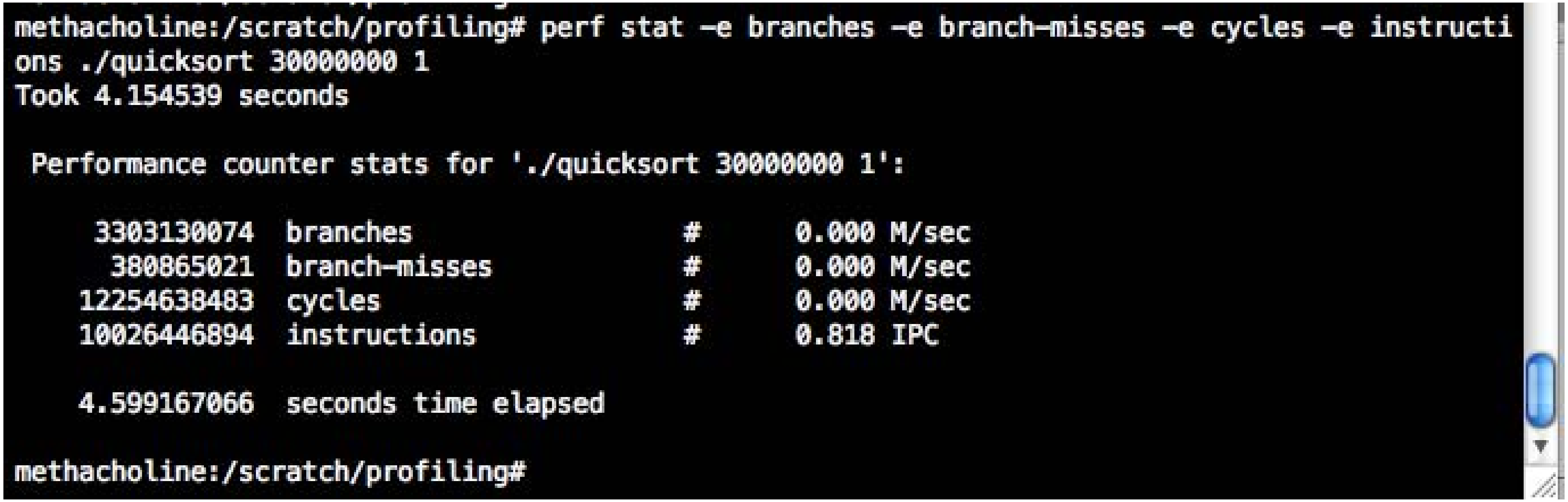
```
intmatrix_multiply_run(const matrix* A, const matrix* B,
matrix* C)

{
inti, j, k;
for (i = 0; i< A->rows; i++) {
for (k = 0; k< A->cols; k++) {
for (j = 0; j< B->cols; j++) {
        C->values[i][j] +=
 A->values[i][k] *
B->values[k][j];
      }
    }
  }
}
```



```
methacholine:/scratch/profiling# perf stat -e cycles -e instructions -e L1-dcache-loads -e L1-
dcache-load-misses ./matrix_multiply
Setup
Running matrix_multiply_run()...
Elapsed execution time: 8.312905 sec

 Performance counter stats for './matrix_multiply':

    22229922882  cycles                    #      0.000 M/sec
    11040488591  instructions              #      0.497 IPC
     7012548051  L1-dcache-loads           #      0.000 M/sec
     1313164727  L1-dcache-load-misses     #      0.000 M/sec

    8.341564469  seconds time elapsed

methacholine:/scratch/profiling# perf stat -e cycles -e instructions -e L1-dcache-loads -e L1-
dcache-load-misses ./matrix_multiply_xchg
Setup
Running matrix_multiply_run()...
Elapsed execution time: 2.577180 sec

 Performance counter stats for './matrix_multiply_xchg':

     6904246362  cycles                    #      0.000 M/sec
    10037693657  instructions              #      1.454 IPC
     6012235277  L1-dcache-loads           #      0.000 M/sec
       63685905  L1-dcache-load-misses     #      0.000 M/sec

    2.590953283  seconds time elapsed

methacholine:/scratch/profiling#
```

Case Study: Sorting & Branching (What the 6.172 Staff Did Yesterday)

