# Applied Algorithms Lec 4: Cache Misses and External Memory

Sam McCauley

October 21, 2021

Williams College

# Admin

- Office hours 2:30–4 and 5–6:30 today
- No office hours tomorrow; Wed 2–4 in TCL 312 instead
- Slack vs email (everything important will be on email)
- Some reading today! Optional/potentially useful for reference. We don't cover the topic in exactly the same way
  - For ex: we'll have $K = 1$; no distribution sort; no $B$-trees

# Admin: Talks coming up

- Tomorrow 7PM: Landing a tech internship (featuring your fellow Williams students)
- Thursday 5PM: “How’d you get there?” talk about jobs in sustainable tech (need preregister)
- Friday 2:30: Colloquium is by Andrew McGregor, who works in randomized streaming algorithms (topic in Part 2 of course)
  - More details later. But almost certainly one of the most relevant talks to this course this year

# Admin: Assignment 1

- Assignment 1 due Wednesday 10pm
- Emailed out small Assignment 1 updates
- Extra test run at 7pm Wed
- Add `-lm` to `make debug`
- Tips for fast sorting added to website

# Any Assignment 1 Questions?

# Avoiding branch mispredictions

- Avoid branches (ifs, etc.)
- If you do create a branch, ask yourself how easy it is to predict!
- Compiler can remove a lot of branch mispredictions
- Only way to be sure is to experiment

# 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
  - (Can be issues with optimizations, especially `-O3`)
- I may ask you to use this, but be aware of its limitations

# Profilers examples: cachegrind

- Compile with debugging info on `-g` AND optimizations on
    - What does this entail immediately?
- Then `valgrind --tool=cachegrind [your program]`
- Use `--branch-sim=yes` for branch prediction statistics.
    - Very oversimplified unfortunately
- 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

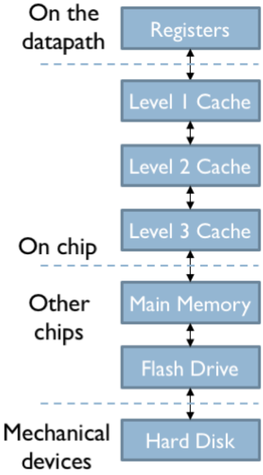
# Final major cost: cache misses!

- Data is stored in different places on the computer

- Cost to access it frequently dominates running time

# A Typical Memory Hierarchy

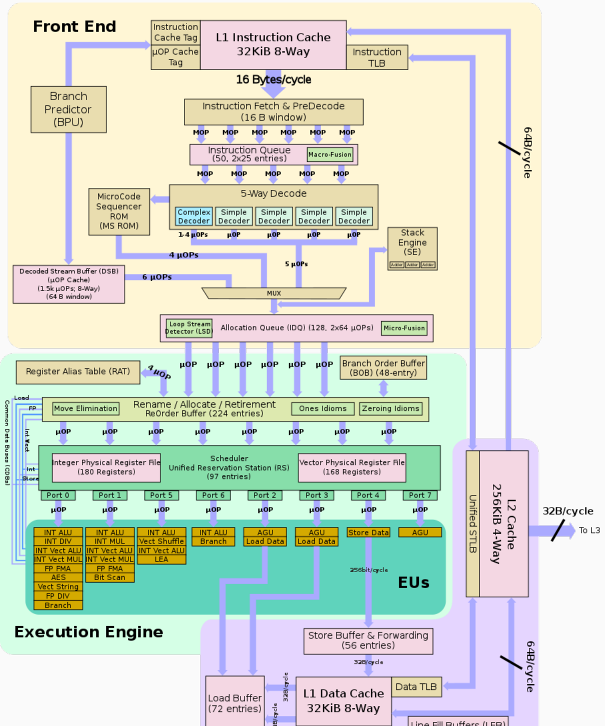
- Everything is a cache for something else…




| 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
- Example: `cachemisses.c`
- Modern caches are **very** complicated
- 
- 



