CS61C: Great Ideas in Computer Architecture (aka Machine Structures)

Lecture 30: Caches Part 2

Instructors: Dan Garcia, Justin Yokota

Computing in the News

Ransomware crooks are exploiting IBM file-exchange bug with a 9.8 severity

"The IBM Aspera Faspex is a centralized file-exchange application that large organizations use to transfer large files or large volumes of files at very high speeds. Rather than relying on TCP-based technologies such as FTP to move files, Aspera uses IBM's proprietary FASP—short for Fast, Adaptive, and Secure Protocol—to better utilize available network bandwidth. The product also provides fine-grained management that makes it easy for users to send files to a list of recipients in distribution lists or shared inboxes or workgroups, giving transfers a workflow that's similar to email.

In late January, IBM warned of a critical vulnerability in Aspera versions 4.4.2 Patch Level 1 and earlier and urged users to install an update to patch the flaw. Tracked as CVE-2022-47986, the vulnerability makes it possible for unauthenticated threat actors to remotely execute malicious code by sending specially crafted calls to an outdated programming interface. The ease of exploiting the vulnerability and the damage that could result earned CVE-2022-47986 a severity rating of 9.8 out of a possible 10."

Agenda

- Matrix Multiply Example
- Eviction Policy
- Write-Back/Write-Through Caches
- Analyzing Caches

Caching Example: Matrix Multiply

- For today, we'll be using Matrix Multiply as an example.
- Assumptions:
  - Elements are 4 bytes
    - But only the bottom byte will be shown in the cache for space
  - Block size is 16 bytes (one row of this matrix)
  - Matrix A stored at 0x1000
  - Matrix B stored at 0x2000
  - Matrix C stored at 0x3000
  - No other memory used
  - Caches start cold
- Let's see how this works on a fully associative cache with 4 blocks

B

17	18	19	20
21	22	23	24
25	26	27	28
29	30	31	32

A

1	2	3	4
5	6	7	8
9	10	11	12
13	14	15	16

C

Caching Example: Fully Associative

- Start: Cache starts cold
  - Note: random data in the cache, but all valid bits are zero
- 0 misses, 0 hits

Tag	Valid	Data
0x100	0	0xBF 0x16 0x88 0x2B
0x733	0	0x3B 0x18 0xF1 0xB3
0x156	0	0xE6 0x57 0x49 0xEE
0x4E9	0	0xB5 0x81 0x67 0x3F

B

17	18	19	20
21	22	23	24
25	26	27	28
29	30	31	32

A

1	2	3	4
5	6	7	8
9	10	11	12
13	14	15	16

C

Caching Example: Fully Associative

- Access A[0] = 0x1000
  - Miss: Data isn't valid, so pull from main memory
- 1 miss, 0 hits

Tag	Valid	Data
0x100	0	0xBF 0x16 0x88 0x2B
0x733	0	0x3B 0x18 0xF1 0xB3
0x156	0	0xE6 0x57 0x49 0xEE
0x4E9	0	0xB5 0x81 0x67 0x3F

B

17	18	19	20
21	22	23	24
25	26	27	28
29	30	31	32

A

1	2	3	4
5	6	7	8
9	10	11	12
13	14	15	16

C

Caching Example: Fully Associative

- Access B[0] = 0x2000
  - Miss: Data isn't valid, so pull from main memory
- 2 miss, 0 hits

B

17	18	19	20
21	22	23	24
25	26	27	28
29	30	31	32

Tag	Valid	Data
0x100	1	0x01 0x02 0x03 0x04
0x733	0	0x3B 0x18 0xF1 0xB3
0x156	0	0xE6 0x57 0x49 0xEE
0x4E9	0	0xB5 0x81 0x67 0x3F

A

1	2	3	4
5	6	7	8
9	10	11	12
13	14	15	16

C

Caching Example: Fully Associative

- Access A[1]: 0x1004
  - Hit
- Access B[4]: 0x2010
  - Miss
- 3 Misses, 1 Hit

Tag	Valid	Data
0x100	1	0x01 0x02 0x03 0x04
0x200	1	0x11 0x12 0x13 0x14
0x156	0	0xE6 0x57 0x49 0xEE
0x4E9	0	0xB5 0x81 0x67 0x3F