- Demo:
  - Using QuickSort to sort 30 million integers



```
methacholine:/scratch/profiling# perf stat -e branches -e branch-misses -e cycles -e instructi
ons ./quicksort 30000000 1
Took 4.154539 seconds

 Performance counter stats for './quicksort 30000000 1':

     3303130074  branches                 #      0.000 M/sec
      380865021  branch-misses            #      0.000 M/sec
    12254638483  cycles                   #      0.000 M/sec
    10026446894  instructions             #      0.818 IPC

    4.599167066  seconds time elapsed

methacholine:/scratch/profiling#
```

Case Study: Sorting & Branching

- Quicksort: pivoting = unpredictable branches:

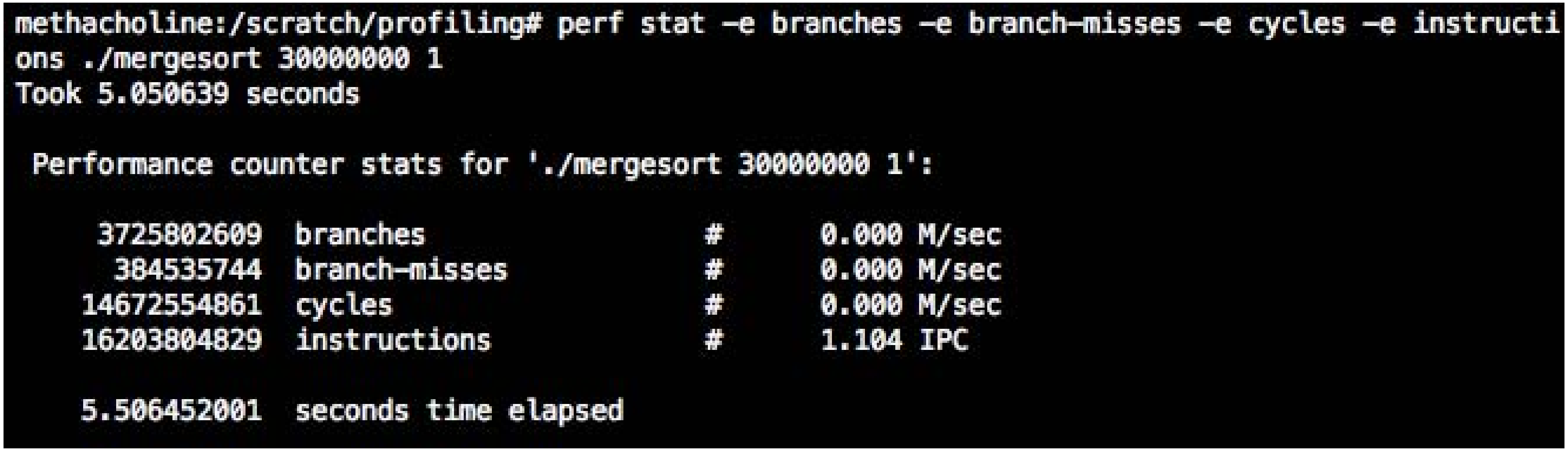
```
while (left < right) {
while (left < right && *left <= pivot) left++;
while (left < right && *right > pivot) right--;
if (left < right) swap(left, right);
  }
```

Case Study: Sorting & Branching

- Let’s try mergesort!

```
staticvoidbranch_merge(long *C, long *A, long *B, ssize_tna, ssize_tnb)
{
while (na>0&&nb>0) {
// We want: *C = min(*A, *B); then increment *A or *B accordingly
if (*A <= *B) {
      *C++ = *A++; na--;
    } else {
      *C++ = *B++; nb--;
    }
  }
while (na>0) {
    *C++ = *A++;
na--;
  }
while (nb>0) {
    *C++ = *B++;
nb--;
  }
}
```

Demo: Profile Mergesort



```
methacholine:/scratch/profiling# perf stat -e branches -e branch-misses -e cycles -e instructi
ons ./mergesort 30000000 1
Took 5.050639 seconds

 Performance counter stats for './mergesort 30000000 1':

     3725802609  branches                  #      0.000 M/sec
      384535744  branch-misses             #      0.000 M/sec
    14672554861  cycles                    #      0.000 M/sec
    16203804829  instructions              #      1.104 IPC

    5.506452001  seconds time elapsed
```

Case Study: Sorting & Branching

- Our mergesort is slower than quicksort!
  - Reason: Still mispredicting branches
- What’s wrong? Caching or Branching?
  - Nehalem vs. Core2: Faster cache; deeper pipeline
    - L1 Hit: ~3-4 cycles; L2 Hit: ~15 cycles
    - Branch Mispredict: ~16-24 cycles
  - Bad branch predictions might be as undesirable as bad memory access patterns
  - Might be worth it to optimize mergesort’s**branching** behavior

