Computer Architecture
Lecture 20: Cache Coherence

Dr. Mohammad Sadrosadati
Prof. Onur Mutlu
ETH Zürich
Fall 2023
1 December 2023
Recall:
Weak Memory Ordering
Recall: Issues with Sequential Consistency?

- Performance enhancement techniques that could make SC implementation difficult

- Out-of-order execution
  - Loads happen out-of-order with respect to each other and with respect to independent stores → makes it difficult for all processors to see the same global order of all memory operations

- Caching
  - A memory location is now present in multiple places
  - Prevents the effect of a store to be seen by other processors → makes it difficult for all processors to see the same global order of all memory operations
Recall: Weaker Memory Consistency

- The ordering of operations is important when the order affects operations on shared data → i.e., when processors need to synchronize to execute a “program region”

- Weak consistency
  - Idea: Programmer specifies regions in which memory operations do not need to be ordered
  - “Memory fence” instructions delineate those regions
    - All memory operations before a fence must complete before fence is executed
    - All memory operations after the fence must wait for the fence to complete
    - Fences complete in program order
  - All synchronization operations act like a fence
Examples of Weak Consistency Models


A more relaxed consistency model can be derived by relating memory request ordering to synchronization points in the program. The weak consistency model (WC) proposed by Dubois et al. [4, 5] is based on the above idea and guarantees a consistent view of memory only at synchronization points. As an example, consider a process updating a data structure within a critical section. Under SC, every access within the critical section is delayed until the previous access completes. But such delays are unnecessary if the programmer has already made sure that no other process can rely on the data structure to be consistent until the critical section is exited. Weak consistency exploits this by allowing accesses within the critical section to be pipelined. Correctness is achieved by guaranteeing that all previous accesses are performed before entering or exiting each critical section.

Figure 1: Ordering restrictions on memory accesses.
More on Weak Consistency


Examples of Weak Consistency Models


Release consistency (RC) [8] is an extension of weak consistency that exploits further information about synchronization by classifying them into acquire and release accesses. An acquire synchronization access (e.g., a lock operation or a process spinning for a flag to be set) is performed to gain access to a set of shared locations. A release synchronization access (e.g., an unlock operation or a process setting a flag) grants this permission. An acquire is accomplished by reading a shared location until an appropriate value is read. Thus, an acquire is always associated with a read synchronization access (see [8] for discussion of read-modify-write accesses). Similarly, a release is always associated with a write synchronization access. In contrast to WC, RC does not require accesses following a release to be delayed for the release to complete; the purpose of the release is to signal that previous accesses are complete, and it does not have anything to say about the ordering of the accesses following it. Similarly, RC does not require an acquire to be delayed for its previous accesses. The data-race-free-0 (DRF0) [2] model

Figure 1: Ordering restrictions on memory accesses.
More on Release Consistency

Tradeoffs: Weaker Consistency

■ Advantage
  - No need to guarantee a (very) strict order of memory operations
    → Enables the hardware implementation of performance enhancement techniques to be simpler
    → Can be higher performance than stricter ordering

■ Disadvantage
  - More burden on the programmer or software (need to get the “fences” and labeling of synchronization operations correct)
  - Debugging is harder → harder to reason about what went wrong

■ Another example of the programmer-microarchitect tradeoff
More on Weak Consistency Models


Abstract

The memory consistency model supported by a multiprocessor directly affects its performance. Thus, several attempts have been made to relax the consistency models to allow for more buffering and pipelining of memory accesses. Unfortunately, the potential increase in performance afforded by relaxing the consistency model is accompanied by a more complex programming model. This paper introduces two general implementation techniques that provide higher performance for all the models. The first technique involves prefetching values for accesses that are delayed due to consistency model constraints. The second technique employs speculative execution to allow the processor to proceed even though the consistency model requires the memory accesses to be delayed. When combined, the above techniques alleviate the limitations imposed by a consistency model on buffering and pipelining of memory accesses, thus significantly reducing the impact of the memory consistency model on performance.
Cache Coherence
Caching in Multiprocessors