B

17	18	19	20
21	22	23	24
25	26	27	28
29	30	31	32

A

1	2	3	4
5	6	7	8
9	10	11	12
13	14	15	16

C

Caching Example: Fully Associative

- Access A[2]: 0x1008
  - Hit
- Access B[8]: 0x2020
  - Miss
- 4 Misses, 2 Hit

Tag	Valid	Data
0x100	1	0x01 0x02 0x03 0x04
0x200	1	0x11 0x12 0x13 0x14
0x201	1	0x15 0x16 0x17 0x18
0x4E9	0	0xB5 0x81 0x67 0x3F

B

17	18	19	20
21	22	23	24
25	26	27	28
29	30	31	32

A

1	2	3	4
5	6	7	8
9	10	11	12
13	14	15	16

C

Caching Example: Fully Associative

- Access A[3]: 0x100C
  - Hit
- Access B[12]: 0x2030
  - Miss
- Problem: The cache is full
  - Which block do we evict to make space?

Tag	Valid	Data
0x100	1	0x01 0x02 0x03 0x04
0x200	1	0x11 0x12 0x13 0x14
0x201	1	0x15 0x16 0x17 0x18
0x202	1	0x19 0x1A 0x1B 0x1C

B

17	18	19	20
21	22	23	24
25	26	27	28
29	30	31	32

A

1	2	3	4
5	6	7	8
9	10	11	12
13	14	15	16

C

Eviction Policies

- Our cache is necessarily smaller than main memory, so we will eventually reach a point where our cache is full, and we need to evict some block.
- Ideally, choose an eviction policy that evicts the least useful data. We have several options:
  - LRU
  - MRU
  - FIFO
  - LIFO
  - Random

Eviction Policies

- Least Recently Used (LRU): evict the block in the set that is the oldest previous access.
  - Pro: temporal locality
    - recent past use implies likely future use: in fact, this is a very effective policy
  - Con: Requires complicated hardware and much time to keep track of this
- Most Recently Used (MRU): evict the block in the set that is the newest previous access.
- First-In-First-Out (FIFO): evict the oldest block in the set (queue).
- Last-In-First-Out (LIFO): evict the newest block in the set (stack).
  - Both FIFO and LIFO ignore order of accesses after/before the first/last access, but they serve as good approximations to LRU and MRU without adding too much excess hardware
- Random: randomly selects a block to evict
  - Works surprisingly okay, especially given a low temporal locality workload
  - Does not perform exceedingly well, but also not exceedingly poorly

Eviction Policies

- Our cache is necessarily smaller than main memory, so we will eventually reach a point where our cache is full, and we need to evict some block.
- Ideally, choose an eviction policy that evicts the least useful data. We have several options
- For now: Let's restart our example assuming we use a LRU cache

Caching Example: Fully Associative with LRU

- Start: Cache starts cold
  - Note: New LRU field containing which block accessed first
    - Lower number = more recent for today
- 0 misses, 0 hits

Tag	Valid	LRU	Data
0x100	0	0	0xBF 0x16 0x88 0x2B
0x733	0	0	0x3B 0x18 0xF1 0xB3
0x156	0	0	0xE6 0x57 0x49 0xEE
0x4E9	0	0	0xB5 0x81 0x67 0x3F

B

17	18	19	20
21	22	23	24
25	26	27	28
29	30	31	32

A

1	2	3	4
5	6	7	8
9	10	11	12
13	14	15	16

C

Caching Example: Fully Associative with LRU

- Access A[0] = 0x1000
  - Miss

Tag	Valid	LRU	Data
0x100	0	0	0xBF 0x16 0x88 0x2B
0x733	0	0	0x3B 0x18 0xF1 0xB3
0x156	0	0	0xE6 0x57 0x49 0xEE
0x4E9	0	0	0xB5 0x81 0x67 0x3F

B

17	18	19	20
21	22	23	24
25	26	27	28
29	30	31	32

A

1	2	3	4
5	6	7	8
9	10	11	12
13	14	15	16

C

Caching Example: Fully Associative with LRU

- Access B[0] = 0x2000
  - Miss

B

17	18	19	20
21	22	23	24
25	26	27	28
29	30	31	32

Tag	Valid	LRU	Data
0x100	1	1	0x01 0x02 0x03 0x04
0x733	0	0	0x3B 0x18 0xF1 0xB3
0x156	0	0	0xE6 0x57 0x49 0xEE
0x4E9	0	0	0xB5 0x81 0x67 0x3F

A

1	2	3	4
5	6	7	8
9	10	11	12
13	14	15	16

C

Caching Example: Fully Associative with LRU

- Access A[1]: 0x1004
  - Hit
- Access B[4]: 0x2010
  - Miss

Tag	Valid	LRU	Data
0x100	1	2	0x01 0x02 0x03 0x04
0x200	1	1	0x11 0x12 0x13 0x14
0x156	0	0	0xE6 0x57 0x49 0xEE
0x4E9	0	0	0xB5 0x81 0x67 0x3F