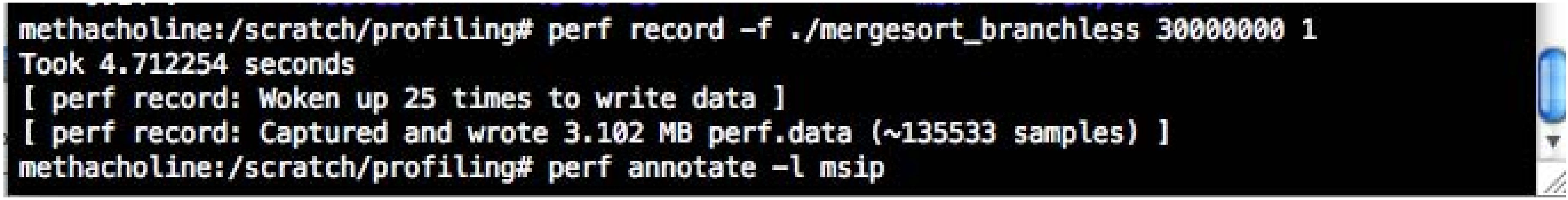
Case Study: Sorting & Branching
Getting rid of mergesort branching:

```
staticvoidbranch_merge(long *C, long *A, long *B,
        ssize_tna, ssize_tnb)
{
while (na>0&&nb>0) {
// We want: *C = min(*A, *B); then increment *A or *B
        accordingly
if (*A <= *B) {
      *C++ = *A++; na--;
    } else {
      *C++ = *B++; nb--;
    }
  }
[…]
}
```

```
staticvoidbranch_merge(long *C, long *A, long *B,
        ssize_tna, ssize_tnb)
{
while (na>0&&nb>0) {
// We want: *C = min(*A, *B); then increment *A or *B
        accordingly
intcmp = (*A <= *B);
long min = *B ^ ((*B ^ *A) & (-cmp));
    *C++ = min;
    A += cmp;
    B += !cmp;
na -= cmp;
nb -= !cmp;
}
[…]
}
```

