Applied Algorithms Lec 3: Cache Efficiency

Sam McCauley

September 12, 2025

Williams College

Admin

- Career fair in Chandler today
- Assignment 1 out! Start early!!
    - I'll get gradescope set up soon
- Don't do extra credit until you're done with the rest of the assignment
- 312 Lab computers are now accessible via ssh; link sent and also available on assignments page
- Meet and greet colloquium today. Next week: an (applied?!) algorithms colloquium

Wrapping up MITM

Meet in the Middle

```
1  for each subset A_1 of S_1:
2      s_1 ← h(A_1)
3      for each subset A_2 of S_2 :
4          if h(A_2) + s_1 ≤ h(S)/2 :
5          updateMax(h(A_2) + s_1)
```

- What is this inner poriton doing?
- Finds the set A_2 with height closest to $h(S)/2$
- How can we preprocess S_2 to answer these queries quickly? Let's split into pairs and discuss for a few minutes.
- Answer: we're looking for the largest subset of S_2 with height at most $h(S)/2 - s_1$.
- Sort all subsets of S_2. Then can answer this query using binary search!

Meet in the Middle

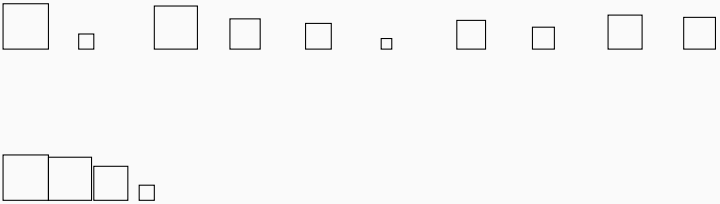
```
1  Fill array P with all subsets of S_2
2  Sort P by height
3  for each subset A_1 of S_1:
4      s_1 ← h(A_1)
5      binsearch(P, h(S)/2 − s_1)
6      updateMax(h(A_2) + s_1)
```

- Analysis?
- P has length $O(2^{n/2})$. Sorting it takes $O(n2^{n/2})$
- Each binary search takes $O(n)$ time; perform $O(2^{n/2})$ of them
- Total: $O(n2^{n/2})$ space, $O(n2^{n/2})$ time

Meet in the Middle

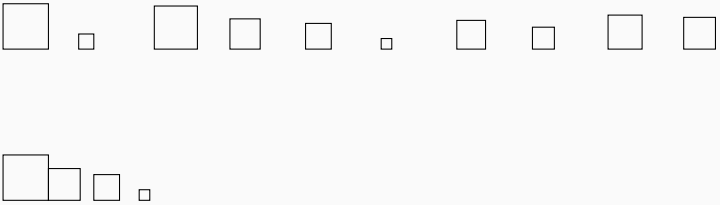
- Before we go forward, let’s go over the high level strategy

Meet in the Middle



Let’s say we have a set of blocks. Normally we use will try all subsets of these blocks and find the largest that’s at most half the total size.

Meet in the Middle



Let’s say we have a set of blocks. Normally we use will try all subsets of these blocks and find the largest that’s at most half the total size.

Meet in the Middle

Partition the blocks into two equal-sized sets.

Meet in the Middle

Partition the blocks into two equal-sized sets. Question: what subset of the *yellow* blocks is used in the correct solution?

Meet in the Middle

0.0	00000
7.2	00001
5.1	00010
12.3	00011
9.8	00100
17.0	00101

...

First, let's do some brute force preprocessing on the blue blocks. Go through all subsets of the blue blocks, and store their heights in a table.

Meet in the Middle

0.0	00000
5.1	00010
7.2	00001
9.8	00100

...

First, let's do some brute force preprocessing on the blue blocks. Go through all subsets of the blue blocks, and store their heights in a table. Then, sort the table by height.

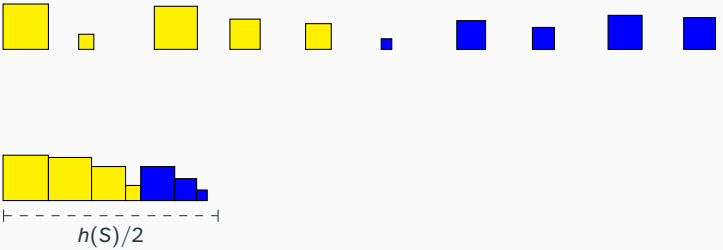
Meet in the Middle

0.0	00000
5.1	00010
7.2	00001
9.8	00100

...