- Caching not only complicates ordering of all operations...
  - A memory location can be present in multiple caches
  - Prevents the effect of a store or load to be seen by other processors → makes it difficult for all processors to see the same global order of (all) memory operations

- ... but it also complicates ordering of operations on a single memory location
  - A single memory location can be present in multiple caches
  - Makes it difficult for processors that have cached the same location to have the correct value of that location (in the presence of updates to that location)
Memory Consistency vs. Cache Coherence

- **Consistency** is about ordering of all memory operations from different processors (i.e., to different memory locations)
  - **Global ordering** of accesses to all memory locations

- **Coherence** is about ordering of operations from different processors to the same memory location
  - **Local ordering** of accesses to each cache block
Readings: Cache Coherence

Required
- Culler and Singh, *Parallel Computer Architecture*
  - Chapter 5.1 (pp 269 – 283), Chapter 5.3 (pp 291 – 305)
- P&H, *Computer Organization and Design*
  - Chapter 5.8 (pp 534 – 538 in 4th and 4th revised eds.)

Recommended
Shared Memory Model

- Many parallel programs communicate through *shared memory*
- Proc 0 writes to an address, followed by Proc 1 reading
  - This implies communication between the two

  **Proc 0**
  Mem[A] = 1

  **Proc 1**
  ...

  Print Mem[A]

- Each read should receive the value last written by anyone
  - This requires synchronization (what does last written mean?)
- What if Mem[A] is cached (at either end)?
Basic question: If multiple processors cache the same block, how do they ensure they all see a consistent state?
The Cache Coherence Problem

```
ld r2, x
```

Interconnection Network

```
x 1000
```

Main Memory
The Cache Coherence Problem

P1
1000

P2
1000

Interconnection Network

Main Memory

ld r2, x

ld r2, x
The Cache Coherence Problem

ld r2, x
add r1, r2, r4
st x, r1

ld r2, x

P1

2000

P2

1000

Interconnection Network

Main Memory

x 1000

ld r2, x
The Cache Coherence Problem

P1

ld r2, x
add r1, r2, r4
st x, r1

P2

ld r2, x

Interconnection Network

Main Memory

ld r5, x

ld r2, x

1000

Should NOT load 1000

1000

2000
Cache Coherence: Whose Responsibility?

- **Software**
  - Can programmer ensure coherence if caches invisible to software?
  - **Coarse-grained:** Page-level coherence has overheads
  - Non-solution: Make shared locks/data non-cacheable
  - A combination of non-cacheable and coarse-grained is doable
  - **Fine-grained:** What if the ISA provided a cache flush instruction?
    - FLUSH-LOCAL A: Flushes/invalidates the cache block containing address A from a processor’s local cache.
    - FLUSH-GLOBAL A: Flushes/invalidates the cache block containing address A from all other processors’ caches.
    - FLUSH-CACHE X: Flushes/invalidates all blocks in cache X.

- **Hardware**
  - Greatly simplifies software’s job
  - One idea: Invalidate all other copies of block A when a core writes to A
A Very Simple Coherence Scheme (VI)

- Caches “snoop” (observe) each other’s write/read operations. If a processor writes to a block, all others invalidate the block.
- A simple protocol:

Write-through, no-write-allocate cache

Actions of the local processor on the cache block: PrRd, PrWr,

Actions that are broadcast on the bus for the block: BusRd, BusWr
(Non-)Solutions to Cache Coherence

- No hardware based coherence
  - Keeping caches coherent is software’s responsibility
    + Makes microarchitect’s life easier
  - Makes average programmer’s life much harder
    - need to worry about hardware caches to maintain program correctness?
  - Overhead in ensuring coherence in software (e.g., page protection, page-based software coherence, non-cacheable)

- All caches are shared between all processors
  + No need for coherence
  - Shared cache becomes the bottleneck
  - Very hard to design a scalable system with low-latency cache access this way
Maintaining Coherence

- Need to guarantee that all processors see a consistent value (i.e., consistent updates) for the same memory location

- Writes to location A by P0 should be seen by P1 (eventually), and all writes to A should appear in some order

- Coherence needs to provide:
  - **Write propagation**: guarantee that updates will propagate
  - **Write serialization**: provide a consistent order seen by all processors for the same memory location