B

17	18	19	20
21	22	23	24
25	26	27	28
29	30	31	32

A

1	2	3	4
5	6	7	8
9	10	11	12
13	14	15	16

C

Caching Example: Fully Associative with LRU

- Access A[2]: 0x1008
  - Hit
- Access B[8]: 0x2020
  - Miss
- Access A[3]: 0x100C
  - Hit

Tag	Valid	LRU	Data
0x100	1	2	0x01 0x02 0x03 0x04
0x200	1	3	0x11 0x12 0x13 0x14
0x201	1	1	0x15 0x16 0x17 0x18
0x4E9	0	0	0xB5 0x81 0x67 0x3F

B

17	18	19	20
21	22	23	24
25	26	27	28
29	30	31	32

A

1	2	3	4
5	6	7	8
9	10	11	12
13	14	15	16

C

Caching Example: Fully Associative with LRU

- Access B[12]: 0x2030
  - Miss
  - Cache is full, so evict the block with tag 0x200

Tag	Valid	LRU	Data
0x100	1	2	0x01 0x02 0x03 0x04
0x200	1	4	0x11 0x12 0x13 0x14
0x201	1	3	0x15 0x16 0x17 0x18
0x202	1	1	0x19 0x1A 0x1B 0x1C

B

17	18	19	20
21	22	23	24
25	26	27	28
29	30	31	32

A

1	2	3	4
5	6	7	8
9	10	11	12
13	14	15	16

C

Caching Example: Fully Associative with LRU

- Note: At this point, we've gotten a lot of misses, especially since we just evicted 0x2004, even though we never used it.
  - If we keep going, 60 hits and 68 misses from A and B alone
  - Can be resolved if we transpose B before running this code

B

17	18	19	20
21	22	23	24
25	26	27	28
29	30	31	32

Tag	Valid	LRU	Data
0x100	1	3	0x01 0x02 0x03 0x04
0x203	1	1	0x1D 0x1E 0x1F 0x20
0x201	1	4	0x15 0x16 0x17 0x18
0x202	1	2	0x19 0x1A 0x1B 0x1C

A

1	2	3	4
5	6	7	8
9	10	11	12
13	14	15	16

C

Caching Example: Fully Associative with LRU

- Write to C[0] = 0x3000
  - Miss, so retrieve from memory
- Problem: How to keep the cache and main memory consistent when you write data?

Tag	Valid	LRU	Data
0x100	1	3	0x01 0x02 0x03 0x04
0x203	1	1	0x1D 0x1E 0x1F 0x20
0x201	1	4	0x15 0x16 0x17 0x18
0x202	1	2	0x19 0x1A 0x1B 0x1C

B

17	18	19	20
21	22	23	24
25	26	27	28
29	30	31	32

A

1	2	3	4
5	6	7	8
9	10	11	12
13	14	15	16

C

Write Policy

- Write-through
  - When a write occurs, update the data both in the cache and in main memory
  - Slows down writes a lot, since we need to write to main memory every time
- Write-back
  - When a write occurs, only update the data in the cache
  - Purposefully let main memory go "stale", and rely on the cache as the main source of truth
  - When we later evict the block, write all changes to main memory at the same time
  - Needs a "dirty" bit to signify that the block has to be written to memory
  - Also means we need to be careful that we don't access main memory before we write back our data-more on this Friday
  - Much faster than write-through, and also extends computer lifespan, since RAM tends to degrade with each write.
- Let's rerun the simulation with a write-back cache, and after we transpose B.

Caching Example: Write-Back Fully Associative with LRU

- Start: Cache starts cold
  - Note: New Dirty field
  - Note: Matrix B has been transposed into Bt to optimize cache efficiency
- 0 misses, 0 hits

Tag	V	Dirty	LRU	Data
0x100	0	0	0	0xBF 0x16 0x88 0x2B
0x733	0	0	0	0x3B 0x18 0xF1 0xB3
0x156	0	0	0	0xE6 0x57 0x49 0xEE
0x4E9	0	0	0	0xB5 0x81 0x67 0x3F

