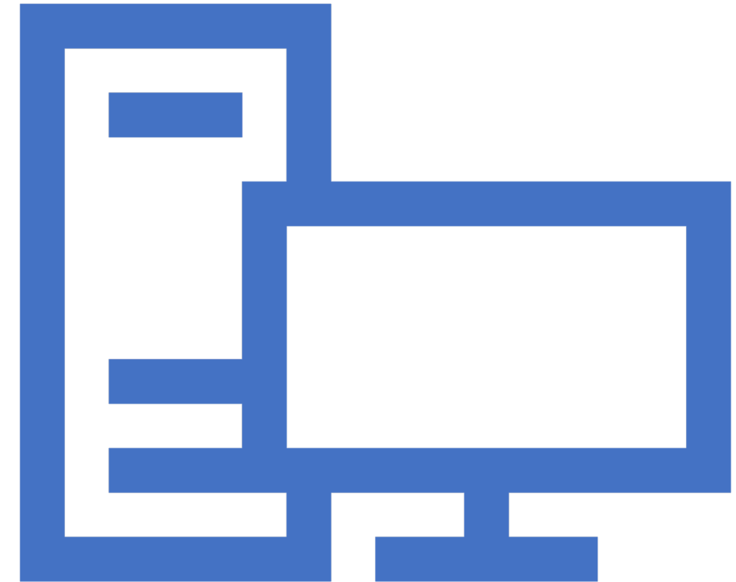
Applied Algorithms

Lecture 7: Profiling, pipelining, and loop optimization



Admin

- Assignment 2 typo fixed on site (want O(m) space)
- Mini-Midterm out tomorrow
  - Works like assignment
  - Individual work
  - No leaderboard
- I will spend time on Thursday (quickly) explaining the algorithm

Admin

- Graded Assignment 1 out by 3PM today (I just need to push)
- Comments appear inline in your Assignment1.tex
- Just a heads up: I am doing anonymous grading
  - Using a script to open all your files at once in random order
  - Plus hacking vim to not display the file name

Today

- Finish Assgn 1 discussion from last class
- Piplining and loop unrolling/optimization
- Profilers: how to see what's slowing down your code

Sorting is now the entire (two towers) problem

- qsort() is pretty slow
- Why?
  - Backend problem: C cannot inline the comparison function
  - Need to set up a function on the call stack for every comparison

- Three better options:
  - Make your own sort
  - Call C++’s sort
  - Call a library with a good sort (like timsort)

Make your own sort

- Not too hard to beat qsort by ~10-20%

- Quicksort implementation, switch to insertion sort if problem size is small
  - Why?
  - Constants
  - Nearly-sorted data

Calling C++

- C++ has std::sort()
- It's just a good quicksort implementation
  - Optimized more than you likely have time to do
  - CAN inline comparisons! (due to improved backend)
  - About 10x faster than qsort() for simple types
- Pretty easy to call C++ from C code
  - Do need to compile with g++ though
- (Please don't go crazy with this; make sure your code is readable in C)

Call a library

- Not many “official” sorting libraries for C
  - I don’t know why
- Only two submissions got this working
- Some libraries have fancy sorts, like timsort

std::sort()

- Quicksorts large data
- Switches to insertion sort after a certain point

- Also: detects poor pivot performance, switches to another sort if things are going badly (should not happen for you!)

- Pivot selection is implementation-dependent so far as I can tell
  - Often median-of-3

Timsort

- More recent sorting method
- On sufficiently small arrays, timsort does insertion sort

- Info about what it does otherwise today if we have time
  - Short answer is: optimizes if data is nearly-already-sorted

Final optimizations

- We are searching for the optimal smaller tower
- Idea: instead, search for the tower closest to the target (above or below) that contains the first item
  - Advantage: one table becomes half as big
  - Disadvantage: binary search needs to be “closest” instead of predecessor
  - This might cost an extra “if” statement
    - Why do I care about that?

Final optimizations

- Second idea: the tables have a non-obvious pattern
  - First and second halves are in same permutation
  - Can take advantage of this to greatly reduce sort time

Assignment 1 questions

- Applied **Algorithms**
- Spend time on the questions—they're about half of the course!
- Why do I care?
  - In the course title/it's something I like
  - This course is about using algorithms to write good/fast code
  - Correctness!!!!!!!!!!!!