- Need a global point of serialization for this write ordering
Hardware Cache Coherence

- Basic idea:
  - A processor/cache broadcasts its write/update to a memory location to all other processors
  - Another cache that has the location either updates or invalidates its local copy
Coherence: Update vs. Invalidate

- How can we *safely update replicated data*?
  - Option 1 (Update protocol): push an update to all copies
  - Option 2 (Invalidate protocol): ensure there is only one copy (local), update it

- **On a Read:**
  - If local copy is Invalid, put out request
  - (If another node has a copy, it returns it, otherwise memory does)
Coherence: Update vs. Invalidate (II)

- **On a Write:**
  - Read block into cache as before

**Update Protocol:**
- Write to block, and simultaneously broadcast written data and address to sharers
- (Other nodes update the data in their caches if block is present)

**Invalidate Protocol:**
- Write to block, and simultaneously broadcast invalidation of address to sharers
- (Other nodes invalidate block in their caches if block is present)
Update vs. Invalidate Tradeoffs

- Which one is better? Update or invalidate?
  - Write frequency and sharing behavior are critical
- **Update**
  - If sharer set is constant and updates are infrequent, avoids the cost of invalidate-reacquire (broadcast update pattern)
  - If data is rewritten without intervening reads by other cores, updates would be useless
  - Write-through cache policy → bus can become a bottleneck
- **Invalidate**
  - After invalidation, core has exclusive access rights
  - Only cores that keep reading after each write retain a copy
  - If write contention is high, leads to ping-ponging (rapid invalidation-reacquire traffic from different processors)
Two Cache Coherence Methods

- How do we ensure that the proper caches are updated?

- **Snoopy Bus** [Goodman ISCA 1983, Papamarcos+ ISCA 1984]
  - Bus-based, single point of serialization *for all memory requests*
  - Processors observe other processors’ actions
    - E.g.: P1 makes “read-exclusive” request for A on bus, P0 sees this and invalidates its own copy of A

- **Directory** [Censier and Feautrier, IEEE ToC 1978]
  - Single point of serialization *per block*, distributed among nodes
  - Processors make explicit requests for blocks
  - Directory tracks which caches have each block
  - Directory coordinates invalidations and updates
    - E.g.: P1 asks directory for exclusive copy, directory asks P0 to invalidate, waits for ACK, then responds to P1
Directory Based
Cache Coherence
Directory Based Coherence

- **Idea:** A logically-central directory keeps track of where the copies of each cache block reside. Caches consult this directory to ensure coherence.

- **An example mechanism:**
  - For each cache block in memory, store P+1 bits in directory
    - One bit for each cache, indicating whether the block is in cache
    - Exclusive bit: indicates that a cache has the only copy of the block and can update it without notifying others
  - On a read: set the cache’s bit and arrange the supply of data
  - On a write: invalidate all caches that have the block and reset their bits
  - Have an “exclusive bit” associated with each block in each cache (so that the cache can update the exclusive block silently)
Directory Based Coherence Example (I)

Example directory based scheme

\[ P+1 \]
\[ \text{bits:} \]
\[ \text{for block } A \]
\[ P_1 \text{ takes a read miss to block A} \]
\[ 000000 \]
\[ \rightarrow \]
\[ 010000 \]

No cache has the block

\[ P_3 \text{ takes a read miss} \]
\[ 01010 \]
3. P2 takes a write miss
   → Invalidate P1 & P3's caches
   → Write request → P2 has the exclusive copy of the block
   now. Set the Exclusive bit
   → P2 can now update the block without notifying any other processor or the directory
   → P2 needs to have a bit in its cache indicating it can perform exclusive updates to that block
   → private/exclusive bit per cache block

4. P3 takes a write miss
   → Mem Controller requests block from P2
   → Mem Controller gives block to P3
   → P2 invalidates its copy

5. P2 takes a read miss
   → P3 supplies it
Directory Optimizations

- Directory is the coordinator for all actions to be performed on a given block by any processor
  - Guarantees correctness, ordering

- Yet, there are many opportunities for optimization
  - Enabled by bypassing the directory and directly communicating between caches
  - We will see examples of these optimizations later
Snoopy Cache Coherence
Snoopy Cache Coherence

- **Idea:**
  - All caches “snoop” all other caches’ read/write requests and keep the cache block coherent.
  - Each cache block has “coherence metadata” associated with it in the tag store of each cache.