Bt

17	21	25	29
18	22	26	30
19	23	27	31
20	24	28	32

A

1	2	3	4
5	6	7	8
9	10	11	12
13	14	15	16

C

Caching Example: Write-Back Fully Associative with LRU

- Access A[0] = 0x1000
  - Miss
- Access Bt[0] = 0x2000
  - Miss

Tag	V	Dirty	LRU	Data
0x100	0	0	0	0xBF 0x16 0x88 0x2B
0x733	0	0	0	0x3B 0x18 0xF1 0xB3
0x156	0	0	0	0xE6 0x57 0x49 0xEE
0x4E9	0	0	0	0xB5 0x81 0x67 0x3F

Bt

17	21	25	29
18	22	26	30
19	23	27	31
20	24	28	32

A

1	2	3	4
5	6	7	8
9	10	11	12
13	14	15	16

C

Caching Example: Write-Back Fully Associative with LRU

- A[1]: Hit Bt[1]: Hit
- A[2]: Hit Bt[2]: Hit
- A[3]: Hit Bt[3]: Hit
- 6 Hits, 2 Misses!

Bt

17	21	25	29
18	22	26	30
19	23	27	31
20	24	28	32

Tag	V	Dirty	LRU	Data
0x100	1	0	2	0x01 0x02 0x03 0x04
0x200	1	0	1	0x11 0x15 0x19 0x1D
0x156	0	0	0	0xE6 0x57 0x49 0xEE
0x4E9	0	0	0	0xB5 0x81 0x67 0x3F

A

1	2	3	4
5	6	7	8
9	10	11	12
13	14	15	16

C

Caching Example: Write-Back Fully Associative with LRU

- Write to C[0] = 0x3000
  - Miss + Write, so set dirty bit
- Main memory not updated yet

Bt

17	21	25	29
18	22	26	30
19	23	27	31
20	24	28	32

Tag	V	Dirty	LRU	Data
0x100	1	0	2	0x01 0x02 0x03 0x04
0x200	1	0	1	0x11 0x15 0x19 0x1D
0x156	0	0	0	0xE6 0x57 0x49 0xEE
0x4E9	0	0	0	0xB5 0x81 0x67 0x3F

A

1	2	3	4
5	6	7	8
9	10	11	12
13	14	15	16

C

Caching Example: Write-Back Fully Associative with LRU

- A[0]: Hit Bt[4]: Miss … A[3]: Hit Bt[7]: Hit. Write C[1]: Hit
- 8 Hits, 1 Miss here
- Totals: 4 Misses, 14 Hits

Bt

17	21	25	29
18	22	26	30
19	23	27	31
20	24	28	32

Tag	V	Dirty	LRU	Data
0x100	1	0	3	0x01 0x02 0x03 0x04
0x200	1	0	2	0x11 0x15 0x19 0x1D
0x300	1	1	1	0xFA 0x?? 0x?? 0x??
0x4E9	0	0	0	0xB5 0x81 0x67 0x3F

A

1	2	3	4
5	6	7	8
9	10	11	12
13	14	15	16

C

Caching Example: Write-Back Fully Associative with LRU

- A[0]: Hit Bt[8]: Miss, evict block 0x200, … , Write C[2]: Hit
- 8 Hits, 1 Miss here
- Totals: 5 Misses, 22 Hits

Tag	V	Dirty	LRU	Data
0x100	1	0	3	0x01 0x02 0x03 0x04
0x200	1	0	4	0x11 0x15 0x19 0x1D
0x300	1	1	1	0xFA 0x04 0x?? 0x??
0x201	1	0	2	0x12 0x16 0x1A 0x1E

A

1	2	3	4
5	6	7	8
9	10	11	12
13	14	15	16

Bt

17	21	25	29
18	22	26	30
19	23	27	31
20	24	28	32

C

Caching Example: Write-Back Fully Associative with LRU

- Compute C[3]: 8 hits, 1 miss, block 0x201 evicted
- A[4]: Miss, Block 0x202 evicted
- 8 Hits, 2 Misses here
- Totals: 7 Misses, 30 Hits

Tag	V	Dirty	LRU	Data
0x100	1	0	3	0x01 0x02 0x03 0x04
0x202	1	0	2	0x13 0x17 0x1B 0x1F
0x300	1	1	1	0xFA 0x04 0x0E 0x??
0x201	1	0	4	0x12 0x16 0x1A 0x1E

