Lecture 2: Caches and Cache Coherence

Teaching assistant: Salvatore Di Girolamo

Motivational video: https://www.youtube.com/watch?v=zJybFF6PqEQ

Scientific integrity – or how to report benchmark results?

1991 – the classic!

2012 – the shocking

2013 – the extension


Scientific Benchmarking: Pitfalls of Relative Performance Reporting (Rule 1)

Most common (and oldest) problem with reporting

- First seen 1988 – also included in Bailey's 12 ways
- Speedups can look arbitrarily good if it's relative to a bad baseline
- Imagine an unoptimized vs. optimized matrix multiplication:
  The optimized MM is 10x faster than the unoptimized!

What does this mean?

- Class question: how could we improve the situation?
- Recently rediscovered in the "big data" universe

A. Rowstron et al.: Nobody ever got fired for using Hadoop on a cluster,
HotCDP 2012

F. McSherry et al.: Scalability! but at what cost?,
HotOS 2015

Both plots show speedups calculated from the same data. The only difference is the baseline.

Rule 1: When publishing parallel speedup, report if the base case is a single parallel process or best serial execution, as well as the absolute execution performance of the base case.

A simple generalization of this rule implies that one should never report ratios without absolute values.

Recent examples from the "big data" universe:

A. Rowstron et al.: Nobody ever got fired for using Hadoop on a cluster, HotCDP 2012
F. McSherry et al.: Scalability! but at what cost?, HotOS 2015

Goals of this lecture

- Memory Trends – Short Refresher on Locality and Caches!
- Cache Coherence in Multiprocessors
- Advanced Memory Consistency
Measure processor speed as "throughput"
- FLOPS/s, IOPS/s, …
- Moore's law - ~60% growth per year

Today's architectures
- POWER8: 425 dp GFLOP/s – 340 GB/s memory bw
- Intel E5-2630 v4: 496 dp GFLOPS/s ~140 GB/s memory bw
- Trend: memory performance grows 10% per year

Memory – CPU gap widens
- How to measure bandwidth?
  - Data sheet (often peak performance, may include overheads)
  - Microbenchmark performance
  - Stride 1 access (32 MiB): 46 GiB/s
  - Random access (8 B out of 32 MiB): 4.7 GiB/s
- Why?
  - Application performance
  - As observed (performance counters)
  - Somewhere in between stride 1 and random access

- How to measure Latency?
  - Data sheet (often optimistic, or not provided)
  - Random pointer chase
  - 28 ns with one core, 75 ns with 10 cores!

Why Caches Work: Locality
- Locality: Programs tend to use data and instructions with addresses near or equal to those they have used recently, cf. "Denning: "The locality principle", CACM’75
- Temporal locality:
  - Recently referenced items are likely to be referenced again in the near future
- Spatial locality:
  - Items with nearby addresses tend to be referenced close together in time

Example: Locality?
```c
sum = 0;
for (i = 0; i < n; i++)
  sum += a[i];
return sum;
```

Conjecture: Buffering/caching is a must!
- Two most common examples:
  - Write Buffers
    - Delayed write back saves memory bandwidth
    - Data is often overwritten or re-read
  - Caching
    - Directory of recently used locations
    - Stored as blocks (cache lines)
- Many others deep in architectures:
  - Translation Lookahead Buffer
  - Branch Predictors
  - Trace Caches
- …

Why Caches Work: Locality
- Locality: Programs tend to use data and instructions with addresses near or equal to those they have used recently, cf. "Denning: "The locality principle", CACM’75
- Temporal locality:
  - Recently referenced items are likely to be referenced again in the near future
- Spatial locality:
  - Items with nearby addresses tend to be referenced close together in time

Issues (Intel Xeon E5-2630 v4 as Example)
- How to measure bandwidth?
  - Data sheet (often peak performance, may include overheads)
  - Microbenchmark performance
  - Stride 1 access (32 MiB): 46 GiB/s
  - Random access (8 B out of 32 MiB): 4.7 GiB/s
- Why?
  - Application performance
  - As observed (performance counters)
  - Somewhere in between stride 1 and random access

- How to measure Latency?
  - Data sheet (often optimistic, or not provided)
  - Random pointer chase
  - 28 ns with one core, 75 ns with 10 cores!