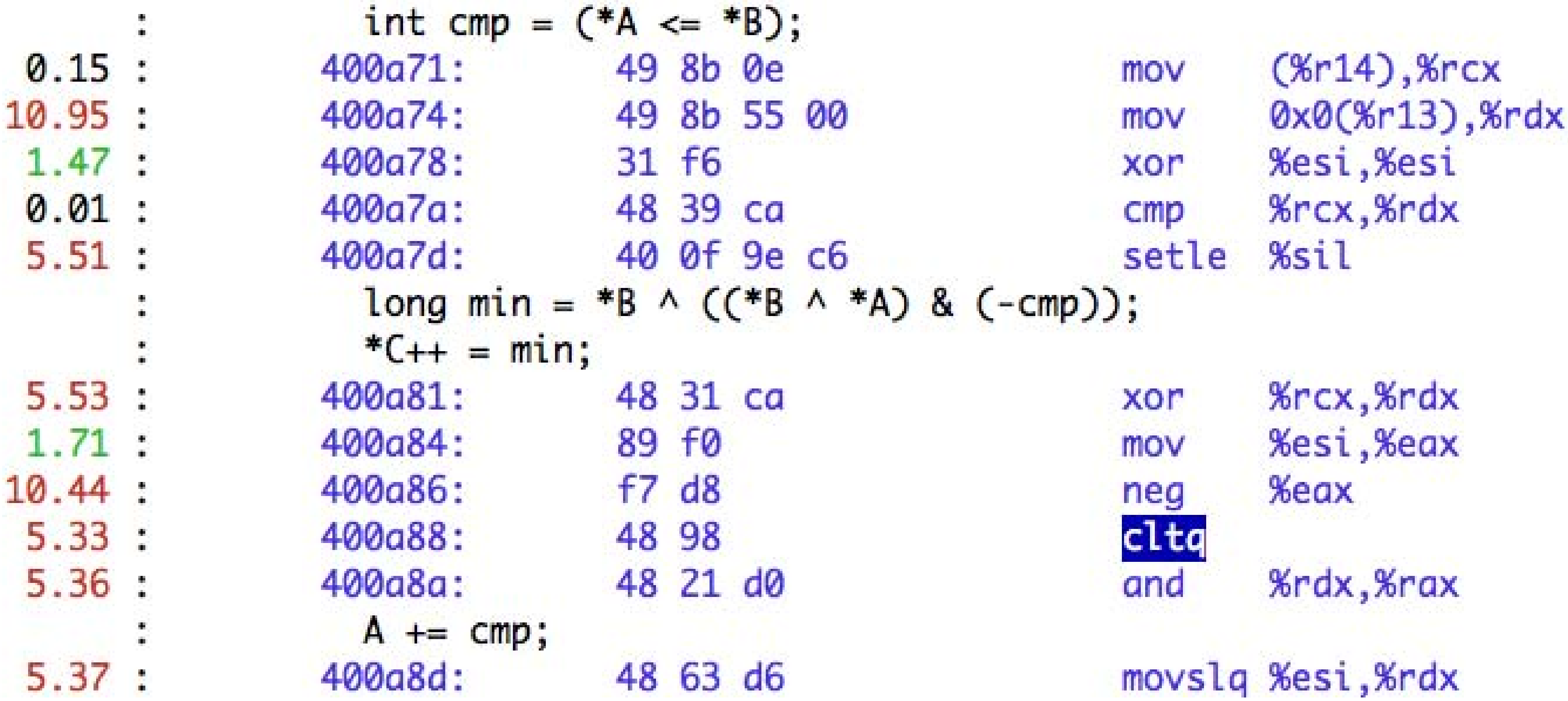
Demo: Profile Branchless Mergesort

- Must record before annotating.
- Annotate takes in function name to annotate around. **msip**was one of the recursive merging functions that called the merge function.



```
methacholine:/scratch/profiling# perf record -f ./mergesort_branchless 30000000 1
Took 4.712254 seconds
[ perf record: Woken up 25 times to write data ]
[ perf record: Captured and wrote 3.102 MB perf.data (~135533 samples) ]
methacholine:/scratch/profiling# perf annotate -l msip
```

Doing Better
(aka: GRR Stupid Compiler!)



```
       :          int cmp = (*A <= *B);
  0.15 :        400a71:       49 8b 0e                   mov    (%r14),%rcx
 10.95 :        400a74:       49 8b 55 00                mov    0x0(%r13),%rdx
  1.47 :        400a78:       31 f6                      xor    %esi,%esi
  0.01 :        400a7a:       48 39 ca                   cmp    %rcx,%rdx
  5.51 :        400a7d:       40 0f 9e c6                setle  %sil
       :          long min = *B ^ ((*B ^ *A) & (-cmp));
       :          *C++ = min;
  5.53 :        400a81:       48 31 ca                   xor    %rcx,%rdx
  1.71 :        400a84:       89 f0                      mov    %esi,%eax
 10.44 :        400a86:       f7 d8                      neg    %eax
  5.33 :        400a88:       48 98                      cltq
  5.36 :        400a8a:       48 21 d0                   and    %rdx,%rax
       :          A += cmp;
  5.37 :        400a8d:       48 63 d6                   movslq %esi,%rdx
```

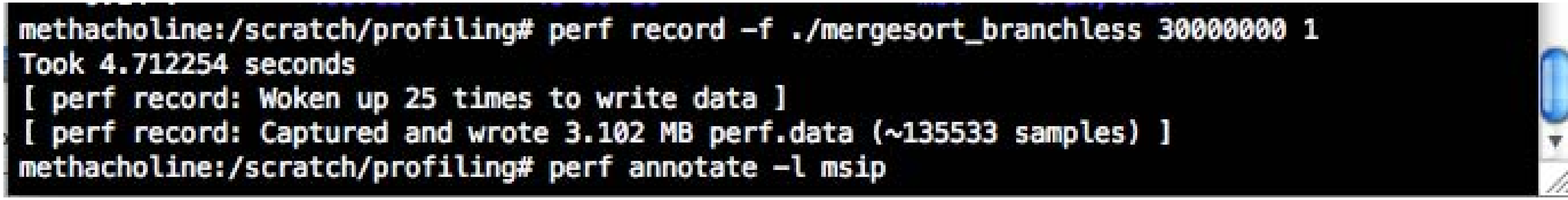
cltq: Sign-extend **%eax** to 64-bits, and place in **%rax**

Doing Better
(aka: GRR Stupid Compiler!)

```
staticvoidbranch_merge(long *C, long *A, long *B,
        ssize_tna, ssize_tnb)
{
while (na>0&&nb>0) {
// We want: *C = min(*A, *B); then increment *A or *B
        accordingly
intcmp = (*A <= *B);
long min = *B ^ ((*B ^ *A) & (-cmp));
    *C++ = min;
    A += cmp;
    B += !cmp;
na -= cmp;
nb -= !cmp;
}
[…]
}
```