- **Easy to implement if all caches share a common bus**
  - Each cache broadcasts its read/write operations on the bus.
  - Good for small-scale multiprocessors.
  - What if you would like to have a 10,000-node multiprocessor?
SNOOPY CACHE

Each Cache observes its own processor & the bus
- Changes the state of the cached block based on observed actions by processors & the bus

Processor action to a block:  
- PR (Proc. Read)  
- RW (Proc. Write)

Bus action to a block:  
- BR (Bus Read)  
- BW (Bus Write)  
- or BRxe (Bus Read Exclusive)
A Simple Snoopy Cache Coherence Protocol

- Caches “snoop” (observe) each others’ write/read operations
- A simple protocol (VI protocol):

- **Write-through**, no-write-allocate cache
- Actions of the local processor on the cache block: PrRd, PrWr,
- Actions that are broadcast on the bus for the block: BusRd, BusWr
Extending the Protocol

- What if you want write-back caches?
  - We want a “modified” state
A More Sophisticated Protocol: MSI

- Extend metadata per block to encode three states:
  - **M** (modified): cache line is the only cached copy and is dirty
  - **S** (shared): cache line is one of potentially several cached copies and it is clean (i.e., at least one clean cached copy)
  - **I** (invalid): cache line is not present in this cache

- Read miss makes a *Read* request on bus, transitions to **S**
- Write miss makes a *ReadEx* request, transitions to **M** state
- When a processor snoops *ReadEx* from another writer, it must invalidate its own copy (if any)
- **S** → **M** upgrade can be made without accessing memory (via *Invalidations*)
MSI State Machine

[Culler/Singh96]
The Problem with MSI

- A block is in no cache to begin with

- Problem: On a read, the block immediately goes to “Shared” state although it may be the only copy to be cached (i.e., no other processor will cache it)

- Why is this a problem?
  - Suppose the cache that reads the block wants to write to it at some point
  - It needs to broadcast “invalidate” even though it has the only cached copy!
  - *If the cache knew it had the only cached copy in the system, it could have written to the block without notifying any other cache* → saves unnecessary broadcasts of invalidations
The Problem with MSI

[Culler/Singh96]
The Solution: MERSI

- Idea: Add another state indicating that this is the only cached copy and it is clean.
  - *Exclusive* state

- Block is placed into the *exclusive* state if, during *BusRd*, no other cache had it
  - Wired-OR “shared” signal on bus can determine this: snooping caches assert the signal if they also have a copy

- Silent transition *Exclusive* $\rightarrow$ *Modified* is possible on write!

- MERSI is also called the *Illinois protocol*
Illinois Protocol

4 States
M: Modified (Exclusive copy, modified)
E: Exclusive ("", "", clean)
S: Shared (Shared copy, clean)
I: Invalid

BI: Invalidate, but already have the data (do not supply it)
BRI: Invalidate, but also need the data (supply it)
MESI State Machine

[Diagram of MESI State Machine with states and transitions]

- **INV**
- **Exc. Mod.**
- **Exc. Clean**
- **Shared Clean**

Transitions and States:
- **PR/BR**
- **PR/BR**
- **PW/BR**
- **PW/BR**
- **PR/PR, PW**
- **BR/BR, supply data**
- **BRI/BR, BRI, BR**
- **BRI/supply data**
MESI State Machine

[Culler/Singh96]
A transition from a single-owner state (Exclusive or Modified) to Shared is called a **downgrade**, because the transition takes away the owner's right to modify the data.