Typical Memory Hierarchy
- Smaller, faster, costlier per byte
- Larger, slower, cheaper per byte
- Remote secondary storage (tapes, distributed file systems, Web servers)

Why Caches Work: Locality
- Locality: Programs tend to use data and instructions with addresses near or equal to those they have used recently, cf. "Denning: "The locality principle", CACM’75
- Temporal locality:
  - Recently referenced items are likely to be referenced again in the near future
- Spatial locality:
  - Items with nearby addresses tend to be referenced close together in time

Example: Locality?
```c
sum = 0;
for (i = 0; i < n; i++)
  sum += a[i];
return sum;
```

Conjecture: Buffering/caching is a must!
- Two most common examples:
  - Write Buffers
    - Delayed write back saves memory bandwidth
    - Data is often overwritten or re-read
  - Caching
    - Directory of recently used locations
    - Stored as blocks (cache lines)
- Many others deep in architectures:
  - Translation Lookahead Buffer
  - Branch Predictors
  - Trace Caches
  - …

Why Caches Work: Locality
- Locality: Programs tend to use data and instructions with addresses near or equal to those they have used recently, cf. "Denning: "The locality principle", CACM’75
- Temporal locality:
  - Recently referenced items are likely to be referenced again in the near future
- Spatial locality:
  - Items with nearby addresses tend to be referenced close together in time

Example: Locality?
```c
sum = 0;
for (i = 0; i < n; i++)
  sum += a[i];
return sum;
```
Cache

- **Definition:** Computer memory with short access time used for the storage of frequently or recently used instructions or data.

- Naturally supports *temporal locality*
- Spatial locality is supported by transferring data in blocks
  - E.g., Intel's Core family: one block = 64 B = 8 doubles

General Cache Organization (S, E, B)

- \( S = 2^s \) sets
- \( E = 2^e \) lines per set
- \( B = 2^b \) bytes per cache block

- \( S x E x B \) data bytes

Locality Example

How to improve locality?

- Ignore the variables \( i, j, k, sum \)

Example (S=4, E=2)

```c
int sum_array_row(double a[2][2][2])
{
    int i, j, k;
    return sum;
}
```

```c
int sum_array_col(double a[2][2][2])
{
    int i, j;
    double sum = 0;
    return sum;
}
```

Cache Structure

- Simplest design: direct mapped!
- Adding 2-way associativity

General Cache Organization (S, E, B)

- \( S = 2^s \) sets
- \( E = 2^e \) lines per set
- \( B = 2^b \) bytes per cache block

- \( S x E x B \) data bytes

Cache size:

- \( S x E x B \) data bytes

Cache Read

- \( S = 2^s \) sets
- \( E = 2^e \) lines per set
- \( B = 2^b \) bytes per cache block

- Valid bit (+ others later)
- \( B = 2^b \) bytes per cache block (the data)

Cache Read

- \( S = 2^s \) sets
- \( E = 2^e \) lines per set
- \( B = 2^b \) bytes per cache block

- Address of word:
- Tag set block index offset
- Data begins at this offset

Example (S=4, E=2)

```c
int sum_array_row(double a[2][2][2])
{
    int i, j, k,
    double sum = 0;
    return sum;
}
```

```c
int sum_array_col(double a[2][2][2])
{
    int i, j;
    double sum = 0;
    return sum;
}
```

Cache Read

- \( S = 2^s \) sets
- \( E = 2^e \) lines per set
- \( B = 2^b \) bytes per cache block

- Address of word:
- Tag set block index offset
- Data begins at this offset

Example (S=4, E=2)

```c
int sum_array_row(double a[2][2][2])
{
    int i, j, k,
    double sum = 0;
    return sum;
}
```

```c
int sum_array_col(double a[2][2][2])
{
    int i, j;
    double sum = 0;
    return sum;
}
```
Direct mapped cache:
- Cache with $E = 1$
- Means every block from memory has a unique location in cache

Fully associative cache
- Cache with $S = 1$ (i.e., maximal $E$)
- Means every block from memory can be mapped to any location in cache
- In practice too expensive to build
- One can view the register file as a fully associative cache

LRU (least recently used) replacement
- When selecting which block should be replaced (happens only for $E > 1$), the least recently used one is chosen

**Compulsory (cold) miss**
- Occurs on first access to a block

**Capacity miss**
- Occurs when the working set is larger than the cache

**Conflict miss**
- Occurs when the cache is large enough, but multiple data objects all map to the same slot
- Not a clean classification but still useful

**What about writes?**

- **What to do on a write-hit?**
  - Write-through: write immediately to memory
  - Write-back: defer write to memory until replacement of line

- **What to do on a write-miss?**
  - Write-allocate: load into cache, update line in cache
  - No-write-allocate: writes immediately to memory

**Types of Cache Misses (The 3 C’s)**

- **Compulsory (cold) miss**
  - Occurs on first access to a block

- **Capacity miss**
  - Occurs when working set is larger than the cache

- **Conflict miss**
  - Conflict misses occur when the cache is large enough, but multiple data objects all map to the same slot
  - Not a clean classification but still useful