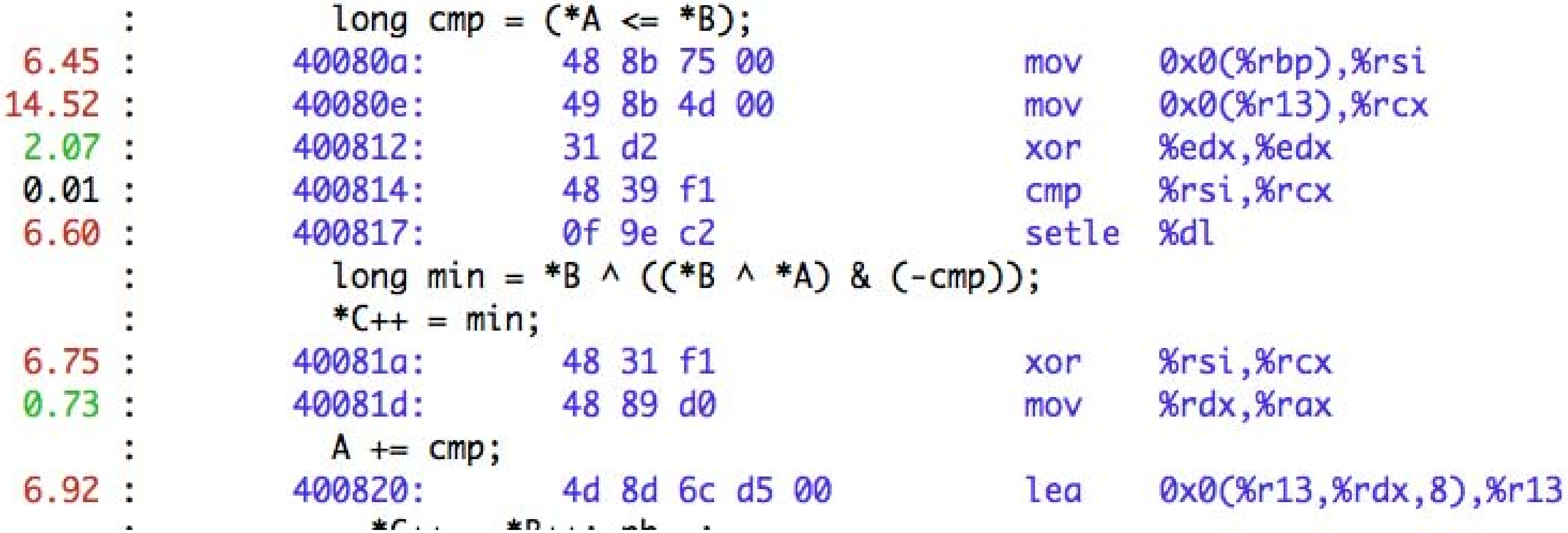
```
staticvoidbranch_merge(long *C, long *A, long *B,
        ssize_tna, ssize_tnb)
{
while (na>0&&nb>0) {
// We want: *C = min(*A, *B); then increment *A or *B
        accordingly
long cmp = (*A <= *B);
long min = *B ^ ((*B ^ *A) & (-cmp));
    *C++ = min;
    A += cmp;
    B += !cmp;
na -= cmp;
nb -= !cmp;
}
[…]
}
```

Demo: Profile Branchless Mergesort: Take 2: (int→ long)



```
methacholine:/scratch/profiling# perf record -f ./mergesort_branchless 30000000 1
Took 4.712254 seconds
[ perf record: Woken up 25 times to write data ]
[ perf record: Captured and wrote 3.102 MB perf.data (~135533 samples) ]
methacholine:/scratch/profiling# perf annotate -l msip
```