A transition from Shared to a single-owner state (Exclusive or Modified) is called an **upgrade**, because the transition grants the ability to the owner (the cache which contains the respective block) to write to the block.
MESI State Machine from Bonus Lab 6
Intel Pentium Pro Coherence Protocol

- Write Allocate
- L1 can have data not in L2
- Hit: Someone has it clean
  HitM: Someone has it dirty
Snoopy Invalidation Tradeoffs

- Should a downgrade from M go to S or I?
  - S: if data is likely to be reused (before it is written to by another processor)
  - I: if data is likely to be not reused (before it is written to by another)

- Cache-to-cache transfers?
  - On a BusRd, should data come from another cache or memory?
    - Another cache
      - May be faster, if memory is slow or highly contended
    - Memory
      - Simpler: no need to wait to see if another cache has the data first
      - Less contention at the other caches
      - Requires writeback on M downgrade

- Writeback on Modified->Shared needed since Shared state is Clean
  - Can we avoid this writeback to memory?
  - One possibility: **Owner** (O) state (MOESI protocol)
    - One cache owns the latest data (memory is not updated)
    - Memory writeback happens when all caches evict copies
The Problem with MESI

- **Observation:** Shared state requires the data to be clean
  - i.e., all caches that have the block have the up-to-date copy and so does the memory

- **Problem:** Need to write the block to memory when BusRd happens when the block is in Modified state

- **Why is this a problem?**
  - Memory can be updated unnecessarily → some other processor may want to write to the block again
Improving on MESI

- Idea 1: Do not transition from M→S on a BusRd. Invalidate the copy and supply the modified block to the requesting processor directly without updating memory.

- Idea 2: Transition from M→S, but designate one cache as the owner (O), who will write the block back when it is evicted.
  - Now “Shared” means “Shared and potentially dirty”
  - This is a version of the MOESI protocol.
The protocol can be optimized with more states and prediction mechanisms to
+ Reduce unnecessary invalidates and transfers of blocks

However, more states and optimizations
-- Are more difficult to design and verify (lead to more cases to take care of, race conditions)
-- Provide diminishing returns
Revisiting Two Cache Coherence Methods

- How do we ensure that the proper caches are updated?

- **Snoopy Bus** [Goodman ISCA 1983, Papamarcos+ ISCA 1984]
  - Bus-based, *single point of serialization for all memory requests*
  - Processors observe other processors’ actions
    - E.g.: P1 makes “read-exclusive” request for A on bus, P0 sees this and invalidates its own copy of A

- **Directory** [Censier and Feautrier, IEEE ToC 1978]
  - *Single point of serialization per block*, distributed among nodes
  - Processors make explicit requests for blocks
  - Directory tracks which caches have each block
  - Directory coordinates invalidation and updates
    - E.g.: P1 asks directory for exclusive copy, directory asks P0 to invalidate, waits for ACK, then responds to P1
Snoopy Cache vs. Directory Coherence

**Snoopy Cache**

+ Miss latency (critical path) is short: request → bus transaction to mem.
+ Global serialization is easy: bus provides this already (arbitration)
+ Simple: can adapt bus-based uniprocessors easily
  - Relies on broadcast messages to be seen by all caches (in same order):
    → single point of serialization (bus): *not scalable*
    → *need a virtual bus (or a totally-ordered interconnect)*

**Directory**

- Adds indirection to miss latency (critical path): request → dir. → mem.
- Requires extra storage space to track sharer sets
  - Can be approximate (false positives are OK for correctness)
- Protocols and race conditions are more complex (for high-performance)
+ Does not require broadcast to all caches
+ Exactly as scalable as interconnect and directory storage
  *(much more scalable than bus)*
Revisiting Directory-Based Cache Coherence
Remember: Directory Based Coherence

- **Idea:** A logically-central directory keeps track of where the copies of each cache block reside. Caches consult this directory to ensure coherence.