First, preprocess the blue blocks. Go through all subsets of the blue blocks, and store their heights in a table. Then, sort the table by height.

Meet in the Middle



Now, go through every possible subset of yellow blocks. We want blue blocks with height as close to $h(S)/2 - h(A_1)$ as possible.

Meet in the Middle

0.0	00000
5.1	00010
7.2	00001
9.8	00100

…

How quickly can we find the *best* set of blue blocks with height at most $h(S)/2 - h(A_1)$? Why don't we need to check any other subsets of blue blocks?

What we get

- $O(n2^{n/2})$ space, $O(n2^{n/2})$ time. (Everyone remember how?)

What we get

- $O(n2^{n/2})$ space, $O(n2^{n/2})$ time. (Everyone remember how?)
- “Meet in the middle”—rather than considering all subsets, we break into two halves. We search in the yellow and blue halves *one at a time*, then combine them to get one solution.

What we get

- $O(n2^{n/2})$ space, $O(n2^{n/2})$ time. (Everyone remember how?)
- "Meet in the middle"—rather than considering all subsets, we break into two halves. We search in the yellow and blue halves *one at a time*, then combine them to get one solution.
- Very wide uses: optimization problems, cryptography, etc.

What does this mean?

- What is $O(n2^n)$ vs $O(n2^{n/2})$ time? Do they differ by more than a constant?

What does this mean?

- What is $O(n2^n)$ vs $O(n2^{n/2})$ time? Do they differ by more than a constant?
- $O(n2^{n/2})$ space is a lot. Is this worth it?

What does this mean?

- What is $O(n2^n)$ vs $O(n2^{n/2})$ time? Do they differ by more than a constant?
- $O(n2^{n/2})$ space is a lot. Is this worth it?
- Wait, can we do better than this?

Some questions about meet in the middle

- How else can we store solutions from the blue subproblems? What does this data structure need to support?

Some questions about meet in the middle

- How else can we store solutions from the blue subproblems? What does this data structure need to support?
  - Needs to support predecessor queries!

Some questions about meet in the middle

- How else can we store solutions from the blue subproblems? What does this data structure need to support?
  - Needs to support predecessor queries!
- What if we wanted to search for two towers that were exactly equal? Would our strategy change? Could we get improved running time?

Some questions about meet in the middle

- How else can we store solutions from the blue subproblems? What does this data structure need to support?
  - Needs to support predecessor queries!
- What if we wanted to search for two towers that were exactly equal? Would our strategy change? Could we get improved running time?
- What property must a problem have for MitM to work?
  - Can *all* brute force search problems with N solutions be solved in something like $O(\sqrt{N})$ time?

Some questions about meet in the middle

- How else can we store solutions from the blue subproblems? What does this data structure need to support?
  - Needs to support predecessor queries!
- What if we wanted to search for two towers that were exactly equal? Would our strategy change? Could we get improved running time?
- What property must a problem have for MitM to work?
  - Can *all* brute force search problems with N solutions be solved in something like $O(\sqrt{N})$ time?
  - No: need the two halves to be *independent*. (We build the table on the blue half once. That table needs to work for every query.)

Some questions about meet in the middle

- How else can we store solutions from the blue subproblems? What does this data structure need to support?
  - Needs to support predecessor queries!
- What if we wanted to search for two towers that were exactly equal? Would our strategy change? Could we get improved running time?
- What property must a problem have for MitM to work?
  - Can *all* brute force search problems with N solutions be solved in something like $O(\sqrt{N})$ time?
  - No: need the two halves to be *independent*. (We build the table on the blue half once. That table needs to work for every query.)
  - For example, 3SAT doesn't work here.

Looking at Assignment 1

- You’ll do 4 implementations

Looking at Assignment 1

- You'll do 4 implementations
- They will (hopefully) get *much* faster

Looking at Assignment 1

- You'll do 4 implementations
- They will (hopefully) get *much* faster
- But they all have the same asymptotic running time!