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
Instruction Queue
(50, 2x25 entries)
Macro-Fusion
MicroCode Sequencer ROM (MS ROM)
5-Way Decode
Complex Decoder
Simple Decoder
1-4 µOPs
µOP
4 µOPs
5 µOPs
6 µOPs
Stack Engine (SE)
Adder
Decoded Stream Buffer (DSB)
(µOP Cache)
(1.5k µOPs; 8-Way)
(64 B window)
MUX
Loop Stream Detector (LSD)
Allocation Queue (IDQ) (128, 2x64 µOPs)
Micro-Fusion
64B/cycle
Register Alias Table (RAT)
4 µOP
Branch Order Buffer (BOB) (48-entry)
Common Data Buses (CDBs)
Load
FP
Int Vect
Int
Store
Move Elimination
Rename / Allocate / Retirement
ReOrder Buffer (224 entries)
Ones Idioms
Zeroing Idioms
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 FMA
AES
Vect String
FP DIV
Branch
INT MUL
Bit Scan
Vect Shuffle
LEA
AGU
Load Data
Store Data
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)
L1 Data Cache
32KiB 8-Way
Data TLB
Line Fill Buffers (LFB)

# How caches work

- Stores data in the optimal(ish) place
- Moves data around in *cache lines* of 64 bytes
- Modern caches are **very** complicated
- Basically: close is good; recent is good; jumping around is bad.
- Your compiler has limited capability to improve cache efficiency!

# Optimization 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.

# Plan for next few lectures

- Today: cache misses. (Best place to gain performance algorithmically)
- Thursday: Assignment 2 algorithm, discussion about optimizing loops and functions
    - Inlining, loop unrolling, more about compiler options
- Monday: more external memory, Assignment 1 code review, any other loose ends

# External Memory Model

# Measuring cache misses

- Takeaway from today's examples: cache performance is often *more important* than number of operations
- But algorithmic analysis measures number of operations
- Can we algorithmically examine the cache performance of a program?
- Yes: external memory model

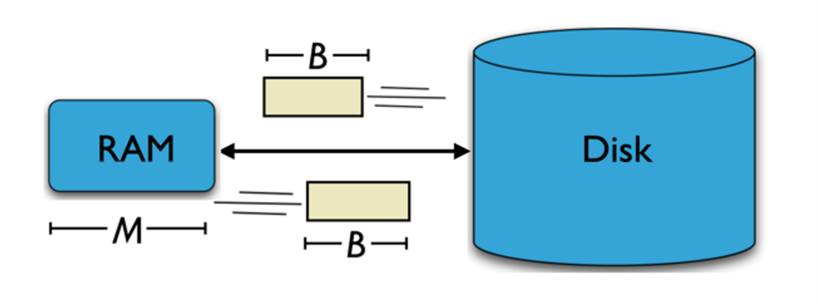
# What do we want out of this model?

- Simple, but able to capture major performance considerations
- Parameters for the model? How can we make it universal across computers that may have very different cache parameters?
- Do we want asymptotics? Worst case?

# External memory model basics

- Cache of size $M$
- Cache line of size $B$
- 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!
- 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
- 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”.

## **Simple example:** `cachemisses.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 the real-world slowdown of this process.

# 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?
- 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.

# Why does this 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?

# Joke to break up the material

almost impossible to convince programmers to stick to that subset. The C compiler which I use can generate warning messages concerning portability, but it is no effort at all to write a non-portable program which generates no compiler warnings.

programmers are the same people who were playing with toy computers as adolescents? We said at the time that using BASIC as a first language would create bad habits which would be very difficult to eradicate. Now we're seeing the evidence of that.

***C is a medium-level language combining the power of assembly language with the readability of assembly language.***

# Joke to break up the material

# Matrix Multiplication in External Memory

# Matrix Multiplication Reminder

- Given two $n \times n$ matrices $A$, $B$
- Want to compute their product $C$:
- $c_{ij} = \sum_{k=1}^{n} a_{ik} b_{kj}$

Example:

$$\begin{bmatrix} 1 & 2 \\ 8 & -1 \end{bmatrix} \times \begin{bmatrix} 2 & 3 \\ -2 & 7 \end{bmatrix} = \begin{bmatrix} -2 & 17 \\ 18 & 17 \end{bmatrix}$$

# Compute Product Directly

```
for i = 1 to n:
  for j = 1 to n:
    for k = 1 to n:
      C[i][j] += A[i][k] +
          B[k][j]
```

- Recall: $c_{ij} = \sum_{k=1}^{n} a_{ik} b_{kj}$
- How many I/Os does this take?
- Assume matrices are stored in row-major order.
  - First: assume $M < n^2$
  - What if $M > n^2$?
  - Answer: $O(n^3)$ I/Os. Every inner loop operation requires an I/O for $B$.

