Applied Algorithms Lec 3: Optimization

Sam McCauley

September 13, 2024

Williams College

Admin

- Highly relevant talk during colloquium today (right after class)
- Reminder: this is *Homework* 1 (one or two places old versions of the class have slipped through; should be fixed). You can collaborate on code; questions should be your own
- Homework 1 (should be) fully available; we'll talk about momentarily

How to do Homework/Assignments

Basics of submission

- Each of you have a git repo
- Your code is automatically run twice a day on TCL312 machines
- Feedback from automatic runs given back to you in `feedback/` directory
- Also some thought questions in `handout/` directory; please answer in the latex
- Grading is anonymous (I use a local script) (This also means you shouldn't put your name in your code or latex)

Handing in Assignments document

Accessing lab computers

- Ssh into one of the listed lab computers, or work in person
- Use software you’re comfortable with. But I’m happy to help if you’re learning something. (vim and/or emacs are something you should all probably know the basics of at some point.)
- Whatever editor you want; VSCode is widely recommended
- Use Remote-SSH on VSCode for best of both worlds
  - You should almost certainly use software with syntax highlighting and some automatic indentation. Nowadays it’s best if you use software with an LSP (or something similar) that tells you about errors immediately.
- Working in person on lab machines mean you can use a GUI application

How to work on Homework/Assignments

- Work in the repo I gave you
- Don’t alter `test.c`; only alter `Makefile` if you want to (or need to)
- Really it’s best to work on lab computers; but not 100% necessary

Accessing lab computers from off campus

- Cannot ssh into the lab computers directly from off-campus!
    - First ssh into `lohani.` From lohani, ssh into lab computers.
    - This double ssh is a bit of a pain, and may cause issues with things like syntax highlighting; has anyone done this with VSCode?
    - Working on your computer; pushing and running on lab computers; is always an option.

Homework 1

- Code can be done collaboratively if you cite
- Question should be done on your own
- We’ll talk about experiments in a bit: this question is fairly open-ended, but the important part is thinking about what goes into an experiment
    - Don’t need to generate your own data
- Questions 3a and 3b are intended to only have short answers. 3b in particular is mostly a hint for 3c.
- Homeworks and assignments are meant to challenge everyone across a fairly broad set of areas!
    - May be particularly difficult when not in your wheelhouse
    - Talk to me; remember that you’re not expected to get all As on every homework or assignment

Automatic Runs

- Twice a day
- Uses different files than those available to you
    - Pros/cons of this
    - Debugging without full information can be important, but also can be frustrating
    - I’m happy to help; talk to me if something seems insurmountable

Feedback

- Each run generates a timestamped text file with information about what happened
- Also have an .html file summarizing. (Can open in the browser and it should look reasonably nice)
- The HTML feedback file also contains your ID for the leaderboard
- Let’s look at that too

Homework questions

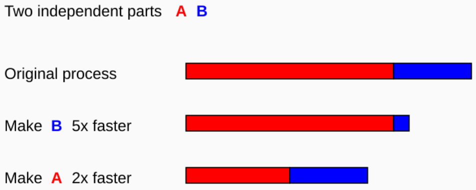
- I'll give you the latex under `handout/` in your repo; answer the questions there
- Can upload to overleaf if you want
- Honestly I'll probably grade it just by reading the latex; I'll only compile it if I can't read it. So don't worry too much about it being perfect.

Optimization

Thought question

- What part of a program is most important to speed up?
- Let's say I have several functions. How can I choose which to try to optimize first?
- Answer: the one that takes the most *total* time
    - Time it takes $\times$ number of times it's called
    - May not be the slowest function—in fact, it's often a very fast but very frequently-used function
- Probably need to take into account potential to speed it up as well—I want the function that takes up the most time that I can save.

Amdahl's Law



If a function takes up a p fraction of the entire program's runtime, and you speed it up by a factor s, then the overall program speeds up by a factor

$$\frac{1}{1-p+p/s}$$

Amdahl’s Law and Asymptotics

- Can *estimate* the total time of an algorithm asymptotically

- Example: Where to improve Dijkstra’s algorithm?

Dijkstra’s Algorithm

```
function Dijkstra(Graph, source):
    create vertex set Q
    for each vertex v in Graph:
        dist[v] ← INFINITY
        prev[v] ← UNDEFINED
        add v to Q
    dist[source] ← 0
    while Q is not empty:
        u ← vertex in Q with min dist[u]
        remove u from Q
        for each neighbor v of u still in Q:
            alt ← dist[u] + length(u, v)
            if alt < dist[v]:
                dist[v] ← alt
                prev[v] ← u
    return dist[], prev[]
```

Dijkstra’s Algorithm

```
function Dijkstra(Graph, source):
    while Q is not empty:
        u ← vertex in Q with min dist[u]
        remove u from Q
        for each neighbor v of u still in Q:
            alt ← dist[u] + length(u, v)
            if alt < dist[v]:
                dist[v] ← alt
                prev[v] ← u
    return dist[], prev[]
```

The inner for loop (blue part) is, at first glance, by far the most important part to optimize.