- Very few people got the questions right on Assignment 1
- I was generous with partial credit

Problem 2: Find k best solutions

How to think about algorithmic problems like this?

- Bad way: “I already have an algorithm for the best solution. What can I add to it to get the k best?”
- Better: “In my current algorithm, what guarantees that my answer is the best? How can I extend that guarantee to provide the k best?

Problem 2: Find k best solutions

Why the algorithm works:

- There exists some optimal shorter tower—the one closest to half height without going over it. This tower must have some elements from the first half of the input, and some elements from the second half of the input.
- Our algorithm iterates through every possible “first half” subset, and finds the best possible second half. This always finds the solution.

Problem 2: Find k best solutions

Extend it to k:

- There exists some optimal k shorter towers—the ones closest to half height without going over it. These tower must have some elements from the first half of the input, and some elements from the second half of the input.
- Our algorithm iterates through every possible “first half” subset, and finds the best possible k second halves. If the final solution is one of the k best, it MUST be one of the k best for some first-half subset

Problem 2: Find k best solutions

- That is to say: we cannot look through the solutions as usual and keep the k best. We need to find the k best for EACH subset we search, and merge them with the k best so far.

- How can we do this?

- What is the final running time?
  - Comment: if I do not state that a variable is a constant, you should not treat it as one. (I would like you to parameterize by k here-but I didn't take off)

Problem 3: Balanced towers

The bottom two blocks must be one apart.

Again, break it down:

- The best (smaller) balanced tower must consist of some blocks from the first half, and some blocks from the second half. AND the largest block from the second half must be within 1 of the largest from the first half.

Problem 3: Balanced towers

Why won’t the following work?

- Iterate through each first half and find the second half closest to the target. If the second half is balanced with the first, check if it is a better solution. Otherwise, throw it out.

- Underspecified. (What does “throw it out” mean? Do we keep the same first half, or move on to the next one?)
- Might not find the best/might take a long time to find the best.

Problem 3: Balanced towers

“Algorithmic” strategy:

- The smaller tower either has the largest block in it; or it doesn’t have the largest block in it.
- Can we solve these two cases?
- Assume without loss of generality that the largest block is in the first half of the input (can just reverse the input if not true)

Problem 3: Balanced towers

Case 1 (smaller tower does not have largest block):

- There needs to be at least one block in the smaller tower within 1 of largest
- Build two tables: Table1 only contains subsets which have at least one block within 1 of largest. Table2 contains all subsets.
- For each first-half subset:
  - If the subset contains no block within 1 of largest, search in Table 1
  - If it does have a block within 1 of largest, search in Table 2

Problem 3: Balanced towers

Case 2 (smaller tower does have largest block):

- There needs to be at least one block in the larger tower within 1 of largest (must be *excluded* from our small-tower solution)
- Build two tables: Table1 only contains subsets which are *missing* at least one block within 1 of largest. Table2 contains all subsets.
- For each first-half subset:
  - If the subset contains all possible blocks within 1 of largest, search in Table 1
  - If it is missing a block within 1 of largest, search in Table 2

Problem 3: Balanced towers (Running time)

Case 1 (smaller tower does not have largest block):

- There needs to be at least one block in the smaller tower within 1 of largest
- Build two tables: Table1 only contains subsets which have at least one block within 1 of largest. Table2 contains all subsets.
- For each first-half subset:
  - If the subset contains no block within 1 of largest, se… in Table 1
  - If it does have a block within 1 of largest, search in Tab…

$O(n\ 2^{n/2})$ (just like before!)

Problem 3: Balanced towers (Running time)

Case 1 (smaller tower does not have largest block):

- There needs to be at least one block in the smaller tower within 1 of largest
- Build two tables: Table1 only contains subsets which have at least one block within 1 of largest. Table2 contains all subsets.
- For each first-half subset:
  - If the subset contains no block within 1 of largest, search in Table 1
  - If it does have a block within 1 of largest, search in Table 2

Can check in O(n) time per subset

Problem 3: Balanced towers (Running time)

Case 2 (smaller tower does have largest block):

- There needs to be at least one block in the larger tower within 1 of largest (must be *excluded* from our small-tower solution)
- Build two tables: Table1 only contains subsets which are *missing* at least one block within 1 of largest. Table2 contains all subsets.
- For each first-half subset:
  - If the subset contains all possible blocks within 1 of l est, search in Table 1
  - If it is missing a block within 1 of largest, search in Tab

$O(n\ 2^{n/2})$ (just like before!)

