Practical Caches
COMP 252 - Lecture 3

Antoniu Pop

antoniu.pop@manchester.ac.uk

5 February 2019
Previous Lecture: How/why caches work

- Locality

- Temporal & Spatial

- What parameters have an impact on locality

- Spatial: cache line size (how much data is stored in each entry)

- Temporal: Replacement policy

- Associativity

  - Fully associative: good for temporal locality / expensive
  - Direct mapped: bad for temporal locality / cheap
  - Set associative: compromise

- Cache Replacement Policy

  - LRU, cyclic

- Read vs. Write behaviour

  - write through / back
  - write allocate / around
Previous Lecture: How/why caches work

- **Locality**
  - Temporal & Spatial

- **Spatial**: cache line size (how much data is stored in each entry)
- **Temporal**: replacement policy
  - Fully associative: good for temporal locality / expensive
  - Direct mapped: bad for temporal locality / cheap
  - Set associative: compromise

- **Cache Replacement Policy**
  - LRU, cyclic, random

- **Read vs. Write behaviour**
  - write through / back
  - write allocate / around
Previous Lecture: How/why caches work

- **Locality**
  - Temporal & Spatial
  - What parameters have an impact on locality
Locality

Temporal & Spatial

What parameters have an impact on locality

Spatial: cache line size (how much data is stored in each entry)
Previous Lecture: How/why caches work

- **Locality**
  - Temporal & Spatial
  - What parameters have an impact on locality
    - Spatial: cache line size (how much data is stored in each entry)
    - Temporal: Replacement policy
Previous Lecture: How/why caches work

- **Locality**
  - Temporal & Spatial
  - What parameters have an impact on locality
  - Spatial: cache line size (how much data is stored in each entry)
  - Temporal: Replacement policy

- **Associativity**
Previous Lecture: How/why caches work

- **Locality**
  - Temporal & Spatial
  - What parameters have an impact on locality
  - Spatial: cache line size (how much data is stored in each entry)
  - Temporal: Replacement policy

- **Associativity**
  - Fully associative: good for temporal locality / expensive
Previous Lecture: How/why caches work

- **Locality**
  - Temporal & Spatial
  - What parameters have an impact on locality
  - Spatial: cache line size (how much data is stored in each entry)
  - Temporal: Replacement policy

- **Associativity**
  - Fully associative: good for temporal locality / expensive
  - Direct mapped: bad for temporal locality / cheap
Previous Lecture: How/why caches work

▶ Locality
  ▶ Temporal & Spatial
  ▶ What parameters have an impact on locality
  ▶ Spatial: cache line size (how much data is stored in each entry)
  ▶ Temporal: Replacement policy

▶ Associativity
  ▶ Fully associative: good for temporal locality / expensive
  ▶ Direct mapped: bad for temporal locality / cheap
  ▶ Set associative: compromise
Previous Lecture: How/why caches work

- **Locality**
  - Temporal & Spatial
  - What parameters have an impact on locality
  - Spatial: cache line size (how much data is stored in each entry)
  - Temporal: Replacement policy

- **Associativity**
  - Fully associative: good for temporal locality / expensive
  - Direct mapped: bad for temporal locality / cheap
  - Set associative: compromise

- **Cache Replacement Policy**
Previous Lecture: How/why caches work

- **Locality**
  - Temporal & Spatial
  - What parameters have an impact on locality
  - Spatial: cache line size (how much data is stored in each entry)
  - Temporal: Replacement policy

- **Associativity**
  - Fully associative: good for temporal locality / expensive
  - Direct mapped: bad for temporal locality / cheap
  - Set associative: compromise

- **Cache Replacement Policy**
  - LRU
Previous Lecture: How/why caches work

- **Locality**
  - Temporal & Spatial
  - What parameters have an impact on locality
  - Spatial: cache line size (how much data is stored in each entry)
  - Temporal: Replacement policy

- **Associativity**
  - Fully associative: good for temporal locality / expensive
  - Direct mapped: bad for temporal locality / cheap
  - Set associative: compromise

