CS 107
Lecture 15:
Making code fly

Friday, November 17, 2017

Computer Systems
Fall 2017
Stanford University
Computer Science Department

Lecturers: Julie Zelenski and Chris Gregg
Today’s topics

- **Compiler optimization**
  What optimizing compilers do and don’t do

- **Memory performance**
  How memory is organized
  Impact of temporal and spatial locality

- **Profiling tools**
  Measuring runtime and memory performance
“We should forget about small efficiencies, say about 97% of the time: premature optimization is the root of all evil.”
—— Donald Knuth

“More computing sins are committed in the name of efficiency (without necessarily achieving it) than for any other single reason - including blind stupidity.”
—— W.A. Wulf

“Bottlenecks occur in surprising places, so don't try to second guess and put in a speed hack until you have proven that's where the bottleneck is.”
—— Rob Pike
Rules of thumb

- **Is there even a problem?**
  Measure! If ok at expected scale, you’re done!

- **KISS (keep it simple stupid)**
  If low-traffic/small input: simplest code, easy to understand and debug
  (optimize use of programmer's time!)

- **Choose correct algorithm/design**
  Optimization reduces constants, doesn’t change Big-O or fix bad design

- **Let gcc do its magic!**
  No pre-optimize, don’t get in compiler’s way
  Read generated assembly to know what you are getting

- **Only then take action of your own**
  Measure again, attend only to actual bottleneck
Let's see what gcc does!

/afs/ir/class/cs107/samples/lect15

code.c

GCC explorer https://godbolt.org/g/F8x1Uc
-00 // faithful/literal match to C, best for debugging
-0g // streamlined, but debug-friendly
-02 // apply all acceptable optimizations

Compiler knows the score when it comes to the hardware
  Register allocation
  Instruction choice
  Alignment

Transformations should be legal, equivalent
  Compiler has only knowledge of CT, not RT
  Operates conservatively
  "Do no harm"
How much help is gcc?

- **How do we measure performance?**
  - System timers, rtdsc!
  - Profiling tools (valgrind --tool=callgrind)

- **How does gcc do?**
  - Time unoptimized vs optimized: mult.c sorts.c fact.c

- **What can gcc not do for you?**
  - Cannot fix algorithmic weakness, big-O

- **What can block/prevent gcc optimization?**
  - Pre-optimizing, function calls, use of pointers
Instruction mix

Figure 5.12 B&O

<table>
<thead>
<tr>
<th>Operation</th>
<th>Latency</th>
<th>Issue</th>
<th>Capacity</th>
</tr>
</thead>
<tbody>
<tr>
<td>Integer add</td>
<td>1</td>
<td>1</td>
<td>4</td>
</tr>
<tr>
<td>Integer mult</td>
<td>3</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>Integer divide</td>
<td>3-30</td>
<td>3-30</td>
<td>1</td>
</tr>
</tbody>
</table>

What instruction was most common? (Remember lab6?)

How do we make that instruction fast?

Mov between registers is fast, but accessing memory... slow
myth Core 2 Duo

CPU, Reg

L1 I-cache, 32 KB

L1 D-cache

~4 MB, L2 unified cache

~4 GB, Main Memory

~500 GB, Disk

Throughput: 16 B/cycle, 8 B/cycle, 2 B/cycle, 1 B/30 cycles
Latency: 3 cycles, 14 cycles, 100 cycles, millions
Location, location, location

All of caching relies on locality!

Temporal locality
Repeat access to same data tends to be co-located in TIME
Things I have used recently, I am likely to use again soon

Spatial locality
Related data tends to be co-located in SPACE
Data that is near a used item is more likely to also be accessed

Realistic scenario:
97% cache hit rate
Cache hit costs 1 cycle
Cache miss costs 100 cycles
How much of your memory access time spent on 3% of accesses that are cache misses?
Run program under callgrind, creates file callgrind.out.pid
valgrind --tool=callgrind ./array_opt
Process file to see source annotated with count per line
  callgrind_annotate --auto=yes callgrind.out.<pid>