Problem 3: Balanced towers (Running time)

Case 2 (smaller tower does have largest block):

- There needs to be at least one block in the larger tower within 1 of largest (must be *excluded* from our small-tower solution)
- Build two tables: Table1 only contains subsets which are *missing* at least one block within 1 of largest. Table2 contains all subsets.
- For each first-half subset:
  - If the subset contains all possible blocks within 1 of largest, search in Table 1
  - If it is missing a block within 1 of largest, search in Table 2

Can check in O(n) time per subset

Problem 3: Balanced towers

Final algorithm:

- Split in two cases: either the smallest tower has the largest block, or it does not. We will run the algorithm for both cases and take the better solution
- (Cases 1 is as described above; Case 2 is as described above)

- Running time: $O(n\ 2^{n/2})$

Problem 3: Balanced towers

Whoa…do I really need all that?

Problem answer expectations

- Make sure your algorithm is clear (I should be able to implement it with your description)
  - "Store the best k elements in a sorted array. When considering a new element, insert it into the sorted array, shifting remaining elements down" – OK
  - "Store the best k elements in a (k+1)-sized max heap. Remove the smallest element before inserting a new element" – also OK (I know what a heap is)
  - "Keep track of the best when going through each subset. If it's not balanced, throw it out" – not enough detail!
- I went very slowly through the problem
  - Want to give an example of algorithmic thinking
  - Want to give an example of a VERY clear description
- There existed correct 3-4 line solutions to both of these problems