- **Cache Replacement Policy**
  - LRU, cyclic
Previous Lecture: How/why caches work

- **Locality**
  - Temporal & Spatial
  - What parameters have an impact on locality
  - Spatial: cache line size (how much data is stored in each entry)
  - Temporal: Replacement policy

- **Associativity**
  - Fully associative: good for temporal locality / expensive
  - Direct mapped: bad for temporal locality / cheap
  - Set associative: compromise

- **Cache Replacement Policy**
  - LRU, cyclic, random
Previous Lecture: How/why caches work

- **Locality**
  - Temporal & Spatial
  - What parameters have an impact on locality
  - Spatial: cache line size (how much data is stored in each entry)
  - Temporal: Replacement policy

- **Associativity**
  - Fully associative: good for temporal locality / expensive
  - Direct mapped: bad for temporal locality / cheap
  - Set associative: compromise

- **Cache Replacement Policy**
  - LRU, cyclic, random

- **Read vs. Write behaviour**
Previous Lecture: How/why caches work

- **Locality**
  - Temporal & Spatial
  - What parameters have an impact on locality
  - Spatial: cache line size (how much data is stored in each entry)
  - Temporal: Replacement policy

- **Associativity**
  - Fully associative: good for temporal locality / expensive
  - Direct mapped: bad for temporal locality / cheap
  - Set associative: compromise

- **Cache Replacement Policy**
  - LRU, cyclic, random

- **Read vs. Write behaviour**
  - write through / back
Previous Lecture: How/why caches work

- **Locality**
  - Temporal & Spatial
  - What parameters have an impact on locality
  - Spatial: cache line size (how much data is stored in each entry)
  - Temporal: Replacement policy

- **Associativity**
  - Fully associative: good for temporal locality / expensive
  - Direct mapped: bad for temporal locality / cheap
  - Set associative: compromise

- **Cache Replacement Policy**
  - LRU, cyclic, random

- **Read vs. Write behaviour**
  - write through / back
  - write allocate / around
Today’s Lecture – Learning Objectives

To understand

- Additional Control Bits in Cache Lines
Today’s Lecture – Learning Objectives

To understand

- Additional Control Bits in Cache Lines
- Cache Line Size Tradeoffs
Today’s Lecture – Learning Objectives

To understand

▶ Additional Control Bits in Cache Lines
▶ Cache Line Size Tradeoffs
▶ Separate I&D caches
Today’s Lecture – Learning Objectives

To understand

▶ Additional Control Bits in Cache Lines
▶ Cache Line Size Tradeoffs
▶ Separate I&D caches
▶ Multiple Level Caches
Cache Control Bits

- We have, so far, ignored detail of cache initialisation
Cache Control Bits

- We have, so far, ignored detail of cache initialisation

- At some time it must start empty. We need a valid bit for each entry to indicate meaningful data
Cache Control Bits

- We have, so far, ignored detail of cache initialisation.

- At some time it must start empty. We need a valid bit for each entry to indicate meaningful data.

- We also need a “dirty” bit if we are using “Write Back” rather than “Write Through”
Exploiting Spatial Locality

- Storing and comparing the address or tag (part of address) is expensive
Exploiting Spatial Locality

- Storing and comparing the address or tag (part of address) is expensive
- So far we have assumed that each address relates to a single data item (byte or word)
Exploiting Spatial Locality

- Storing and comparing the address or tag (part of address) is expensive
- So far we have assumed that each address relates to a single data item (byte or word)
- We can use a wider cache “line” and store more data per address/tag
Exploiting Spatial Locality

▶ Storing and comparing the address or tag (part of address) is expensive

▶ So far we have assumed that each address relates to a single data item (byte or word)

▶ We can use a wider cache “line” and store more data per address/tag

▶ Spatial locality suggests we will make use of it (e.g. series of instructions)
Direct Mapped Cache – 2 words per line