Looking at Assignment 1

- You'll do 4 implementations
- They will (hopefully) get *much* faster
- But they all have the same asymptotic running time!
- Topic for next couple lectures: why is that??

Any lingering questions about Assignment 1 or MITM?

Optimization

Making Software Fast

What do we mean by fast?

- Generally: runs in a small amount of time (wall-clock time)

Making Software Fast

What do we mean by fast?

- Generally: runs in a small amount of time (wall-clock time)
- Lots of caveats:

Making Software Fast

What do we mean by fast?

- Generally: runs in a small amount of time (wall-clock time)
- Lots of caveats:
  - Can be *very* machine-dependent

Making Software Fast

What do we mean by fast?

- Generally: runs in a small amount of time (wall-clock time)
- Lots of caveats:
  - Can be *very* machine-dependent
  - May depend on the structure of the data

Making Software Fast

What do we mean by fast?

- Generally: runs in a small amount of time (wall-clock time)
- Lots of caveats:
  - Can be *very* machine-dependent
  - May depend on the structure of the data
  - May depend on random choices of the program

Making Software Fast

What do we mean by fast?

- Generally: runs in a small amount of time (wall-clock time)
- Lots of caveats:
  - Can be *very* machine-dependent
  - May depend on the structure of the data
  - May depend on random choices of the program
  - Can be impacted by other software running on the computer

Making Software Fast

What do we mean by fast?

- Generally: runs in a small amount of time (wall-clock time)
- Lots of caveats:
  - Can be *very* machine-dependent
  - May depend on the structure of the data
  - May depend on random choices of the program
  - Can be impacted by other software running on the computer
  - May depend on the compiler used

Making Software Fast

What do we mean by fast?

- Generally: runs in a small amount of time (wall-clock time)
- Lots of caveats:
  - Can be *very* machine-dependent
  - May depend on the structure of the data
  - May depend on random choices of the program
  - Can be impacted by other software running on the computer
  - May depend on the compiler used
  - Oftentimes: not obvious what causes the improvement!

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

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"

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"
- 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

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 Assignment 1

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 Assignment 1
- Can also use unix `time` function

Making Software Fast: Things to Bear in Mind

In this class:

- Measure time using built-in tools

Making Software Fast: Things to Bear in Mind

In this class:

- Measure time using built-in tools
- On lab machines

Making Software Fast: Things to Bear in Mind

In this class:

- Measure time using built-in tools
- On lab machines
- Always be specific about datasets

Making Software Fast: Things to Bear in Mind

In this class:

- Measure time using built-in tools
- On lab machines
- Always be specific about datasets
- Run multiple times to try to avoid issues where other software interferes, or we get lucky/unlucky with random choices

Making Software Fast: Things to Bear in Mind

In this class:

- Measure time using built-in tools
- On lab machines
- Always be specific about datasets
- Run multiple times to try to avoid issues where other software interferes, or we get lucky/unlucky with random choices
- Look for *significant* improvement: usually you want experiments where the running time is close to a second, or more

Making Software Fast: Things to Bear in Mind

In this class:

- Measure time using built-in tools
- On lab machines
- Always be specific about datasets
- Run multiple times to try to avoid issues where other software interferes, or we get lucky/unlucky with random choices
- Look for *significant* improvement: usually you want experiments where the running time is close to a second, or more
    - Very difficult to draw conclusions from “Program A ran in 10 milliseconds; Program B ran in 15 milliseconds.”

Let’s say I have a piece of code, and I want to make it faster. How should I do that?

Thought question

- What part of a program is most important to speed up?

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?

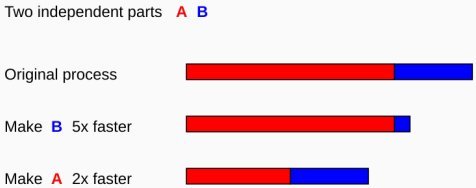
Thought question

- 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: gain the most from 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

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: gain the most from 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.