Measuring Performance

What units to measure time?

What units to measure time?

- Overall: CPU time
    - Some idiosyncracies in how we’re measuring it
    - CPU vs wall clock time shouldn’t make much difference for us
    - Parallelism doesn’t help
- Costs of specific operations are sometimes given using number of “CPU cycles” (we’ll come back to this in a second)
- Not-really-accurate-anymore definition of a cycle: time to perform one basic operation

Easiest way to measure time: just time it using built-in tools!

Easy, probably reflective of what you want.

But some things to bear in mind:

- Make sure your timing is macroscopic.
  - No timing is exact.
  - CPU clocks usually only have a resolution of $\approx$ 1 million ticks per second (sometimes less)
  - Minimize issues with overhead, external factors
  - Rule of thumb: ideally an experiment will take $\approx$ 1 second
  - Always repeat several times and check consistency
- Let's look at how `test.c` times your code on Homework 1
- Can also use unix `time` function

Timing one portion of your code

For Amdahl’s, we want to time the total time a subroutine takes over all calls. How can we hope to do that if each call is very fast?

1. One option: factor out subroutine using separate testing code
    - Need to get info on how often it’s called; simulate correct types of data.
    - Make sure the compiler does not optimize out your whole experiment!
2. Another option: Run same code with and without subroutine
    - Does that change the data the function is called with? Will the change in data affect running time?
3. Profiling!

We’ll come back to this with some examples later today. Bear in mind: benchmarking itself is an entire area of computer science.

What is Slow on Modern Computers? (Rules of Thumb)

Note on time taken

In the last couple years, Intel has stopped releasing this information!

- Too much else going on for strong conclusions.
- I’ll go over the numbers from a couple years ago anyway; some (very high level) lessons to be learned
- To *know* if something is fast: run an experiment!

Basic operations (latency)

- Integer add, multiply (bit operations, move, push, pop, etc.)
    - fast! 1-2 cycles
- Divide, modulo
    - Pretty slow; 5-20 cycles
- Float add, multiply?
    - Pretty fast on x86; almost as fast as integers

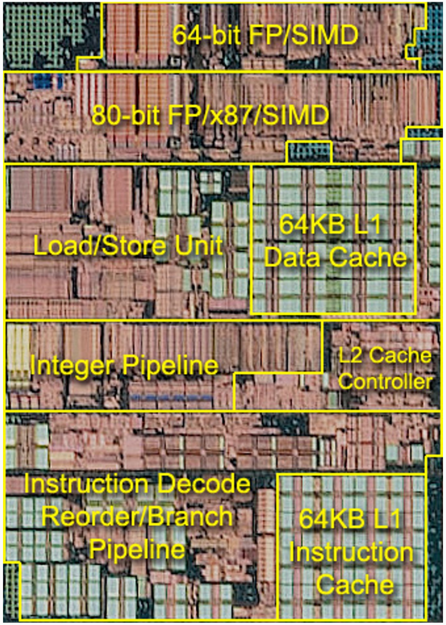
Experiments

- Let's run some (really rough) experiments: `timetests.c`
- Unroll loops to minimize loop overhead; compile with optimizations off
- Why is this important? Let's look at the assembly
- Compiler explorer: recent, *super cool* tool to look at assembly for C code
  - `godbolt.org`
  - Awesome for people (like me) who aren't assembly experts but sometimes care about what exactly the computer is doing

More complicated operations

- Square root?
    - fast on our machines! 1-2 cycles
- memory allocation in bytes? `timetests2.c`
- memory allocation in megabytes?
- how does it grow as we increase the number of operations?
    - Cache efficiency is the problem here, not the memory call itself
    - (For what it's worth: `malloc` really is $O(1)$)

Modern processors



- Lots going on
- Moving things around takes more time than processing

Casts and moving data around

- Casts can be expensive if they require moving the data into another part of the processor!
- (Can be free if they don’t)

Branch mispredictions, etc.