- **An example mechanism:**
  - For each cache block in memory, store $P+1$ bits in directory
    - One bit for each cache, indicating whether the block is in cache
    - Exclusive bit: indicates that the cache that has the only copy of the block and can update it without notifying others
  - On a read: set the cache’s bit and arrange the supply of data
  - On a write: invalidate all caches that have the block and reset their bits
  - Have an “exclusive bit” associated with each block in each cache
Remember: Directory Based Coherence

Example directory based scheme

P=4

No cache has the block

P_1 takes a read miss to block A

P_3 takes a read miss
Directory-Based Protocols

- Required when scaling past the capacity of a single bus
- Distributed:
  - Coherence still requires single point of serialization for a given block (for write serialization)
  - Serialization location can be different for different blocks (striped across nodes/memory-controllers)

- We can reason about the protocol for a single block: one server (directory node), many clients (private caches)

- Directory receives *Read* and *ReadEx* requests, and sends *Invl* requests: invalidation is explicit (as opposed to snoopy bus)
Directory: Data Structures

- Required to support invalidation and cache block requests
- Key operation to support is *set inclusion test*
  - False positives are OK: want to know which caches *may* contain a copy of a block, and spurious invalidations are ignored
  - False positive rate determines *performance*
- Most accurate (and expensive): full bit-vector
- Compressed representation, linked list, Bloom filters are all possible

<table>
<thead>
<tr>
<th></th>
<th>Shared: {P0, P1, P2}</th>
<th>Exclusive: P2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x00</td>
<td>---</td>
<td></td>
</tr>
<tr>
<td>0x04</td>
<td>---</td>
<td></td>
</tr>
<tr>
<td>0x08</td>
<td>---</td>
<td></td>
</tr>
<tr>
<td>0x0C</td>
<td>---</td>
<td></td>
</tr>
<tr>
<td>...</td>
<td>---</td>
<td></td>
</tr>
</tbody>
</table>
Directory: Basic Operations

- Follow *semantics* of a snoop-based system
  - but with explicit request, reply messages

- Directory:
  - Receives *Read, ReadEx, Upgrade* requests from nodes
  - Sends *Inval/Downgrade* messages to sharers if needed
  - Forwards request to memory if needed
  - Replies to requestor and updates sharing state

- Protocol design is flexible
  - Exact forwarding paths depend on implementation
  - For example, do cache-to-cache transfer?
P0 acquires an address for reading:

1. Read

2. DatEx (DatShr)

Culler/Singh Fig. 8.16
RdEx with Former Owner

1. RdEx

P0

Home

2. Invl

Owner

3a. Rev

3b. DatEx
Contestion Resolution (for Write)

P0

1a. RdEx
2a. DatEx
4. Invl
5a. Rev

Home

3. RdEx
2b. NACK
5b. DatEx

P1

1b. RdEx
Issues with Contention Resolution

- Need to escape race conditions by:
  - NACKing requests to busy (pending invalidate) entries
    - Original requestor retries
  - OR, queuing requests and granting in sequence
  - (Or some combination thereof)

- Fairness
  - Which requestor should be preferred in a conflict?
  - Interconnect delivery order, and distance, both matter

- Ping-ponging can be reduced w/ protocol optimizations OR better higher-level synchronization
  - With solutions like combining trees (for locks/barriers) and better shared-data-structure design
Scaling the Directory: Some Questions

- How large is the directory?

- How can we reduce the access latency to the directory?

- How can we scale the system to thousands of nodes?

- Can we get the best of snooping and directory protocols?
  - Heterogeneity
  - E.g., token coherence [Martin+, ISCA 2003]
(f) Directory [11 points]

Assume we have a processor that implements the directory based cache coherence protocol we discussed in class. The physical address space of the processor is 32GB \((2^{35}\text{ bytes})\) and a cache block is 128 bytes. The directory is equally distributed across randomly selected 32 nodes in the system.

You find out that the directory size in each of the 32 nodes is a total of 200 MB.

How many total processors are there in this system? Show your work.
An Example Answer

- **Blocks per node**
  - \( (32\text{GB address space} / 128 \text{ bytes per block}) / 32 \text{ nodes} \)
  - \( 2^{(35-7-5)} = 2^{23} \)

- **Directory storage per node**
  - **200 MB** = \( 25 \times 2^{23} \text{ bytes} = 25 \times 2^{26} \text{ bits} \)

- **Directory storage per block**
  - \( 25 \times 2^{26} \text{ bits} / 2^{23} \text{ blocks} = 200 \text{ bits per block} \)