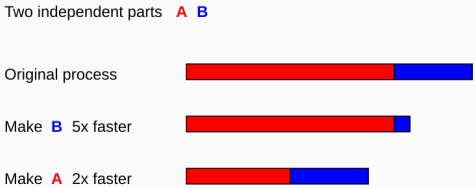
Amdahl's Law



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



If a function takes up a fraction of the entire program's running time and you speed it up by a factor s, the overall program speedup is the factor

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

I want you to know the principle here, but not memorize the formula.

Amdahl’s Law and Asymptotics

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

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.

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 in the real world if each call is very fast?

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 in the real world 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!

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 in the real world 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?

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 in the real world if each call is very fast?

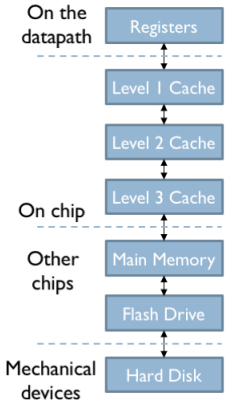
1. One option: factor out subroutine using separate testing code
    - Need to get info on how often it's called; simulate correct types of data.
    - Make sure the compiler does not optimize out your whole experiment!
2. Another option: Run same code with and without subroutine
    - Does that change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

We'll come back to this with some examples in the next couple lectures. Bear in mind: benchmarking itself is an entire area of computer science.

Today: Cost to access data frequently dominates running time

A Typical Memory Hierarchy

- Everything is a cache for something else…

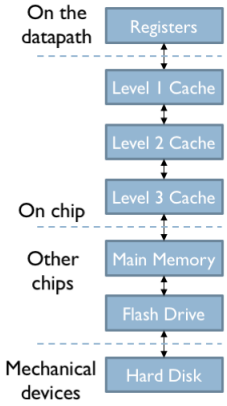


Access time	Capacity	Managed By
1 cycle	1 KB	Software/Compiler
2-4 cycles	32 KB	Hardware
10 cycles	256 KB	Hardware
40 cycles	10 MB	Hardware
200 cycles	10 GB	Software/OS
10-100us	100 GB	Software/OS
10ms	1 TB	Software/OS

Takeaway: if I access RAM one extra time, that is the same as doing (very roughly) *two hundred* extra operations.

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

Tradeoff: lower-level caches are much faster, but much smaller. Only store a small amount of data can be accessed quickly.

How caches work

- Since cache performance is crucial; *where* data is stored is also crucial

How caches work

- Since cache performance is crucial; *where* data is stored is also crucial
- Your computer stores data in the optimal(ish) place

How caches work

- Since cache performance is crucial; *where* data is stored is also crucial
- Your computer stores data in the optimal(ish) place
- Moves data around in *cache lines* of $\approx$ 64 bytes

How caches work

- Since cache performance is crucial; *where* data is stored is also crucial
- Your computer stores data in the optimal(ish) place
- Moves data around in *cache lines* of $\approx$ 64 bytes
- Modern caches are very complicated

How caches work

- Since cache performance is crucial; *where* data is stored is also crucial
- Your computer 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

How caches work

- Since cache performance is crucial; *where* data is stored is also crucial
- Your computer 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.

How caches work

- Since cache performance is crucial; *where* data is stored is also crucial
- Your computer 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`

Profiler example: cachegrind

- Cachegrind helps us analyze the number of cache misses inccured by a program!

Profiler example: cachegrind

- Cachegrind helps us analyze the number of cache misses inccured by a program!
- Compile with debugging info on `-g` AND optimizations on
  - What does this entail immediately?

Profiler example: cachegrind

- Cachegrind helps us analyze the number of cache misses inccured by a program!
- Compile with debugging info on `-g` AND optimizations on
  - What does this entail immediately?
- Then `valgrind --tool=cachegrind -cache-sim=yes [your program]`

Profiler example: cachegrind

- Cachegrind helps us analyze the number of cache misses inccured by a program!
- Compile with debugging info on `-g` AND optimizations on
  - What does this entail immediately?
- Then `valgrind --tool=cachegrind -cache-sim=yes [your program]`
- Outputs number of cache misses for instructions, then data, then combined

Profiler example: cachegrind

- Cachegrind helps us analyze the number of cache misses inccured by a program!
- Compile with debugging info on `-g` AND optimizations on
  - What does this entail immediately?
- Then `valgrind --tool=cachegrind -cache-sim=yes [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