- Instructions need to be moved into the CPU
- Modern CPUs predict what instructions will be next; move while completing other operations
- What if the CPU gets it wrong?
- “Branch misprediction:” 10-20 cycles to fetch the new instructions from memory
- Can have similar issues with calling non-inlined functions (compiler is very good at avoiding this)

Branch predictors

- CPU keeps track of your branches as it runs
    - Divides into four categories of how likely it is to be taken

- `gcc` also predicts your branches during compilation

- Can also give preprocessor directives about branches. Can be helpful (one of the *last* things you should do for optimization)

Avoiding branch mispredictions

```
int max(int a, int b) {
    int diff = a - b;
    int dsgn = diff >> 31;
    return a - (diff & dsgn);
}
```

```
int swap(int a, int b) {
    a = a ^ b;
    b = a ^ b;
    a = a ^ b;
}
```

- Avoid branches (ifs, etc.) by refactoring when possible
- Crazy tricks often not worth it nowadays—true in general; though some exceptions (again, check this only at the very end of optimizing; only for crucial operations)

Avoiding branch mispredictions

```
int max(int a, int b) {
    int diff = a - b;
    int dsgn = diff >> 31;
    return a - (diff & dsgn);
}
```

```
int swap(int a, int b) {
    a = a ^ b;
    b = a ^ b;
    a = a ^ b;
}
```

- cmov operations help a lot in modern processors; compilers are great at avoiding expensive branches
- If you do create a branch, ask yourself how easy it is to predict!
- Only way to be sure is to experiment
- `branchpredictions.c`

Code Profiling

Profiling code

- Why not just have your computer tell you what functions are caused the most, or keep track of how long they run, or monitor specific high-cost operations?
- Lots of such tools! We'll look at a couple of them right now, and use them throughout the class.
  - `gprof`
  - `cachegrind`
  - We won't use `perf` but some people like it
  - We won't use Intel VTune either but seems very cool and powerful
- What do you think some advantages and disadvantages are of using profiling software?

`gprof`

- Older command line tool
- Uses sampling to collect data
- Designed to talk with `gcc` using `-pg` flag
- Gives information about time as well as the call graph
- *Quite* limited. But in some circumstances gives good advice.
    - Recursion; function-level resolution; cannot optimize; overhead; sampling problems
- We’ll look at some examples later

`callgrind` and `cachegrind`

- Features of `valgrind`
- `callgrind` gives `gprof`-like profiling
- `cachegrind` helps determine the cost of *moving* data: cache misses, branch mispredictions, etc.
- Essentially runs the program on a virtual machine
- Gives information about costs you could not otherwise get, but VERY slow.

Costs of Computation

Note on time taken

Latency vs throughput:

- Latency: time it takes for a sequence of *data-dependent* operations of a given type
- Throughput: time after a previous operation when a new operation of the same type can begin.
- Let's look at an example: `latencythroughput2.c`

Designing Experiments

- As said before: make sure macroscopic time
- Try to avoid loop overhead when possible
- Be careful that the code *actually does* what you're testing
- Careful: need to make sure the compiler does not optimize out your test!
- Compiler explorer; or `gcc -S --verbose-asm`
- Let's try the `gcc` method on `latencythroughput1.c`

Profilers examples: gprof

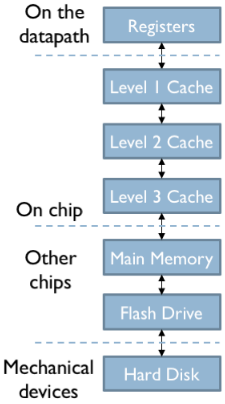
- Compile with `-pg` option; then run normally; then run gprof on the executable
- Gives information about what calls what and how much time is in each
- Not perfect, but gives us some information, especially for simpler programs
  - Can see if one function is called a LOT
  - Can see if one function is only ever called by one other function
- Gets confusing with recursive calls
- I may ask you to use this, but be aware that it’s useful sometimes at best

Final major cost: cache misses!

- Data is stored in different places on the computer
- Cost to access the data frequently dominates running time

A Typical Memory Hierarchy

- Everything is a cache for something else…



Access time	Capacity	Managed By
1 cycle	1 KB	Software/Compiler
2-4 cycles	32 KB	Hardware
10 cycles	256 KB	Hardware
40 cycles	10 MB	Hardware
200 cycles	10 GB	Software/OS
10-100us	100 GB	Software/OS
10ms	1 TB	Software/OS

How caches work

- Stores data in the optimal(ish) place
- Moves data around in *cache lines* of $\approx$ 64 bytes
- Modern caches are very complicated
- Can be advantages of adjacent cache lines
- Basically: close is good; recent is good; jumping around is bad.
- Example: `sortedlinkedlist.c` `unsortedlinkedlist.c`