- Tag RAM
- Index
- Compare
- hit / miss
- Multiplexer
- Word 0
- Word 1
- data
- address
Multiple Word Line Size

- Now bottom bits of address are used to select which word
Multiple Word Line Size

- Now bottom bits of address are used to select which word
- Can be used with fully or set associative as well
Multiple Word Line Size

- Now bottom bits of address are used to select which word
- Can be used with fully or set associative as well
- Typical line size 16, 32 or 64 bytes (most common)
  - 4, 8 or 16 32-bit words
  - today often 64 bytes (8 64-bit values) in 64 bit architectures
Multiple Word Line Size

- Now bottom bits of address are used to select which word
- Can be used with fully or set associative as well
- Typical line size 16, 32 or 64 bytes (most common)
  - 4, 8 or 16 32-bit words
  - today often 64 bytes (8 64-bit values) in 64 bit architectures
- Transfer from RAM in “blocks”, usually equal to line size
  - use burst mode memory access
The Effect of Line Size

- **Spatial locality**: if we access data, then data close by is likely to be accessed as well
The Effect of Line Size

- **Spatial locality**: if we access data, then data close by is likely to be accessed as well

- So a larger line size means we get that nearby data in the cache and avoid misses
The Effect of Line Size

- **Spatial locality**: if we access data, then data close by is likely to be accessed as well

- So a larger line size means we get that nearby data in the cache and avoid misses

- But if line size is too big
  - Data may not be used
  - Displaces other possibly useful data
  - Larger RAM accesses take longer
Impact of Line Size – typical characteristic

![Impact of Line Size Graph]

- The graph shows the miss rate percentage (%) on the y-axis against different line sizes (words) on the x-axis.
- The miss rate decreases as the line size increases, reaching a minimum at approximately 8 words, and then increases again as the line size continues to increase.

This indicates that there is an optimal line size where the miss rate is minimized, which is crucial for cache performance optimization.
Separate Instruction & Data (I&D) Caches

- Instruction fetch every instruction
- Data fetch every 3 instructions
Separate Instruction & Data (I&D) Caches

- Instruction fetch every instruction
- Data fetch every 3 instructions
- Usually working in separate address areas
Separate Instruction & Data (I&D) Caches

- Instruction fetch every instruction
- Data fetch every 3 instructions
- Usually working in separate address areas
- Access patterns different
  - Instructions accessed in serial sections
  - Can use lower associativity

Better utilization - use separate caches

Called "Harvard" architecture
Separate Instruction & Data (I&D) Caches

- Instruction fetch every instruction
- Data fetch every 3 instructions
- Usually working in separate address areas
- Access patterns different
  - Instructions accessed in serial sections
  - Can use lower associativity

- Better utilization - use separate caches
- Called “Harvard” architecture
Split Level 1 (L1) Caches

CPU

fetch

data

Registers

L1 Inst Cache

L1 Data Cache

RAM Memory

On-chip
Multiple Level Caches (1)

- Bigger caches have lower miss rates
Multiple Level Caches (1)

- Bigger caches have lower miss rates
- As chips get bigger we could build bigger caches to perform better
Multiple Level Caches (1)

- Bigger caches have lower miss rates
- As chips get bigger we could build bigger caches to perform better
- But bigger caches always run slower
Multiple Level Caches (1)

- Bigger caches have lower miss rates
- As chips get bigger we could build bigger caches to perform better
- But bigger caches always run slower
- L1 cache needs to run at processor speed
Multiple Level Caches (1)

- Bigger caches have lower miss rates
- As chips get bigger we could build bigger caches to perform better
- But bigger caches always run slower
- L1 cache needs to run at processor speed
- Instead put another cache (Level 2) between L1 and RAM
Multiple Level Caches (2)

- CPU
- Registers
- L1 Inst Cache
- L1 Data Cache
- L2
- RAM Memory
- On-chip

fetch

register

L1

Inst

Cache

L1

Data

Cache

fetch
data

fetch
data

fetch
data
Multiple Level Caches (3)