# Any ideas for how to improve this?

- One idea: transpose $B$.
- Another idea: swap the loops!

```
for i = 1 to n:
  for k = 1 to n:
    for j = 1 to n:
      C[i][j] += A[i][k] + B[k][j]
```

- This gives us $O(n^3/B)$ I/Os. Is this actually worth doing?

# Yep!

I am given two functions for finding the product of two matrices:

```
void MultiplyMatrices_1(int **a, int **b, int **c, int n){
      for (int i = 0; i < n; i++)
          for (int j = 0; j < n; j++)
              for (int k = 0; k < n; k++)
                  c[i][j] = c[i][j] + a[i][k]*b[k][j];
 }

void MultiplyMatrices_2(int **a, int **b, int **c, int n){
      for (int i = 0; i < n; i++)
          for (int k = 0; k < n; k++)
              for (int j = 0; j < n; j++)
                  c[i][j] = c[i][j] + a[i][k]*b[k][j];
}
```

I ran and profiled two executables using `gprof`, each with identical code except for this function. The second of these is significantly (about 5 times) faster for matrices of size 2048 x 2048. Any ideas as to why?

c algorithm matrix matrix-multiplication gprof

share improve this question

edited Sep 13 '11 at 2:47
templatetypedef
295k • 80 • 725 • 933

asked Sep 13 '11 at 0:29
kevlar1818
2,639 • 4 • 19 • 39

add a comment

nswers

active oldest votes

# We haven't used the cache yet

- No $M$s in any running times
- Why? All algorithms so far have read the data once and then thrown it away.
- Goal: bring items into cache so that we can perform *many* computations on them before writing them back.
- Note: can't do this with linear scan. $O(n/B)$ is optimal.

# Blocking

- Standard technique for improving cache performance of algorithms.
- Idea: break problems into subproblems of size $O(M)$
  - Can solve any such problem in $O(M/B)$ I/Os
  - Efficiently combine them for a cache-efficient solution

# Blocked Matrix Multiplication

- Split $A$, $B$, and $C$ into blocks of size $M/3$
  - $\sqrt{M/3} \times \sqrt{M/3}$-sized blocks
  - Really want blocks with size $T = \lfloor \sqrt{M/3} \rfloor$. Assume that $T$ divides $n$ for now.
- Multiply blocks one at a time

# Decomposing matrices into blocks

Classic result: if we treat the blocks as single elements of the matrices, and multiply (and add) them as normal, we obtain the same result as we would have in normal matrix multiplication.

- This idea is used in recursive matrix multiplication
- And Strassen's algorithm for matrix multiplication

# Decomposing matrices into blocks

Example: Recall how to multiply 2$x$2 matrices:

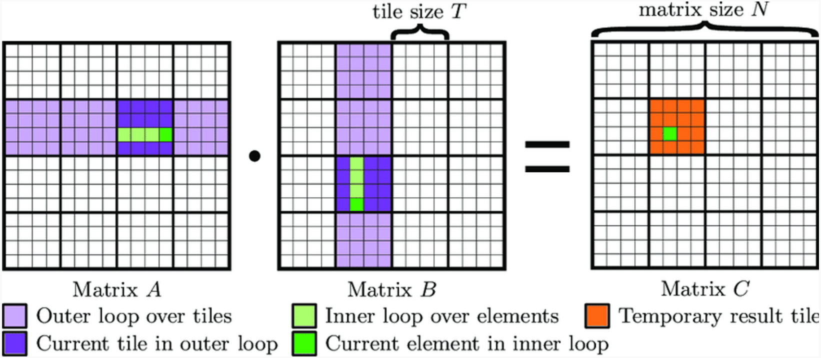
$$\begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix} \cdot \begin{bmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{bmatrix} = \begin{bmatrix} A_{11} \cdot B_{11} + A_{12} \cdot B_{21} & A_{11} \cdot B_{12} + A_{12} \cdot B_{22} \\ A_{21} \cdot B_{11} + A_{22} \cdot B_{21} & A_{21} \cdot B_{12} + A_{22} \cdot B_{22} \end{bmatrix}$$