Profilers examples: cachegrind

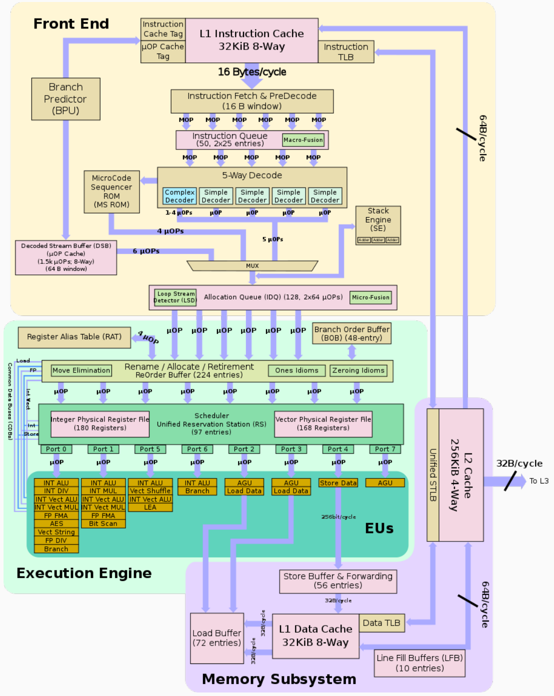
- Compile with debugging info on `-g` AND optimizations on
  - What does this entail immediately?
- Then `valgrind --tool=cachegrind [your program]`
- Outputs number of cache misses for instructions, then data, then combined
- Simulates a simple cache (based on your machine) with separate L1 caches for instructions and data, and unified L2 and (if on machine) L3 caches
- Does L1 misses vs last level (L3) misses
- Virtual machine: not 100% accurate; *slow*

Reading cachegrind output

- I, I1, LLi, etc.: *instruction* misses
- D, D1: first level of cache
- LL: last layer of cache
- Run `cg_annotate cachegrind.out.118717` (the last number will change based on which cachegrind run you are referring to) for function-by-function and line-by-line stats
    - Extremely wide output; probably want to pipe to a file
- Let's look at `sortedlinkedlist.c` and `unsortedlinkedlist.c` again

Real-World Caching

- We looked at simple, constructed examples where caching is easy to reason about
- But: bear in mind that modern caches are very complicated; interact nontrivially with other costs (branch mispredictions; expensive operations; etc.)
- I should at least mention *prefetching*: if your computer thinks it can get a head start on fetching your data, it will
- Model things the best you can, but always use experiments when you’re not sure


Front End
Instruction Cache Tag
µOP Cache Tag
L1 Instruction Cache 32KiB 8-Way
Instruction TLB
16 Bytes/cycle
Branch Predictor (BPU)
Instruction Fetch & PreDecode (16 B window)
MOP MOP MOP MOP MOP MOP
Instruction Queue (50, 2x25 entries)
Macro-Fusion
MOP MOP MOP MOP MOP
MicroCode Sequencer ROM (MS ROM)
5-Way Decode
Complex Decoder
Simple Decoder
Simple Decoder
Simple Decoder
Simple Decoder
1-4 µOPs
µOP µOP µOP µOP
Stack Engine (SE)
Adder Adder Adder
4 µOPs
5 µOPs
Decoded Stream Buffer (DSB) (µOP Cache) (1.5k µOPs; 8-Way) (64 B window)
6 µOPs
MUX
64B/cycle
Loop Stream Detector (LSD)
Allocation Queue (IDQ) (128, 2x64 µOPs)
Micro-Fusion
Register Alias Table (RAT)
4 µOP
µOP µOP µOP µOP µOP µOP
Branch Order Buffer (BOB) (48-entry)
Load
FP
Common Data Buses (CDBs)
Int Vect
Int
Store
Move Elimination
Rename / Allocate / Retirement ReOrder Buffer (224 entries)
Ones Idioms
Zeroing Idioms
µOP µOP µOP µOP µOP µOP µOP µOP
Integer Physical Register File (180 Registers)
Scheduler Unified Reservation Station (RS) (97 entries)
Vector Physical Register File (168 Registers)
Port 0 Port 1 Port 5 Port 6 Port 2 Port 3 Port 4 Port 7
µOP µOP µOP µOP µOP µOP µOP µOP
INT ALU, INT DIV, INT Vect ALU, INT Vect MUL, FP FMA, AES, Vect String, FP DIV, Branch
INT ALU, INT MUL, INT Vect ALU, INT Vect MUL, FP FMA, Bit Scan
INT ALU, Vect Shuffle, INT Vect ALU, LEA
INT ALU, Branch
AGU, Load Data
AGU, Load Data
Store Data
AGU
256bit/cycle
EUs
Execution Engine
Unified STLB
L2 Cache 256KiB 4-Way
32B/cycle
To L3
Store Buffer & Forwarding (56 entries)
32B/cycle
Load Buffer (72 entries)
32B/cycle
32B/cycle
L1 Data Cache 32KiB 8-Way
Data TLB
64B/cycle
Line Fill Buffers (LFB) (10 entries)
Memory Subsystem

Conclusions

- Different places where we can incur costs:
    - Operations
    - Branches and moving around instructions
    - Cache misses
- Determining costs is a matter of experimentation on modern machines!
    - Rarely perfect!
- Theme throughout class: design different experiments to test different aspects of code performance.