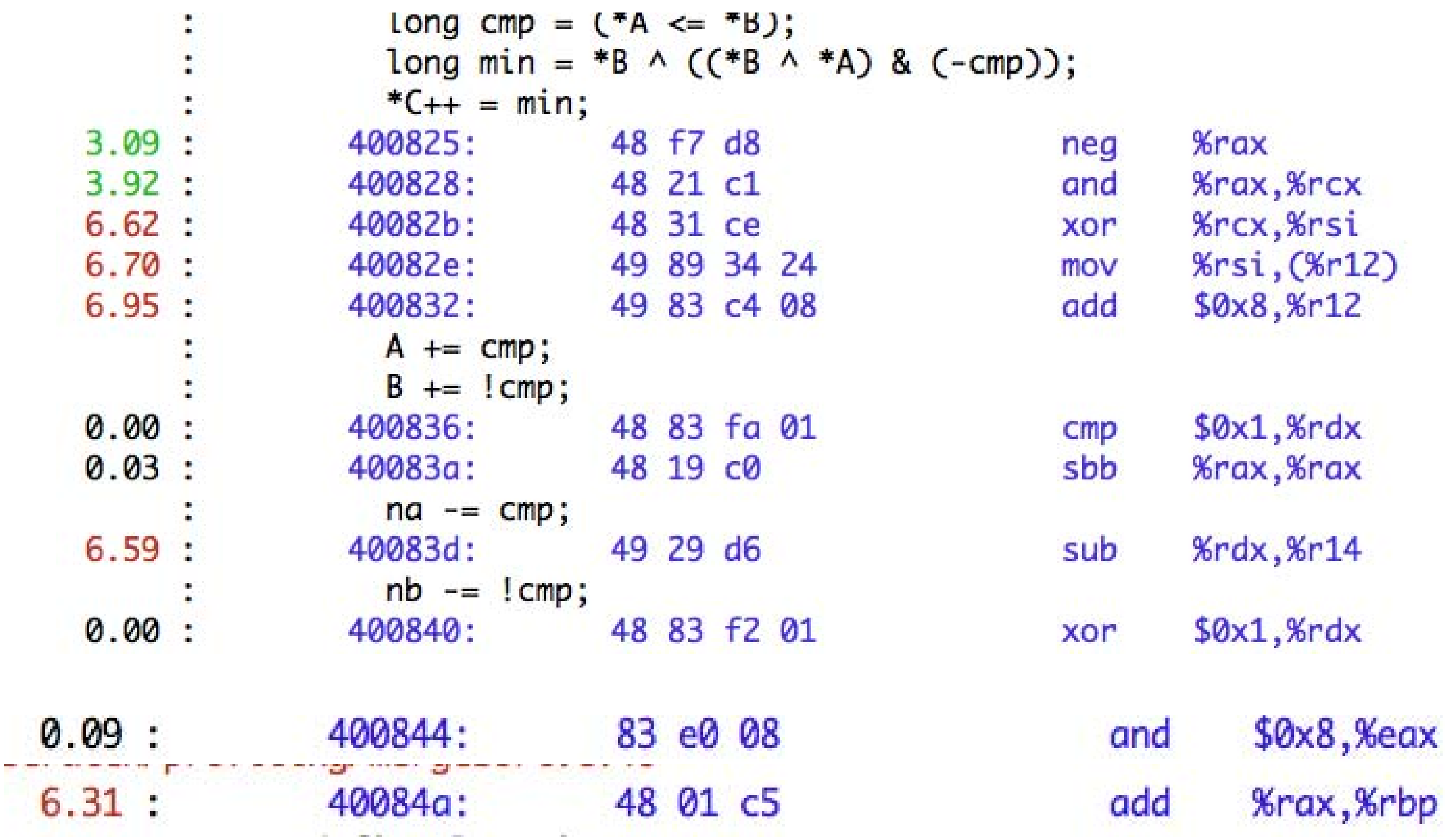
Doing Better
(aka: GRR Stupid Compiler!)



```
      :          long cmp = (*A <= *B);
 6.45 :        40080a:       48 8b 75 00             mov    0x0(%rbp),%rsi
14.52 :        40080e:       49 8b 4d 00             mov    0x0(%r13),%rcx
 2.07 :        400812:       31 d2                   xor    %edx,%edx
 0.01 :        400814:       48 39 f1                cmp    %rsi,%rcx
 6.60 :        400817:       0f 9e c2                setle  %dl
      :          long min = *B ^ ((*B ^ *A) & (-cmp));
      :          *C++ = min;
 6.75 :        40081a:       48 31 f1                xor    %rsi,%rcx
 0.73 :        40081d:       48 89 d0                mov    %rdx,%rax
      :          A += cmp;
 6.92 :        400820:       4d 8d 6c d5 00          lea    0x0(%r13,%rdx,8),%r13
```