$$\begin{bmatrix} 17 & 15 & 20 & 4 \\ 15 & 3 & 20 & 8 \\ 1 & 10 & 15 & 2 \\ 3 & 19 & 3 & 14 \end{bmatrix} \cdot \begin{bmatrix} 4 & 12 & 9 & 1 \\ 4 & 6 & 11 & 2 \\ 13 & 18 & 8 & 20 \\ 3 & 11 & 18 & 9 \end{bmatrix} =$$

$$\begin{bmatrix} \begin{bmatrix} 17 & 15 \\ 15 & 3 \end{bmatrix} \cdot \begin{bmatrix} 4 & 12 \\ 4 & 6 \end{bmatrix} + \begin{bmatrix} 20 & 4 \\ 20 & 8 \end{bmatrix} \cdot \begin{bmatrix} 13 & 8 \\ 3 & 11 \end{bmatrix} & \begin{bmatrix} 17 & 15 \\ 15 & 3 \end{bmatrix} \cdot \begin{bmatrix} 9 & 1 \\ 11 & 2 \end{bmatrix} + \begin{bmatrix} 20 & 4 \\ 20 & 8 \end{bmatrix} \cdot \begin{bmatrix} 8 & 20 \\ 18 & 9 \end{bmatrix} \\ \begin{bmatrix} 1 & 10 \\ 3 & 19 \end{bmatrix} \cdot \begin{bmatrix} 4 & 12 \\ 4 & 6 \end{bmatrix} + \begin{bmatrix} 15 & 2 \\ 3 & 14 \end{bmatrix} \cdot \begin{bmatrix} 13 & 8 \\ 3 & 11 \end{bmatrix} & \begin{bmatrix} 1 & 10 \\ 3 & 19 \end{bmatrix} \cdot \begin{bmatrix} 9 & 1 \\ 11 & 2 \end{bmatrix} + \begin{bmatrix} 15 & 2 \\ 3 & 14 \end{bmatrix} \cdot \begin{bmatrix} 8 & 20 \\ 18 & 9 \end{bmatrix} \end{bmatrix}$$

# Blocked Matrix Multiplication

- Decompose matrix into blocks of length $T$ (recall that $T^2 \leq M/3$)
- Do a normal $n/T \times n/T$ matrix multiplication



# Blocked Matrix Multiplication Pseudocode

```
MatrixMultiply(A, B, C, n, T):
   for i = 1 to n/T:
     for j = 1 to n/T:
      for k = 1 to n/T:
        A’ = TxT matrix with upper left corner A[Ti][Tk]
        B’ = TxT matrix with upper left corner B[Tk][Tj]
        C’ = TxT matrix with upper left corner C[Ti][Tj]
        BlockMultiply(A’, B’, C’, T)

BlockMultiply(A, B, C, n):
   for i = 1 to n:
      for j = 1 to n:
         for k = 1 to n:
            C[i][j] += A[i][k] + B[k][j]
```

Let’s analyze the cost of this algorithm in the EM model

## Analysis

- Creating $A'$, $B'$, $C'$ and passing them to `BlockMultiply` all can be done in $O(T^2/B + T)$ cache misses. If $B = O(M^2)$ then we can simplify this to $O(M/B)$. (Called the "tall cache assumption.")
- `BlockMultiply` only accesses elements of $A'$, $B'$, $C'$. Since all three matrices are in cache, it requires zero additional cache misses
- Therefore, our total running time is the number of loop iterations times the cost of a loop. This is $O((n/T)^3 \cdot (T^2/B)) = O((n/\sqrt{M})^3 \cdot (M/B)) = O(n^3/B\sqrt{M})$.

# Implementation questions!

- What do we do if $n$ is not divisible by $T$?
  - Easy answer: pad it out! Doesn't change asymptotics.
  - Can carefully make it work without padding as well
- How do we figure out $M$? We don't have a two-level cache and we're ignoring that space is used for other programs, other variables, etc.
  - Experiment! Try different values of $M$ and see what's fastest on a particular machine.
- Is blocking actually worthwhile?
  - Yes; it is used all the time to speed up programs with poor cache performance.
  - (Not a panacea; some programs (like linear scan, binary search) can't be blocked.)