Profiler example: cachegrind

- Cachegrind helps us analyze the number of cache misses inccured by a program!
- Compile with debugging info on `-g` AND optimizations on
  - What does this entail immediately?
- Then `valgrind --tool=cachegrind -cache-sim=yes [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

Profiler example: cachegrind

- Cachegrind helps us analyze the number of cache misses inccured by a program!
- Compile with debugging info on `-g` AND optimizations on
  - What does this entail immediately?
- Then `valgrind --tool=cachegrind -cache-sim=yes [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

Reading cachegrind output

- I, I1, LLi, etc.: *instruction* misses
- D, D1: first level of cache

Reading cachegrind output

- I, I1, LLi, etc.: *instruction* misses
- D, D1: first level of cache
- LL: last layer of cache

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

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

Real-World Caching

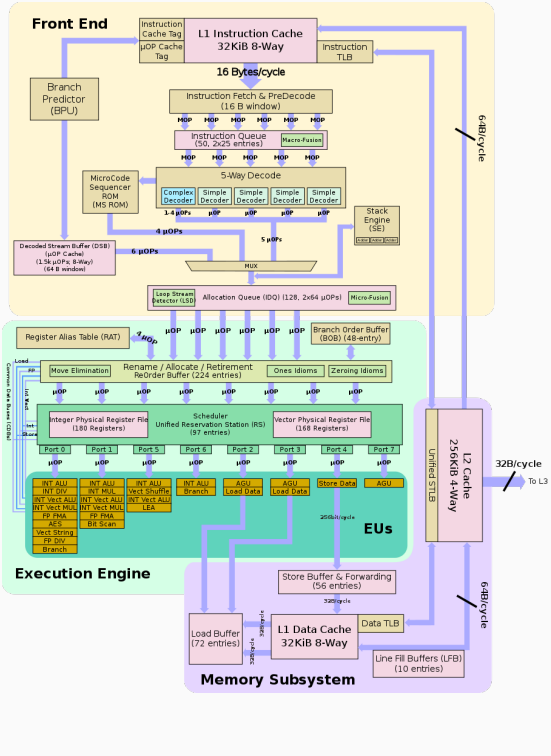
- We looked at simple, constructed examples where caching is easy to reason about
- But: bear in mind that modern caches are very complicated; interact nontrivially with other costs (branch mispredictions; expensive operations; etc.)

Real-World Caching

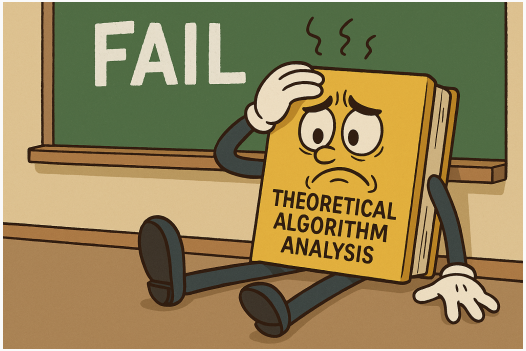
- We looked at simple, constructed examples where caching is easy to reason about
- But: bear in mind that modern caches are very complicated; interact nontrivially with other costs (branch mispredictions; expensive operations; etc.)
- I should at least mention *prefetching*: if your computer thinks it can get a head start on fetching your data, it will

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)
Instruction Queue (50, 2x25 entries)
Macro-Fusion
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)
6-8B/cycle
4 µOPs
5 µOPs
Decoded Stream Buffer (DSB) (µOP Cache) (1.5k µOPs, 8-Way) (64 B window)
6 µOPs
MUX
Loop Stream Detector (LSD)
Allocation Queue (IDQ) (128, 2x64 µOPs)
Micro-Fusion
Register Alias Table (RAT)
4 µOPs
Branch Order Buffer (BOB) (48-entry)
Load
RF
Move Elimination
Rename / Allocate / Retirement ReOrder Buffer (224 entries)
Ones Idioms
Zeroing Idioms
Store
Int Vect
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
INT ALU
INT DIV
INT Vect ALU
INT Vect MUL
FP MAA
AES
Vect String
FP DIV
Branch
INT ALU
INT MUL
INT Vect ALU
INT Vect MUL
FP MAA
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)
64B/cycle
64B/cycle
L1 Data Cache 32KiB 8-Way
Data TLB
Line Fill Buffers (LFB) (10 entries)
6-8B/cycle
Memory Subsystem

Summary



- Algorithms has failed us
- Not all operations are equal in practice!
- Constants matter!