Bt

17	21	25	29
18	22	26	30
19	23	27	31
20	24	28	32

A

1	2	3	4
5	6	7	8
9	10	11	12
13	14	15	16

C

Caching Example: Write-Back Fully Associative with LRU

- Bt[0]: Miss, evict block 0x100
- A[5]-Bt[3]: 6 hits
- Total: 8 misses, 36 hits

Bt

17	21	25	29
18	22	26	30
19	23	27	31
20	24	28	32

Tag	V	Dirty	LRU	Data
0x100	1	0	4	0x01 0x02 0x03 0x04
0x101	1	0	1	0x05 0x06 0x07 0x08
0x300	1	1	2	0xFA 0x04 0x0E 0x18
0x203	1	0	3	0x14 0x18 0x1C 0x20

A

1	2	3	4
5	6	7	8
9	10	11	12
13	14	15	16

C

Caching Example: Write-Back Fully Associative with LRU

- Write C[4]: Miss, Write to dirty block, evict block 0x203
- A[4]: Hit
- Total: 9 misses, 37 hits

Tag	V	Dirty	LRU	Data
0x200	1	0	1	0x11 0x15 0x19 0x1D
0x101	1	0	2	0x05 0x06 0x07 0x08
0x300	1	1	3	0xFA 0x04 0x0E 0x18
0x203	1	0	4	0x14 0x18 0x1C 0x20

A

1	2	3	4
5	6	7	8
9	10	11	12
13	14	15	16

Bt

17	21	25	29
18	22	26	30
19	23	27	31
20	24	28	32

C

Caching Example: Write-Back Fully Associative with LRU

- Bt[4]: Miss, evict 0x300
  - Only at this point does main memory get updated
- Total: 10 misses, 37 hits, 1 write

Bt

17	21	25	29
18	22	26	30
19	23	27	31
20	24	28	32

Tag	V	Dirty	LRU	Data
0x200	1	0	3	0x11 0x15 0x19 0x1D
0x101	1	0	1	0x05 0x06 0x07 0x08
0x300	1	1	4	0xFA 0x04 0x0E 0x18
0x301	1	1	2	0x6A 0x?? 0x?? 0x??

A

1	2	3	4
5	6	7	8
9	10	11	12
13	14	15	16

C

Caching Example: Write-Back Fully Associative with LRU

- A[5]-Write C[5]: 7 hits
- Compute C[6]: 1 miss, 8 hits
- Compute C[7]: 1 miss, 8 hits
- Total: 12 misses, 60 hits, 1 write

Bt

17	21	25	29
18	22	26	30
19	23	27	31
20	24	28	32

Tag	V	Dirty	LRU	Data
0x200	1	0	4	0x11 0x15 0x19 0x1D
0x101	1	0	2	0x05 0x06 0x07 0x08
0x201	1	0	1	0x12 0x16 0x1A 0x1E
0x301	1	1	3	0x6A 0x84 0x9E 0xB8

A

1	2	3	4
5	6	7	8
9	10	11	12
13	14	15	16

C

250	260	270	280

Caching Example: Write-Back Fully Associative with LRU

- Compute C[8]: 3 misses, 6 hits
- Compute C[9]: 1 miss, 8 hits, 1 write
- Compute C[10]: 1 miss, 8 hits
- Compute C[11]: 1 miss, 8 hits
- Total: 18 misses, 90 hits, 2 writes

Bt

17	21	25	29
18	22	26	30
19	23	27	31
20	24	28	32

Tag	V	Dirty	LRU	Data
0x202	1	0	4	0x13 0x17 0x1B 0x1F
0x101	1	0	3	0x05 0x06 0x07 0x08
0x203	1	0	2	0x14 0x18 0x1C 0x20
0x301	1	1	1	0x6A 0x84 0x9E 0xB8

A

1	2	3	4
5	6	7	8
9	10	11	12
13	14	15	16

C

250	260	270	280

Caching Example: Write-Back Fully Associative with LRU

- Compute C[12]: 3 misses, 6 hits
- Compute C[13]: 1 miss, 8 hits, 1 write
- Compute C[14]: 1 miss, 8 hits
- Compute C[15]: 1 miss, 8 hits
- Total: 24 misses, 120 hits, 3 writes

Bt

17	21	25	29
18	22	26	30
19	23	27	31
20	24	28	32

Tag	V	Dirty	LRU	Data
0x102	1	0	3	0x09 0x0A 0x0B 0x0C
0x202	1	0	4	0x13 0x17 0x1B 0x1F
0x302	1	1	1	0xDA 0x04 0x2E 0x58
0x203	1	0	2	0x14 0x18 0x1C 0x20