# Sorting in External Memory

## What about algorithms we know?

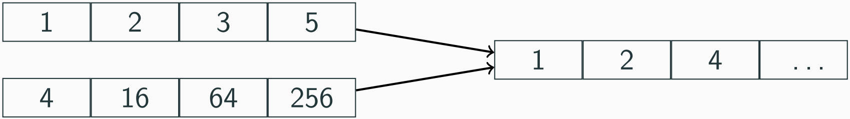
- How long does Mergesort take in external memory?
- Merge is $O(n/B)$; base case is when $n = B$, so total is $n/B \log_2 n/B$.
- How about quicksort?
- Essentially same; partition is $O(n/B)$; total is $n/B \log_2 n/B$.
- Heapsort is $n \log_2 n/B$ unless we're careful...
- Can we do better?

# Using the cache

- Blocking? A little unclear. (We'll come back to this.)
- Does anyone know the sorting lower bound? Where does $n \log n$ come from?
- Answer: each time you compare two numbers, can only have two outcomes.
- Each time we bring a cache line into cache, how many more things can we compare it to?

# Merge sort reminder

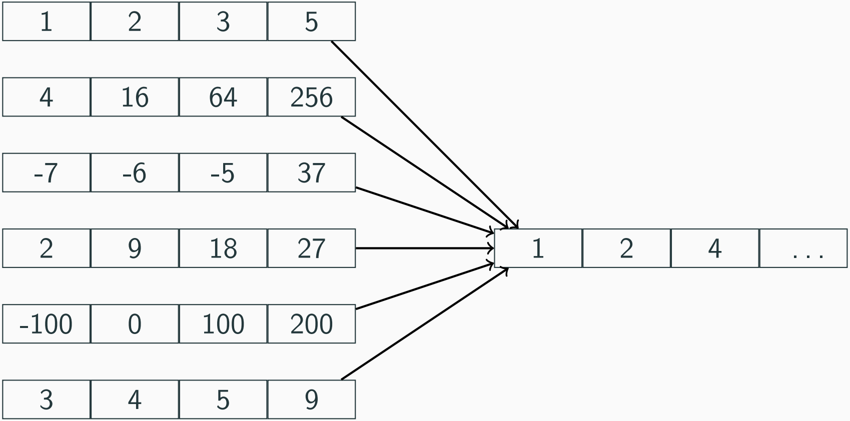
- Divide array into two equal parts
- Recursively sort both parts
- Merge them in $O(n)$ time (and $O(n/B)$ cache misses)



# $M/B$-way merge sort

- Divide array into $M/B$ equal parts
- Recursively sort all $M/B$ parts
- Merge all $M/B$ arrays in $O(n)$ time (and $O(n/B)$ cache misses)

# Diagram of $M/B$-way merge sort



# More Detail on merges

- Keep $B$ slots for each array in cache. ($M/B$ arrays so this fits!)
- When all $B$ slots are empty for the array, take $B$ more items from the array in cache.
- Example on board

# Analysis

- Divide array into $M/B$ parts; combine in $O(N/B)$ cache misses.
- Recursion:

$$T(N) = T(N/(M/B)) + O(N/B)T(B) = O(1)$$

- Solves to $O(\frac{n}{B} \log_{M/B} n/B)$ cache misses
- Optimal!

# Useful?

- Can be useful if your data is VERY large
- Distribution sort: similar idea, but with Quicksort instead of Mergesort
- Another method is most popular in practice: Timsort

## Timsort

- Developed to be the sorting method for python
- Now also used in Java, Rust
- Keeps cache in mind, but focuses more on taking advantage of easy patterns in data

# Blocking revisited: run generation

- Basic idea: sort all $M$-sized subarrays. That would give us sorted subarrays of length $M$ to start out with
- This is wasteful, as we empty out cache between each subarray
- Timsort starts with "run generation": a greedy version of this that uses the same cache for as long as possible. Always outputs sorted runs of length at least $M$; can be MUCH longer

# Timsort after run generation

- First, run generation
- Then, super optimized (2-way) merge sort
- Insertion sort on any very small arrays that are encountered (size $< 64$)

# External Memory Sorting

- $M/B$ way merge sort is most efficient

- Timsort is very popular in practice; uses a simpler blocking approach to stay cache-friendly.