**Cache Coherence in Multiprocessors**

- Different caches may have a copy of the same memory location!
- Cache coherence
- Manages existence of multiple copies
- Cache architectures
  - Multi level caches
  - Shared vs. private (partitioned)
  - Inclusive vs. exclusive
  - Write back vs. write through
  - Victim cache to reduce conflict misses
  - ...
Shared Hierarchical Caches with MT

Write Through Cache
1. CPU₀ reads X from memory
   • loads X=0 into its cache
2. CPU₁ reads X from memory
   • loads X=0 into its cache
3. CPU₀ writes X=1
   • stores X=1 in its cache
   • stores X=1 in memory
4. CPU₁ reads X from its cache
   • loads X=0 from its cache
   CPU₁ may wait for update!

Write Back Cache
1. CPU₀ reads X from memory
   • loads X=0 into its cache
2. CPU₁ reads X from memory
   • loads X=0 into its cache
3. CPU₀ writes X=1
   • stores X=1 in its cache
4. CPU₁ writes X=2
   • stores X=2 in its cache
5. CPU₁ writes back cache line
   • stores X=2 in memory
6. CPU₀ writes back cache line
   • stores X=2 in memory
   Later (!) store X=2 from CPU₁ lost

A simple (?) example
- Assume C99:
  ```
  struct twoint {
    int a;
    int b;
  }
  ```

  - Two threads:
    - Initially: a=0
    - Thread 0: write 1 to a
    - Thread 1: write 1 to b
  - Assume non-coherent write back cache
    - What may end up in main memory?

Caching Strategies (repeat)
- Remember:
  • Write Back?
  • Write Through?
- Cache coherence requirements
  A memory system is coherent if it guarantees the following:
  • Write propagation (updates are eventually visible to all readers)
  • Write serialization (writes to the same location must be observed in order)
  Everything else: memory model issues (later)

Cache Coherence Protocol
- Programmer can hardly deal with unpredictable behavior!
- Cache controller maintains data integrity
  • All writes to different locations are visible

  Fundamental Mechanisms
  • Snooping
    - Shared bus or (broadcast) network
  • Directory-based
    - Record information necessary to maintain coherence:
      E.g., owner and state of a line etc.
Fundamental CC mechanisms
- Snooping
  - Shared bus or (broadcast) network
  - Cache controller "snoops" all transactions
  - Monitors and changes the state of the cache's data
    - Works at small scale, challenging at large-scale
      - E.g., Intel Core (Broadwell, ...)
- Directory-based
  - Record information necessary to maintain coherence
    - E.g., owner and state of a line etc.
  - Central/Distributed directory for cache line ownership
  - Scalable but more complex/expensive
    - E.g., Intel Xeon Phi KNC/KNL

Cache Coherence Parameters
- Concerns/Goals
  - Performance
  - Implementation cost (chip space, more important: dynamic energy)
  - Correctness
    - (Memory model side effects)
- Issues
  - Detection (when does a controller need to act)
  - Enforcement (how does a controller guarantee coherence)
  - Precision of block sharing (per block, per sub-block?)
  - Block size (cache line size?)

An Engineering Approach: Empirical start
- Problem 1: stale reads
  - Cache 1 holds value that was already modified in cache 2
  - Solution:
    - Disallow this state
    - Invalidate all remote copies before allowing a write to complete
- Problem 2: lost update
  - Incorrect write back of modified line writes main memory in different order from the order of the write operations or overwrites neighboring data
  - Solution:
    - Disallow more than one modified copy

Invalidation vs. update – possible implementations
- Invalidation-based:
  - On each write of a shared line, it has to invalidate copies in remote caches
  - Simple implementation for bus-based systems:
    - Each cache snoops
    - Invalidates lines written by other CPUs
    - Signal sharing for cache lines in local cache to other caches
- Update-based:
  - All sharers continue to hit cache line after one core writes
  - Implicit assumption: shared lines are accessed often
  - Supports producer-consumer pattern well
  - Many (local) writes may waste bandwidth!
- Hybrid forms are possible!

Invalidation vs. update – effects
- Invalidation-based:
  - Only write misses hit the bus (works with write-back caches)
  - Subsequent writes to the same cache line are local
  - Good for multiple writes to the same line (in the same cache)
- Update-based:
  - All sharers continue to hit cache line after one core writes
  - Implicit assumption: shared lines are accessed often
  - Supports producer-consumer pattern well
  - Many (local) writes may waste bandwidth!
- Hybrid forms are possible!