- L2 cache is typically $16 \times$ bigger than L1
Multiple Level Caches (3)

- L2 cache is typically 16× bigger than L1
- L2 cache is typically 4× slower than L1
  - But still 10× faster than RAM!
Multiple Level Caches (3)

- L2 cache is typically $16 \times$ bigger than L1
- L2 cache is typically $4 \times$ slower than L1
  - But still $10 \times$ faster than RAM!
- If only 1 in 50 accesses miss in L1 and similar in L2
  - Only have to cover very small number of RAM accesses
Multiple Level Caches (3)

- L2 cache is typically $16 \times$ bigger than L1.
- L2 cache is typically $4 \times$ slower than L1.
  - But still $10 \times$ faster than RAM!
- If only 1 in 50 accesses miss in L1 and similar in L2.
  - Only have to cover very small number of RAM accesses.
- Not quite that easy but works well.
Multiple Level Caches (4)

- Vital to performance of modern processors
Multiple Level Caches (4)

- Vital to performance of modern processors
- L2 is usually shared by L1I and L1D
Multiple Level Caches (4)

- Vital to performance of modern processors
- L2 is usually shared by L1I and L1D
- Replacement strategy and write policy obviously gets more complex
E.g. Xeon E3-1280

- Available 2013Q2
- 4-core, 8-thread
- Core private caches
  - 32Kb L1 I-cache
  - 32Kb L1 D-cache
  - 256Kb L2 cache (I+D)
- 8Mb L3 cache (shared I+D by all cores)
- 2 channel DDR3
Cache Address Splitting

- 8 MB cache

Tag RAM

Compare

hit / miss

Tag

Index

Word ID

Align

Address (32 bits)

Multiplexer

data

Word 0

Word 1

Word 7

? bits

? bits

? bits

? bits
Cache Address Splitting

- 8 MB cache
- 32bit/4byte

Tag RAM

Compare
hit / miss

Multiplexer
data
Cache Address Splitting

- 8 MB cache
- 32bit/4byte
- 8 words/line

Tag RAM

Compare

hit / miss

Tag

Index

Word ID

Align

Address (32 bits)

Multiplexer

Word 0

Word 1

Word 7

Data
Cache Address Splitting

- 8 MB cache
- 32bit/4byte
- 8 words/line
- 256K entries

Tag RAM

Address (32 bits)

? bits 18 bits 3 bits 2 bits

Tag
Index
Word ID
Align

Compare

hit / miss

Multiplexer

Word 0 Word 1

Word 7

data
Cache Address Splitting

- 8 MB cache
- 32bit/4byte
- 8 words/line
- 256K entries

- Tag RAM
- Compare
  - hit / miss
- Multiplexer
  - Word 0
  - Word 1
  - Word 7
  - data
Assume CPU with simple L1 cache only
- L1 cache 98% hit rate
- L1 access time = 1 CPU cycle
- RAM access = 50 cycles
Cache Example

- Assume CPU with simple L1 cache only
  - L1 cache **98%** hit rate
  - L1 access time = 1 CPU cycle
  - RAM access = 50 cycles
  - Suggestion: consider 100 accesses
Cache Example

- Assume CPU with simple L1 cache only
  - L1 cache 98% hit rate
  - L1 access time = 1 CPU cycle
  - RAM access = 50 cycles
  - Suggestion: consider 100 accesses

- What is effective overall memory access time?