BEFORE: 11 instructions
AFTER: 8 instructions

More Compiler Stupidity: Complicated Negations



```
       :          long cmp = (*A <= *B);
       :          long min = *B ^ ((*B ^ *A) & (-cmp));
       :          *C++ = min;
  3.09 :        400825:       48 f7 d8                neg    %rax
  3.92 :        400828:       48 21 c1                and    %rax,%rcx
  6.62 :        40082b:       48 31 ce                xor    %rcx,%rsi
  6.70 :        40082e:       49 89 34 24             mov    %rsi,(%r12)
  6.95 :        400832:       49 83 c4 08             add    $0x8,%r12
       :          A += cmp;
       :          B += !cmp;
  0.00 :        400836:       48 83 fa 01             cmp    $0x1,%rdx
  0.03 :        40083a:       48 19 c0                sbb    %rax,%rax
       :          na -= cmp;
  6.59 :        40083d:       49 29 d6                sub    %rdx,%r14
       :          nb -= !cmp;
  0.00 :        400840:       48 83 f2 01             xor    $0x1,%rdx

  0.09 :        400844:       83 e0 08                and    $0x8,%eax
  6.31 :        40084a:       48 01 c5                add    %rax,%rbp
```

cmp: Stores result to **CF**

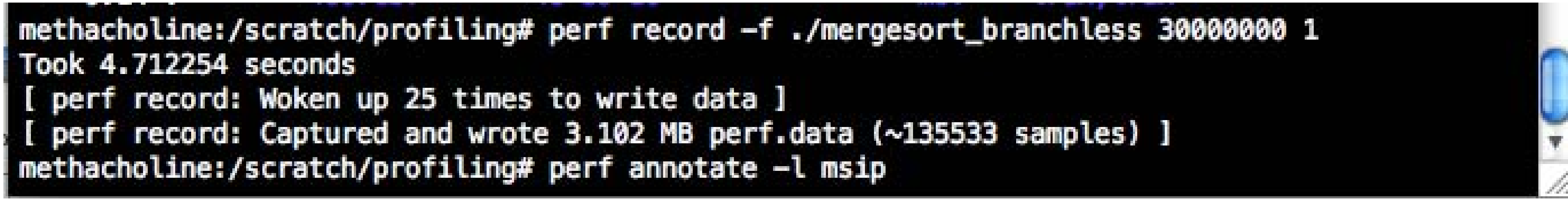
sbb arg1, arg2: arg2 = (arg1 – arg2) - CF

More Compiler Stupidity: Complicated Negations

```
staticvoidbranch_merge(long *C, long *A, long *B,
        ssize_tna, ssize_tnb)
{
while (na>0&&nb>0) {
// We want: *C = min(*A, *B); then increment *A or *B
        accordingly
long cmp = (*A <= *B);
long min = *B ^ ((*B ^ *A) & (-cmp));
    *C++ = min;
    A += cmp;
    B += !cmp;
na -= cmp;
nb -= !cmp;
}
[…]
}
```