MESI Cache Coherence
- Most common hardware implementation of discussed requirements
  - Aka. "Illinois protocol"
Each line has one of the following states (in a cache):
- Modified (M)
  - Local copy has been modified, no copies in other caches
  - Memory is stale
- Exclusive (E)
  - No copies in other caches
  - Memory is up to date
- Shared (S)
  - Unmodified copies may exist in other caches
  - Memory is up to date
- Invalid (I)
  - Line is not in cache
Terminology

- **Clean line:**
  - Content of cache line and main memory is identical (also: memory is up to date)
  - Can be evicted without write-back

- **Dirty line:**
  - Content of cache line and main memory differ (also: memory is stale)
  - Needs to be written back eventually
  - Time depends on protocol details

- **Bus transaction:**
  - A signal on the bus that can be observed by all caches
  - Usually blocking

- **Local read/write:**
  - A load/store operation originating at a core connected to the cache

Transitions in response to local reads

- **State is M**
  - No bus transaction
  - Go to state M

- **State is E**
  - No bus transaction
  - Generate bus read request (BusRd)
    - May force other cache operations (see later)
    - Other cache(s) signal "sharing" if they hold a copy
    - If shared was signaled, go to state S
    - Otherwise, go to state I

- **State is I**
  - Generate bus read request for exclusive ownership (BusRdX)
  - Go to state M

- **After update: return read value**

Transitions in response to local writes

- **State is M**
  - No bus transaction
  - Write cache line back to main memory
  - Discard line and go to I

- **State is E**
  - No bus transaction
  - Discard line and go to I

- **State is S**
  - Disk already local & clean
  - There may be other copies
  - Generate bus read request for upgrade to exclusive (BusRdX*)
    - Go to state M

- **State is I**
  - Generate bus read request for exclusive ownership (BusRdX)
  - Go to state M

Transitions in response to snooped BusRd

- **State is M**
  - Write cache line back to main memory
  - Signal "shared"
    - Go to state S [or E]
  - State is E
    - Signal "shared"
    - Go to state S and signal "shared"
  - State is S
    - Signal "shared"
  - State is I
    - Ignore

Transitions in response to snooped BusRdX

- **State is M**
  - Write cache line back to memory
  - Discard line and go to I

- **State is E**
  - Discard line and go to I

- **State is S**
  - Discard line and go to I

- **State is I**
  - Ignore

- **BusRdX* is handled like BusRdX!**

MESI State Diagram (FSM)
Small Exercise
- Initially: all in I state

<table>
<thead>
<tr>
<th>Action</th>
<th>P1 state</th>
<th>P2 state</th>
<th>P3 state</th>
<th>Bus action</th>
<th>Data from</th>
</tr>
</thead>
<tbody>
<tr>
<td>P1 reads x</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>P2 reads x</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>P1 writes x</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>P2 writes x</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Optimizations?
- Class question: what could be optimized in the MESI protocol to make a system faster?

Related Protocols: MOESI (AMD)
- Extended MESI protocol
  - Cache-to-cache transfer of modified cache lines
  - Cache in M or O state always transfers cache line to requesting cache
  - No need to contact (slow) main memory
- Avoids write back when another process accesses cache line
- Good when cache-to-cache performance is higher than cache-to-memory
  E.g., shared last level cache!

MOESI State Diagram

Related Protocols: MOESI (AMD)
- Modified (M): Modified Exclusive
  - No copies in other caches, local copy dirty
  - Memory is stale, cache supplies copy (reply to BusRd*)
- Owner (O): Modified Shared
  - Exclusive right to make changes
  - Other S copies may exist ("dirty sharing")
  - Memory is stale, cache supplies copy (reply to BusRd*)
- Exclusive (E):
  - Same as MESI (one local copy, up to date memory)
- Shared (S):
  - Unmodified copy may exist in other caches
  - Memory is up to date unless an O copy exists in another cache
- Invalid (I):
  - Same as MESI
Related Protocols: MESIF (Intel)
- Modified (M): Modified Exclusive
  - No copies in other caches, local copy dirty
  - Memory is stale, cache supplies copy (reply to BusRd*)
- Exclusive (E):
  - Same as MESI (one local copy, up to date memory)
- Shared (S):
  - Unmodified copy may exist in other caches
  - Memory is up to date
- Invalid (I):
  - Same as MESI
- Forward (F):
  - Special form of S state, other caches may have line in S
  - Most recent requester of line is in F state
  - Cache acts as responder for requests to this line