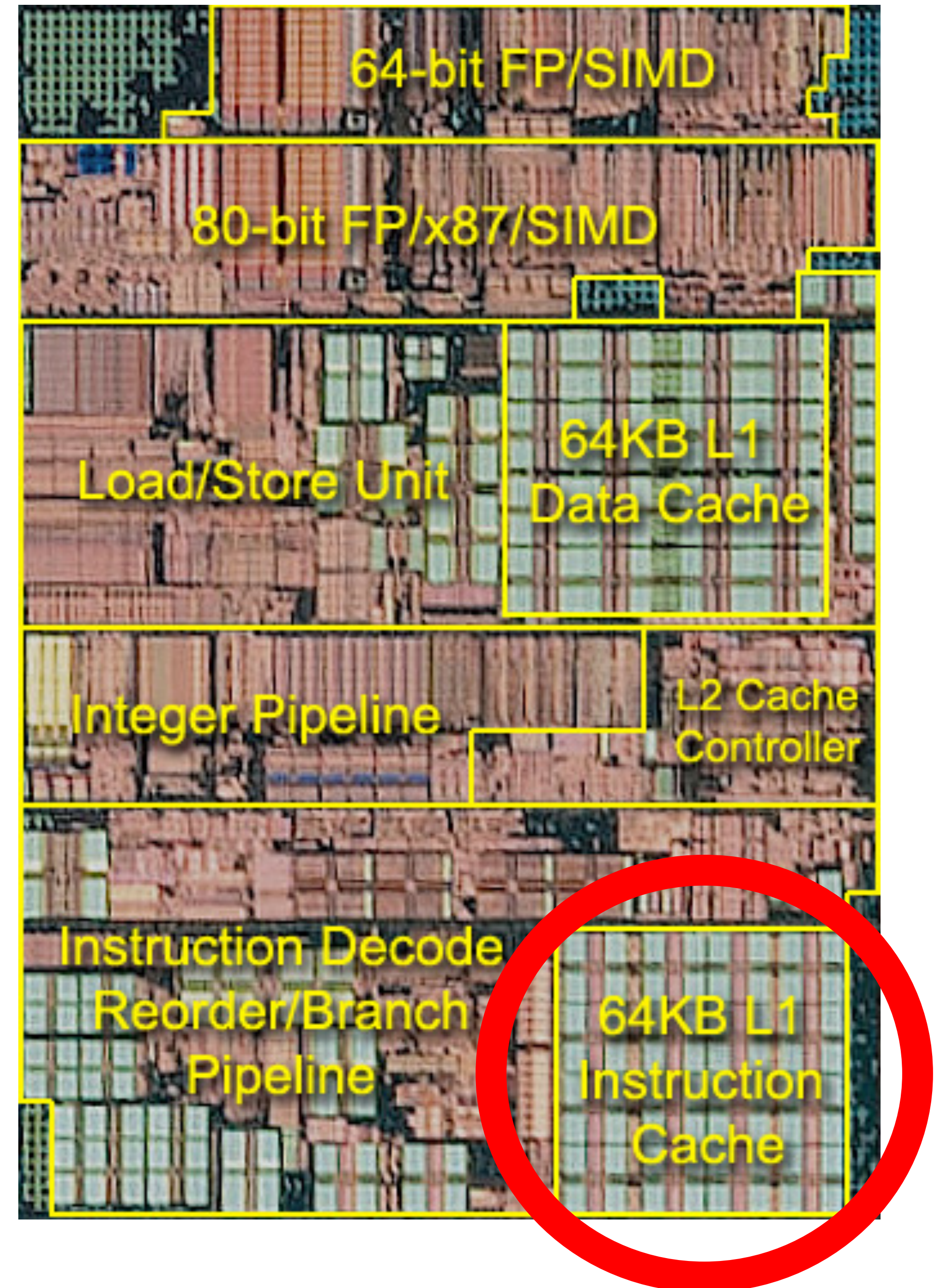
Pipelining

Pipelining

- Most of you have seen briefly
- Some of you have seen in detail
- I’ll go at a moderate pace; ask if you have questions

A discrepancy

- If my instructions live here…
- Shouldn't EVERY operation involve an L1 cache miss?
  - (Or at least equivalent waiting time?)
- How bad are L1 cache misses again?



A Typical Memory Hierarchy

- Everything is a cache for something else…

		Access time	Capacity	Managed By
On the datapath	Registers	1 cycle	1 KB	Software/Compiler
	Level 1 Cache	2-4 cycles	32 KB	Hardware
	Level 2 Cache	10 cycles	256 KB	Hardware
On chip	Level 3 Cache	40 cycles	10 MB	Hardware
Other chips	Main Memory	200 cycles	10 GB	Software/OS
	Flash Drive	10-100us	100 GB	Software/OS
Mechanical devices	Hard Disk	10ms	1 TB	Software/OS

Instruction cache miss

- If normally an addition would be ~1 cycle, we need to first fetch the instruction (2-4 cycles) then do the addition
- If we can avoid this, simple operations would get 3-5x faster!

Pipelining

- Your CPU brings in several instructions at once
- Brings them in even before usage (no cost at all)
- When won’t this work?

Branches

- On a conditional (if, while, switch etc.), CPU may not know what code will be next until it actually hits that branch and evaluates it

- “Branch predictor”: circuit that guesses the result of each conditional to allow for pipelining
  - Probabilistic: “how likely is this if to evaluate to true?”

- Like caching, you can assume this is about as good as possible
  - Pipelining will be great if the conditional is “mostly true” or “mostly false.” It will be wrong ~50% of the time if the if is 50-50.

Back to reality

- Just a note: I am **extremely** oversimplifying all of this
- Rule of thumb: a branch misprediction costs 10-20 clock cycles

What branches are easy to predict?

```
for(int x = 0; x < len1; x++) {
    a1[x] = 4;
}
```

- Pretty easy to predict if len1 is large
- By the way: use memset() for this

Difficult-to-predict example

Cost of conditionals

- Try to take difficult-to-predict conditionals out of your code
- One crazy example that will not always be a good idea:
  (min with an if) vs (min with lots of bit tricks)

```
r = (x < y) ? x : y;                    // min(x, y)

r = y ^ ((x ^ y) & -(x < y)); // min(x, y)
```

Cost of conditionals

- Try to take difficult-to-predict conditionals out of your code
- Better example is things like sorting (if it's cheap), or generally grouping similar answers together
- Also problem-specific tricks:

```
if(x % 2 == 1) {
  y+= 1;
}
```

```
y += x & 1;
```

Optimizing Loops

When do optimizations actually help?

- Your compiler does much of this work for you
- It’s really good at it

- When is it useful to pay attention to this when coding?
  - When you know something the compiler may not

Taking out expensive operations

```
for(int i = 0; i < strlen(str1); i++){
    str1[i] = ‘a’;
}
```

- Unsurprising that this is inefficient—should store strlen(str1) first
- Does the compiler know that the string’s length is unchanged during the body of the loop? (To me this is unclear here…but in other situations it may be obvious that the compiler can’t know!)