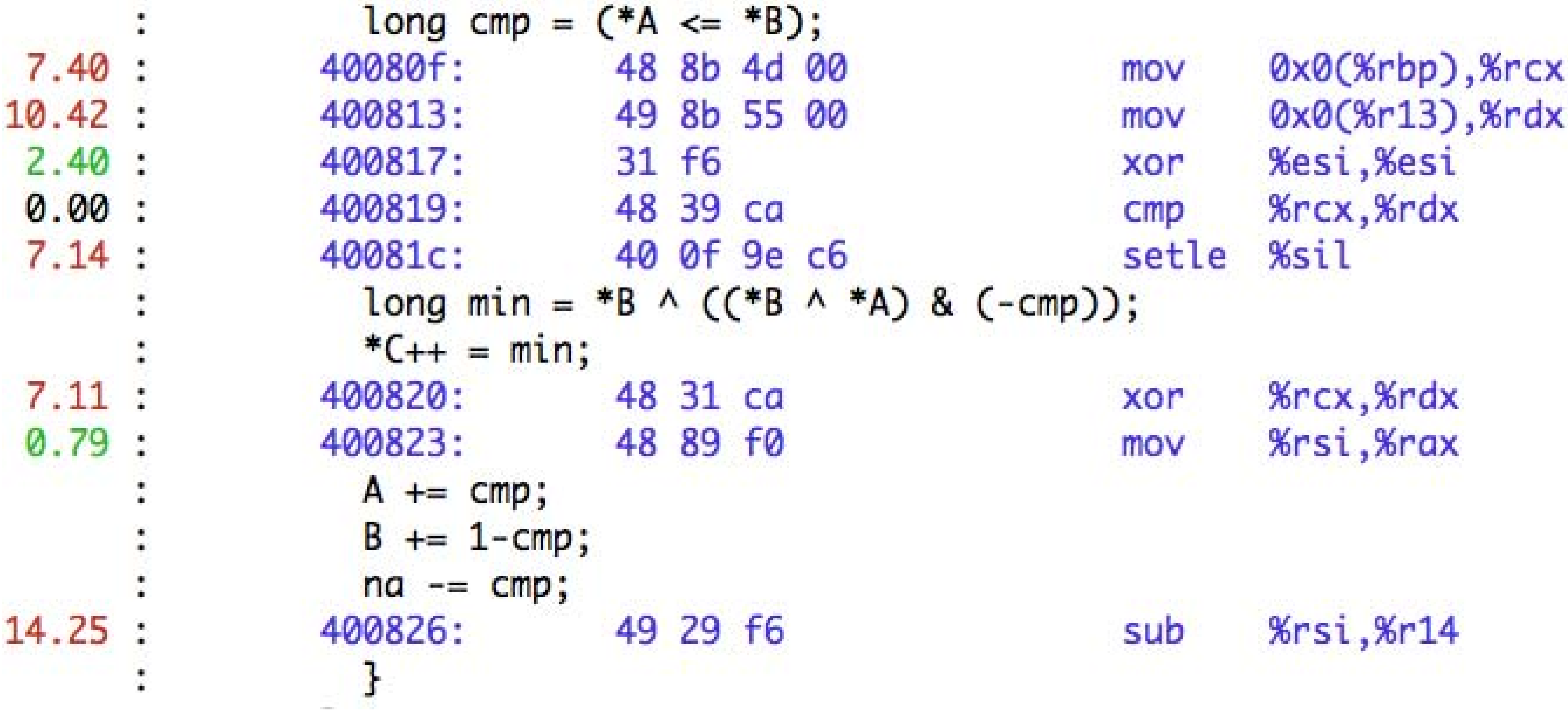
```
staticvoidbranch_merge(long *C, long *A, long *B,
        ssize_tna, ssize_tnb)
{
while (na>0&&nb>0) {
// We want: *C = min(*A, *B); then increment *A or *B
        accordingly
long cmp = (*A <= *B);
long min = *B ^ ((*B ^ *A) & (-cmp));
    *C++ = min;
    A += cmp;
    B += 1-cmp;
na -= cmp;
nb -= 1-cmp;
}
[…]
}
```

Demo: Profile Branchless Mergesort: Take 3: (!cmp→ 1-cmp)



```
methacholine:/scratch/profiling# perf record -f ./mergesort_branchless 30000000 1
Took 4.712254 seconds
[ perf record: Woken up 25 times to write data ]
[ perf record: Captured and wrote 3.102 MB perf.data (~135533 samples) ]
methacholine:/scratch/profiling# perf annotate -l msip
```

More Compiler Stupidity: Complicated Negations



```
        :           long cmp = (*A <= *B);
   7.40 :         40080f:       48 8b 4d 00             mov    0x0(%rbp),%rcx
  10.42 :         400813:       49 8b 55 00             mov    0x0(%r13),%rdx
   2.40 :         400817:       31 f6                   xor    %esi,%esi
   0.00 :         400819:       48 39 ca                cmp    %rcx,%rdx
   7.14 :         40081c:       40 0f 9e c6             setle  %sil
        :           long min = *B ^ ((*B ^ *A) & (-cmp));
        :           *C++ = min;
   7.11 :         400820:       48 31 ca                xor    %rcx,%rdx
   0.79 :         400823:       48 89 f0                mov    %rsi,%rax
        :           A += cmp;
        :           B += 1-cmp;
        :           na -= cmp;
  14.25 :         400826:       49 29 f6                sub    %rsi,%r14
        :           }
```

%sil: Lower byte of **%rsi**

Final **mov** and **sub** have parallelism; fewer “pointless” registers
Fewer ALU ops; Nehalem: only 3 of 6 execution ports have ALUs

Results of Sort Optimizations

Name	Runtime (s)	InsnsPer Clock (IPC)	Branch Miss Rate
QuickSort	4.18	0.813	11.5%
MergeSort	5.04 (+20%)	1.105	10.3%
Branchless Mergesort	4.59 (-8%)	1.762	1.7%
Branchless Mergesort (int→ long)	4.05 (-11.7%)	1.740	1.8%
Branchless Mergesort (!cmp→ 1-cmp)	3.77 (-6.9%)	1.743	1.8%

Overall: **10.8%** Speedup over QuickSort;
33.6% speedup over branching MergeSort

Conclusions

- Profile before you optimize
- Optimize iteratively:
  - Use profiling with intuition
- Look at the annotated assembly
  - Don’t assume the compiler optimizes everything
  - Nudge the compiler in the right direction
- Learn through practice – try these tools yourself (Project 2)