A

1	2	3	4
5	6	7	8
9	10	11	12
13	14	15	16

C

250	260	270	280
618	658	670	696

Caching Example: Write-Back Fully Associative with LRU

- Final step: Flush the cache
  - Write back all dirty blocks
  - Invalidate all blocks
- Final Count: 24 misses, 120 hits, 4 writes

Tag	V	Dirty	LRU	Data
0x202	1	0	4	0x13 0x17 0x1B 0x1F
0x103	1	0	3	0x0D 0x0E 0x0F 0x10
0x203	1	0	2	0x14 0x18 0x1C 0x20
0x303	1	1	1	0x4A 0x84 0xBE 0xF8

Bt

17	21	25	29
18	22	26	30
19	23	27	31
20	24	28	32

A

1	2	3	4
5	6	7	8
9	10	11	12
13	14	15	16

C

250	260	270	280
618	658	670	696
986	1028	1070	1112

Caching Example: Write-Back Fully Associative with LRU

- Final Count: 24 misses, 120 hits, 4 writes
- 83% hit rate
  - Way better than the 50% we got before

Tag	V	Dirty	LRU	Data
0x202	0	0	4	0x13 0x17 0x1B 0x1F
0x103	0	0	3	0x0D 0x0E 0x0F 0x10
0x203	0	0	2	0x14 0x18 0x1C 0x20
0x303	0	0	1	0x4A 0x84 0xBE 0xF8

Bt

17	21	25	29
18	22	26	30
19	23	27	31
20	24	28	32

A

1	2	3	4
5	6	7	8
9	10	11	12
13	14	15	16

C

250	260	270	280
618	658	670	696
986	1028	1070	1112
1354	1412	1470	1528

Analyzing Cache Performance

- The **hit rate** of a cache is the percentage of accesses that result in a hit
- The **miss rate** of a cache is the percentage of accesses that result in a miss
- **Hit time** is how long it takes to check the cache. The **Miss Penalty** is how long it takes to access main memory after a miss.
  - If we get a cache hit, runtime is equal to hit time
  - If we get a cache miss, runtime is equal to hit time + miss penalty
  - If we didn't have a cache, we'd always take miss penalty time to access main memory
- Analogy: Hit time = how long to check the bookshelf, Miss penalty = how long to go to the library
- In the previous example, our hit rate was ⅚ = 83.33%, so our miss rate was ⅙ = 16.66%
- The **Average Memory Access Time** (or AMAT) is the average amount of time it takes for one memory access, given a hit rate.

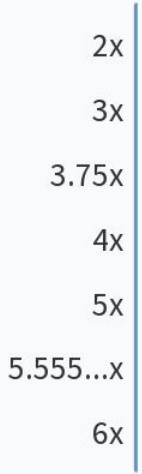
In the matrix multiplication example, we had a 5/6 hit rate. If our hit time was 10 cycles and our miss penalty was 100 cycles, what speedup was attained by our cache?

- 2x
- 3x
- 3.75x
- 4x
- 5x
- 5.555...x
- 6x

In the matrix multiplication example, we had a 5/6 hit rate. If our hit time was 10 cycles and our miss penalty was 100 cycles, what speedup was attained by our cache?

- 2x
- 3x
- 3.75x
- 4x
- 5x
- 5.555...x
- 6x

In the matrix multiplication example, we had a 5/6 hit rate. If our hit time was 10 cycles and our miss penalty was 100 cycles, what speedup was attained by our cache?



Analyzing Cache Performance

- ⅚ of our memory accesses were hits, and therefore took 10 cycles each
- ⅙ of our memory accesses were misses, and therefore took 100+10=110 cycles each
- 5 hits + 1 miss is 160 total cycles, so AMAT is 160/6 cycles
- Alternate formula: AMAT = hit time + (miss rate * miss penalty)
  - AMAT = 10+(⅙*100) = 160/6
- Our original runtime would have been 100 cycles per access
- Speedup is 100/(160/6) = 600/160 = 3.75x

Computing Hit Rate

- Computing AMAT can get tricky for certain programs
  - Main difficulty is in determining the hit rate of a program
- General Tips:
  - Generally speaking, the hit rate starts low, because the cache is cold
  - After warming up, the cache normally reaches a "steady state", where a pattern of hits and misses appears every iteration of some outer loop, until we completely exhaust the memory in a block
  - Once we start a new block, the cache acts cold again, and the cycle repeats.
  - Once you find this cycle, it's possible to use that to get the total hit rate.
  - This is a heuristic (not guaranteed), though, so you need to be rigorous with showing that your pattern will actually hold
- More practice: Homework