More subtle issues

```
int len = strlen(str1);
for(int i = 0;i < len; i++){
    str1[i] = str1[0];
}
```

```
int len = strlen(str1);
int start = str1[0];
for(int i = 0;i < len;i++){
    str1[i] = start;
}
```

Version on right is ~2 as fast for a large array (for me), even with compiler optimizations on
Why??

More subtle issues

```c
int len = strlen(str1);
for(int i=0;i<strlen(str1);i++){
    str1[i] = str1[0];
}
```

```c
int len = strlen(str1);
int start = str1[0];
for(int i=0;i<strlen(str1);i++){
    str1[i] = start;
}
```

What makes this slow?
Every time, need to look up if str1[0] changed and bring it into a register
That is to say: L1 cache miss!

Out-of-order execution

- If possible, modern processers may decide to execute operations out of order if it makes things faster
  - May even be able to do things in parallel, ex: int and float operations

- I won’t go over in this class

- One more reason to make dependencies obvious when possible

What to improve?

Amdahl’s law

Two independent parts **A** **B**

Original process

Make **B** 5x faster

Make **A** 2x faster

- 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

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

Amdahl's law and asymptotics

- If a portion of your program is asymptotically dominated by another, it is less likely to be worth speeding up

Two examples: Edit distance

- How often is “min” called?

- Let’s say I reverse both strings at the beginning of execution. Is that worth speeding up?

Profiling

- There exist tools to help figure out what to speed up

Gprof

- Included on lab machines
- Says what amount of time is spent within each function
- Issue: optimization flags
  - With flags on, gcc is likely to significantly alter your code for speed; makes results difficult to interpret
  - With flags off, might get an inaccurate result

gprof

How to use:

1. Compile with –pg flag
2. Run the program
3. Run gprof with argument [your program]

Example: my edit distance program

Profiling cache

Valgrind --tool=cachegrind [your program]

Gives three sections of output:

- First, instruction L1 cache misses (like branch mispredictions)
- Then, data L1 cache misses
- Finally, combined L3 misses

Valgrind (cachegrind) comments

- Essentially runs your code on a VM
- Consequences:
  - Your code runs differently! May be some surprises/may miss some errors
  - It is VERY slow sometimes
  - A bit out of date compared to our machines
  - Massively simplified---may not pick up on subtle costs (only simulates first and last level of cache)

Profiling cache: detailed output

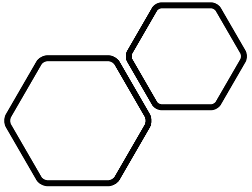
- Run cg_annotate on the output file
  - cachegrind.out.[the pid of the process]

- It will give a function-by-function breakdown

What to write for “optimizations”

- Any optimizations you used (high-level or low level)
- Arguments about what you aimed to speed up and why
  - Amdahl’s law
  - Profilers
  - Branch mispredictions/cache efficiency/etc

- Don’t need to write how the algorithm (meet in the middle/Hirshberg’s) works. But you can if it helps



Ending Section 1 of class (Time and Space)

Improving performance: applied side

- Cost of operations
  - Add, multiply, divide, modulo, allocation
- Cache-efficiency
- Pipelining and loop unrolling

Improving performance: algorithms side

- Tables to decrease search space
- Space efficient algorithms to improve time
- Replacing cache-inefficient tactics with cache-efficient tactics
  - Like sort/scan instead of table lookups/binary search

Timsort

First pass: run generation

- Before sorting a large array, timsort looks through the array for big "runs" of nearly-sorted data
- What does this look like for your cache?

Run generation: cache perspective

Write smallest item in cache

Read in a new item

Second step: merge runs

- Timsort then takes these large-ish runs and merges them together
- Carefully selects merges
  - Merging different-sized arrays is not too helpful

- Merging has a first step of binary search
  - Often, one array is strictly bigger

- One more optimization: big step, small step

Big step, small step

- Let's say we're merging two arrays, and we've passed a large number of items in the smaller array
- One option: binary search for the next place
    - $\log n$ time
- Can we do better?
- Repeatedly double the size of each step, then binary search
    - If we want to skip forward k, this takes $O(\log k)$

Timsort

- Performs much better than quicksort on almost-sorted data
- So if we want to sort really fast:
  - Our starting tables should be somewhat sorted
  - Then perform an “adaptive” sort like timsort
- How can we guarantee somewhat-sorted tables?
  - Sort input