- Assume CPU makes a RAM access (fetch) every cycle
Cache Example (solution)

- For every 100 accesses

...
Cache Example (solution)

- For every 100 accesses
  - 98 hit in cache: 98 * 1 cycle = 98 cycles
Cache Example (solution)

- For every 100 accesses
  - 98 hit in cache: 98 * 1 cycle = 98 cycles
  - 2 miss in cache, go to RAM: 2 * (1+50) cycles = 102 cycles

Average access time = 102 cycles / 100 = 1.02 cycles
Cache Example (solution)

► For every 100 accesses
  ► 98 hit in cache: 98 * 1 cycle = 98 cycles
  ► 2 miss in cache, go to RAM: 2 * (1+50) cycles = 102 cycles
  ► Total: 98 + 102 = 200 cycles

Average access time = \( \frac{200}{100} \) = 2 cycles

CPU on average will only run at \( \frac{1}{2} \) speed
Cache Example (solution)

▸ For every 100 accesses
  ▸ 98 hit in cache: 98 * 1 cycle = 98 cycles
  ▸ 2 miss in cache, go to RAM: 2 * (1+50) cycles = 102 cycles
  ▸ Total: 98 + 102 = 200 cycles

▸ Average access time = 200/100 = 2 cycles
Cache Example (solution)

- For every 100 accesses
  - 98 hit in cache: $98 \times 1$ cycle = 98 cycles
  - 2 miss in cache, go to RAM: $2 \times (1+50)$ cycles = 102 cycles
  - Total: $98 + 102 = 200$ cycles

- Average access time = $200/100 = 2$ cycles

- CPU on average will only run at $\frac{1}{2}$ speed
Two Level Cache

- Now assume L2 cache between L1 & RAM
  - Access time = 4 cycles
  - Hit rate = 90%
Two Level Cache

- Now assume L2 cache between L1 & RAM
  - Access time = 4 cycles
  - Hit rate = 90%

- L2: every 100 accesses take
  - \((90 \times 4) + 10 \times (4 + 50) = 900\)
  - Average access = 9 cycles
Two Level Cache Example

- Assume CPU with L1 and L2 cache
  - L1: 98% hit, 1 CPU cycle
  - L2 + RAM: average 9 CPU cycles
Two Level Cache Example

- Assume CPU with L1 and L2 cache
  - L1: 98% hit, 1 CPU cycle
  - L2 + RAM: average 9 CPU cycles
  - Suggestion: consider 100 accesses
Two Level Cache Example

- Assume CPU with L1 and L2 cache
  - L1: 98% hit, 1 CPU cycle
  - L2 + RAM: average 9 CPU cycles
    - Suggestion: consider 100 accesses

- What is effective overall memory access time?

- Assume CPU makes a RAM access (fetch) every cycle
Two Level Cache Example (solution)

- Back to L1
Two Level Cache Example (solution)

- Back to L1
- For every 100 accesses time taken

\[(1 \times 98) + 2 \times (1 + 9) = 118\text{cycles}\]
Two Level Cache Example (solution)

- Back to L1
- For every 100 accesses time taken

\[(1 \times 98) + 2 \times (1 + 9) = 118\text{cycles}\]

- Average access = 1.18 cycles
Two Level Cache Example (solution)

- Back to L1
- For every 100 accesses time taken

\[(1 \times 98) + 2 \times (1 + 9) = 118\text{cycles}\]

- Average access = 1.18 cycles
- Now approx 85% of potential full speed
Alternatively

- 1000 accesses
Alternatively

- 1000 accesses
  - 980 will hit in L1 (98%)
Alternatively

- 1000 accesses
  - 980 will hit in L1 (98%)
  - 18 will hit in L2 (90% of 20)
Alternatively

- 1000 accesses
  - 980 will hit in L1 (98%)
  - 18 will hit in L2 (90% of 20)
  - 2 will go to main memory
Alternatively

► 1000 accesses
  ► 980 will hit in L1 (98%)
  ► 18 will hit in L2 (90% of 20)
  ► 2 will go to main memory

► So access time is

\[
980 + 18 \times (1 + 4) + 2 \times (1 + 4 + 50) = 1180
\]
Alternatively

- 1000 accesses
  - 980 will hit in L1 (98%)
  - 18 will hit in L2 (90% of 20)
  - 2 will go to main memory

- So access time is

\[ 980 + 18 \times (1 + 4) + 2 \times (1 + 4 + 50) = 1180 \]

- Average = 1180/1000 = 1.18