Return of Algorithms

A very current reference to a comeback story.

- Can we take cache misses into account?
- What if rather than measuring operations (as we did in CS 256), we analyze only *cache misses*?
- Next topic: the External Memory Model

External Memory Model

What do we want out of this model?

- Simple, but able to capture the cost of cache misses

What do we want out of this model?

- Simple, but able to capture the cost of cache misses
- How can we make it universal across computers that may have very different cache parameters?

What do we want out of this model?

- Simple, but able to capture the cost of cache misses
- How can we make it universal across computers that may have very different cache parameters?
    - Answer: we'll use parameters. (The exact size of cache, and a cache line, can *drastically* affect algorithmic performance.)

What do we want out of this model?

- Simple, but able to capture the cost of cache misses
- How can we make it universal across computers that may have very different cache parameters?
    - Answer: we'll use parameters. (The exact size of cache, and a cache line, can *drastically* affect algorithmic performance.)
- Do we want asymptotics? Worst case?

What do we want out of this model?

- Simple, but able to capture the cost of cache misses
- How can we make it universal across computers that may have very different cache parameters?
    - Answer: we'll use parameters. (The exact size of cache, and a cache line, can *drastically* affect algorithmic performance.)
- Do we want asymptotics? Worst case?
    - Yes!

External memory model basics

- Single cache of size M

External memory model basics

- Single cache of size M
- Cache line size is B

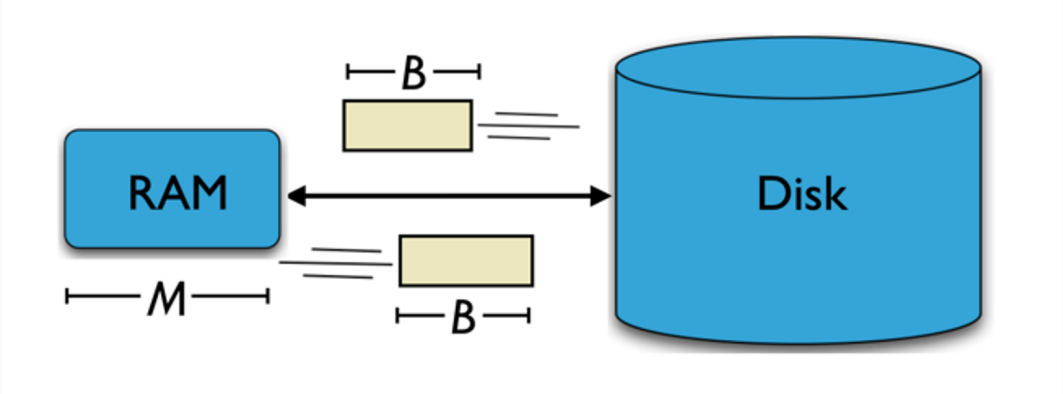
External memory model basics

- Single cache of size M
- Cache line size is B
- Computation is free: *only* count number of “cache misses.” Can perform arbitrary computation on items in cache.

External memory model basics

- Single cache of size M
- Cache line size is B
- Computation is free: *only* count number of "cache misses." Can perform arbitrary computation on items in cache.
- We will say something like "$O(n/B)$ cache misses" rather than "$O(n)$ operations" to emphasize the model.

External Memory Model Basics



Transferring *B* *consecutive* items to/from the disk costs 1. Can only store *M* things in cache.

Memory Evictions

- Can only hold *M* items in cache!

Memory Evictions

- Can only hold *M* items in cache!
- So when we bring *B* in, need to write *B* items back to disk. (We can bring them in later if we need them again)

Memory Evictions

- Can only hold M items in cache!
- So when we bring B in, need to write B items back to disk. (We can bring them in later if we need them again)
- Assume that the computer does this optimally.
  - Reasonable; it's really good at it. Very cool algorithms behind this!

