Project 1 – Meeting the Master

• Set up for the design/code review
  – You should contact your Master as soon as you get the enabling email
  – Review Group adjustments must be done by **tonight**

• Prepare for the review!
  – “Opening statement”

• Look for insights to help with the final
  – See if you can come close to the best performance
  – Make sure you understand the sharing policy

• The MITPOSSE is there to help you! They are volunteers. Accord them your greatest respect.
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**

<table>
<thead>
<tr>
<th>In order to do..</th>
<th>You can use...</th>
</tr>
</thead>
<tbody>
<tr>
<td>Manual Instrumentation</td>
<td><code>printf</code>, (or fancy variants thereof)</td>
</tr>
<tr>
<td>Static Instrumentation</td>
<td><code>gprof</code></td>
</tr>
<tr>
<td>Dynamic Instrumentation</td>
<td><code>callgrind, cachegrind, DTrace</code></td>
</tr>
<tr>
<td>Performance Counters</td>
<td><code>oprofile, perf</code></td>
</tr>
<tr>
<td>Heap Profiling</td>
<td><code>massif, google-perftools</code></td>
</tr>
<tr>
<td>Other tools exist for Network, Disk IO, Software-specific, ...</td>
<td></td>
</tr>
<tr>
<td><strong>TODAY:</strong></td>
<td><code>perf</code></td>
</tr>
</tbody>
</table>
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
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: `perfin linux-tools`

https://perf.wiki.kernel.org/
Demo 1: Matrix Multiply

```c
int matrix_multiply_run(const matrix* A, const matrix* B, matrix* C)
{
    int i, 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];
            }
        }
    }
}
```
Divide these two to get L1 miss rate
Demo #1: Matrix Multiply

(Inner Loop Exchange)

```c
int matrix_multiply_run(const matrix* A, const matrix* B, matrix* C)
{
    int i, 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];
            }
        }
    }
}
```
```bash
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':

| Counter                      | Value   | Rate
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>cycles</td>
<td>22229922882</td>
<td>0.000 M/sec</td>
</tr>
<tr>
<td>instructions</td>
<td>11040488591</td>
<td>0.497 IPC</td>
</tr>
<tr>
<td>L1-dcache-loads</td>
<td>7012548051</td>
<td>0.000 M/sec</td>
</tr>
<tr>
<td>L1-dcache-load-misses</td>
<td>1313164727</td>
<td>0.000 M/sec</td>
</tr>
</tbody>
</table>

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':

| Counter                      | Value   | Rate
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>cycles</td>
<td>6904246362</td>
<td>0.000 M/sec</td>
</tr>
<tr>
<td>instructions</td>
<td>10037693657</td>
<td>1.454 IPC</td>
</tr>
<tr>
<td>L1-dcache-loads</td>
<td>6012235277</td>
<td>0.000 M/sec</td>
</tr>
<tr>
<td>L1-dcache-load-misses</td>
<td>63685905</td>
<td>0.000 M/sec</td>
</tr>
</tbody>
</table>

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
Case Study: Sorting & Branching

• Quicksort: pivoting = unpredictable branches:

```c
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!

```c
static void branch_merge(long *C, long *A, long *B, ssize_t na, ssize_t nb)
{
    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 instructions ./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:

```c
static void branch_merge(long *C, long *A, long *B, ssize_t na, ssize_t nb) {
    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--;
        }
    }
    [...]  // Indentation level reduced for readability
}
```
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.
Doing Better
(aka: GRR Stupid Compiler!)

```
int cmp = (*A <= *B);

cltq: Sign-extend %eax to 64-bits, and place in %rax
```
static void branch_merge(long *C, long *A, long *B, ssize_t na, ssize_t nb)
{
    while (na>0&&nb>0) {
        // We want: *C = min(*A, *B); then increment *A or *B accordingly
        int 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)
Doing Better (aka: GRR Stupid Compiler!)

```
long cmp = (*A <= *B);
```

**BEFORE:** 11 instructions

**AFTER:** 8 instructions
More Compiler Stupidity: Complicated Negations

```c
long cmp = (*A <= *B);
long min = *B ^ ((*B ^ *A) & (-cmp));
*C++ = min;
```

**cmp:** Stores result to CF

**sbb arg1, arg2:** `arg2 = (arg1 - arg2) - CF`
More Compiler Stupidity: Complicated Negations

static void branch_merge(long *C, long *A, long *B, ssize_t na, ssize_t nb) {
    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;
    }
    […]
}

static void branch_merge(long *C, long *A, long *B, ssize_t na, ssize_t nb) {
    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 3: (!cmp\rightarrow 1-cmp)
More Compiler Stupidity: Complicated Negations

`%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

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

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)