Multi-level caches
- Most systems have multi-level caches
- Problem: only “last level cache” is connected to bus or network
  - Yet, snoop requests are relevant for inner levels of cache (L1)
- Modifications of L1 data may not be visible at L2 (and thus the bus)
  - On BusRd check if line is in M state in L1
    - It may be in E or S in L2!
  - On BusRdX(*) send invalidations to L1
  - Everything else can be handled in L2
  - If L1 is write through, L2 could “remember” state of L1 cache line
    - May increase traffic though

Directory-based cache coherence
- Snooping does not scale
  - Bus transactions must be globally-visible
  - Implies broadcast
  - Typical solution: tree-based (hierarchical) snooping
  - Root becomes a bottleneck
  - Directory-based schemes are more scalable
  - Directory entry for each CL keeps track of all owning caches
  - Point-to-point update to involved processors
  - No broadcast
    - Can use specialized (high-bandwidth) network, e.g., HT, QPI ...

Basic Scheme
- System with N processors \( P_i \)
- For each memory block (size: cache line) maintain a directory entry
  - N presence bits (light blue)
  - Set if block is in cache of \( P_i \)
  - 1 dirty bit (red)
  - First proposed by Censier and Feautrier (1978)

Directory-based CC: Read miss
- \( P_0 \) intends to read, misses
  - If dirty bit (in directory) is off
    - Read from main memory
    - Set presence[i]
    - Supply data to reader

Directory-based CC: Read miss
- \( P_0 \) intends to read, misses
  - If dirty bit is on
    - Recall cache line from \( P_j \)
    - Update memory
    - Unset dirty bit, block shared
    - Set presence[i]
    - Supply data to reader
Directory-based CC: Write miss

- \( P_0 \) intends to write, misses
- If dirty bit (in directory) is off
  - Send invalidations to all processors \( P_j \)
  - Set dirty bit
  - Reset presence\[i\], owner \( P_i \)
- If dirty bit is on
  - Recall cache line from owner \( P_j \)
  - Update memory
  - Set presence\[j\]
  - Set dirty bit
  - Acknowledge to writer

Directory
\[
\begin{array}{cccc}
X & 0 & 0 & 1 \\[i\] \[j\] \\
\end{array}
\]

Write \( X = 0 \)

Discussion

- Scaling of memory bandwidth
  - No centralized memory
- Directory-based approaches scale with restrictions
  - Require presence bit for each cache
  - Number of bits determined at design time
  - Directory requires memory (size scales linearly)
  - Shared vs. distributed directory
- Software-emulation
  - Distributed shared memory (DSM)
  - Emulate cache coherence in software (e.g., TreadMarks)
  - Often on a per-page basis, utilizes memory virtualization and paging

Open Problems (for projects, theses, research)

- Tune algorithms to cache-coherence schemes
  - What is the optimal parallel algorithm for a given scheme?
  - Parameterize for an architecture
- Measure and classify hardware
  - Read Maranget et al. "A Tutorial Introduction to the ARM and POWER Relaxed Memory Models" and have fun!
  - RDMA consistency is barely understood!
  - GPU memories are not well understood!
  - Huge potential for new insights!
- Can we program (easily) without cache coherence?
  - How to fix the problems with inconsistent values?
  - Compiler support (issues with arrays)?

Case Study: Intel Xeon Phi

Communication?

Local read: \( R_L = 8.6 \text{ ns} \)

Remote read: \( R_R = 235 \text{ ns} \)

Invalid read: \( R_I = 278 \text{ ns} \)

Inspired by Molka et al.: "Memory performance and cache coherency effects on an Intel Nehalem multiprocessor system"

---

Prediction for both in E state: 479 ns

Measurement: 497 ns (O=18)

Multi-Line Ping Pong

- More complex due to prefetch

\[ T_N = a \cdot N + q - \frac{p}{N} \]

- E state:
  - \( a=76 \text{ ns} \)
  - \( q=1,521 \text{ ns} \)
  - \( p=1,096 \text{ ns} \)

- I state:
  - \( a=95 \text{ ns} \)
  - \( q=2,750 \text{ ns} \)
  - \( p=2,017 \text{ ns} \)

DTD Contention

- E state:
  - \( a=0 \text{ ns} \)
  - \( b=320 \text{ ns} \)
  - \( c=56.2 \text{ ns} \)

---

Optimizations against vendor libraries

- Barrier (7x faster than OpenMP)
- Reduce (5x faster than OpenMP)

---

Image credits

- Slide 23, RAM: © Raimond Spekking / CC BY-SA 4.0 (via Wikimedia Commons) https://commons.wikimedia.org/wiki/File:Apacer_SDRAM-3386.jpg