Vocabulary

- “Cache” of size *M*; “disk” of unlimited size

Vocabulary

- “Cache” of size *M*; “disk” of unlimited size
- With the cost of one “cache miss” can bring in *B* consecutive items
  - (Sometimes called “memory access” or “I/Os” but I will try not to use those terms.)

Vocabulary

- “Cache” of size *M*; “disk” of unlimited size
- With the cost of one “cache miss” can bring in *B* consecutive items
  - (Sometimes called “memory access” or “I/Os” but I will try not to use those terms.)
- These *B* items are called a “block” or a “cache line”.

Let's revisit `sortedlinkedlist.c`

- What is the cost of our algorithm in the external memory model if the items are stored in order?

Let's revisit `sortedlinkedlist.c`

- What is the cost of our algorithm in the external memory model if the items are stored in order?
- Answer: $O(n/B)$

Let's revisit `sortedlinkedlist.c`

- What is the cost of our algorithm in the external memory model if the items are stored in order?
- Answer: $O(n/B)$
- What is the cost of our algorithm in the external memory model if the items have stride $B + 1$?

Let's revisit `sortedlinkedlist.c`

- What is the cost of our algorithm in the external memory model if the items are stored in order?
- Answer: $O(n/B)$
- What is the cost of our algorithm in the external memory model if the items have stride $B + 1$?
- Answer: $O(n)$

Let's revisit `sortedlinkedlist.c`

- What is the cost of our algorithm in the external memory model if the items are stored in order?
- Answer: $O(n/B)$
- What is the cost of our algorithm in the external memory model if the items have stride $B + 1$?
- Answer: $O(n)$
- The external memory model predicts (roughly) the real-world slowdown of this process.

Finding the minimum element in an array

- How many cache misses in the external memory model?

Finding the minimum element in an array

- How many cache misses in the external memory model?
- Answer: $O(n/B)$

Binary search?

- What is the recurrence for binary search in terms of number of operations?

Binary search?

- What is the recurrence for binary search in terms of number of operations?
- What is the recurrence for binary search in terms of the number of cache misses?

Binary search?

- What is the recurrence for binary search in terms of number of operations?
- What is the recurrence for binary search in terms of the number of cache misses?
- Each recursive call takes 1 cache miss.

Binary search?

- What is the recurrence for binary search in terms of number of operations?
- What is the recurrence for binary search in terms of the number of cache misses?
- Each recursive call takes 1 cache miss.
- Base case: can perform *all* operations on B items with only 1 cache miss

Binary search?

- What is the recurrence for binary search in terms of number of operations?
- What is the recurrence for binary search in terms of the number of cache misses?
- Each recursive call takes 1 cache miss.
- Base case: can perform *all* operations on B items with only 1 cache miss
- Total: $O(\log_2(n/B))$ cache misses.

Fitting in Cache

- If you have a sequence of operations on a dataset of size at most M, there is no further cost so long as they all stay in cache!

Fitting in Cache

- If you have a sequence of operations on a dataset of size at most M, there is no further cost so long as they all stay in cache!

- $O(M/B)$ to load the items into cache, then all computation is free

Fitting in Cache

- If you have a sequence of operations on a dataset of size at most M, there is no further cost so long as they all stay in cache!
- $O(M/B)$ to load the items into cache, then all computation is free
- Real-world time: what if instead of a linked list of 100 million items, we repeatedly access a linked list of 100 thousand items?

Why does the external memory model make sense?

- Simple model that captures *one level* of the memory hierarchy

Why does the external memory model make sense?

- Simple model that captures *one level* of the memory hierarchy
- Idea: usually one level has by far the largest cost.

Why does the external memory model make sense?

- Simple model that captures *one level* of the memory hierarchy
- Idea: usually one level has by far the largest cost.
    - Small programs may be dominated by L1 cache misses

Why does the external memory model make sense?

- Simple model that captures *one level* of the memory hierarchy
- Idea: usually one level has by far the largest cost.
  - Small programs may be dominated by L1 cache misses
  - Larger programs it may be by L3 cache misses
- External memory model zooms in on one crucial level of the memory hierarchy (with particular B, M); gives asymptotics for how well we do on that level.

Question about External Memory Model Basics?