- Each directory entry has \( P+1 \) bits
  - \( P+1 = 200 \implies P = 199 \)
Cache Coherence & RowHammer
Prior work shows that Rowhammer attacks—which flip bits in DRAM via frequent activations of the same row(s)—are viable. Adversaries typically mount these attacks via instruction sequences that are carefully-crafted to bypass CPU caches. However, we discover a novel form of hammering that we refer to as coherence-induced hammering, caused by Intel’s implementations of cache coherent non-uniform memory access (ccNUMA) protocols. We show that this hammering occurs in commodity benchmarks on a major cloud provider’s production hardware, the first hammering found to be generated by non-malicious code. Given DRAM’s rising susceptibility to bit flips, it is paramount to prevent coherence-induced hammering to ensure reliability and security in the cloud.

Accordingly, we introduce MOESI-prime, a ccNUMA coherence protocol that mitigates coherence-induced hammering while retaining Intel’s state-of-the-art scalability. MOESI-prime shows that most DRAM reads and writes triggering such hammering are unnecessary. Thus, by encoding additional information in the coherence protocol, MOESI-prime can omit these reads and writes, preventing coherence-induced hammering in non-malicious and malicious workloads. Furthermore, by omitting unnecessary reads and writes, MOESI-prime has negligible effect on average performance (within ±0.61% of MESI and MOESI) and average DRAM power (0.03%–0.22% improvement) across evaluated ccNUMA configurations.
**MOESI-prime: Preventing Coherence-Induced Hammering in Commodity Workloads**

Kevin Loughlin  
University of Michigan

Stefan Saroiu  
Microsoft

Alec Wolman  
Microsoft

Yatin A. Manerkar  
University of Michigan

Baris Kasikci  
University of Michigan

---

![Image](https://www.kevinloughlin.org/moesi-prime.pdf)

**Figure 3:** Activation (ACT) rates on a major cloud provider’s production hardware for (a) commodity benchmarks and (b) worst-case micro-benchmarks. In both cases, dirty sharing across NUMA nodes yields ACTs in excess of current Rowhammer thresholds (MACs).

[https://www.kevinloughlin.org/moesi-prime.pdf](https://www.kevinloughlin.org/moesi-prime.pdf)
Cache Coherence: A Recent Example
Eliminating the Adoption Barriers to PIM

How to Enable Adoption of Processing in Memory
Potential Barriers to Adoption of PIM

1. Applications & software for PIM

2. Ease of programming (interfaces and compiler/HW support)

3. System and security support: coherence, synchronization, virtual memory, isolation, communication interfaces, ...

4. Runtime and compilation systems for adaptive scheduling, data mapping, access/sharing control, ...

5. Infrastructures to assess benefits and feasibility

All can be solved with change of mindset
Amirali Boroumand, Saugata Ghose, Minesh Patel, Hasan Hassan, Brandon Lucia, Kevin Hsieh, Krishna T. Malladi, Hongzhong Zheng, and Onur Mutlu,
"LazyPIM: An Efficient Cache Coherence Mechanism for Processing-in-Memory"
Automatic Data Coherence Support for PIM

- Amirali Boroumand, Saugata Ghose, Minesh Patel, Hasan Hassan, Brandon Lucia, Rachata Ausavarungrunrun, Kevin Hsieh, Nastaran Hajinazar, Krishna T. Malladi, Hongzhong Zheng, and Onur Mutlu,

"CoNDA: Efficient Cache Coherence Support for Near-Data Accelerators"
[Slides (pptx) (pdf)]
[Lightning Talk Slides (pptx) (pdf)]
[Poster (pptx) (pdf)]
[Lightning Talk Video (4 minutes)]

CoNDA: Efficient Cache Coherence Support for Near-Data Accelerators

Amirali Boroumand†
Brandon Lucia†
Saugata Ghose†
Rachata Ausavarungrunrun†‡
Minesh Patel*
Hasan Hassan*
Nastaran Hajinazar◊†
Krishna T. Malladi§
Hongzhong Zheng§
Kevin Hsieh†
Onur Mutlu*†

†Carnegie Mellon University
◊Simon Fraser University
*ETH Zürich
§Samsung Semiconductor, Inc.
‡KMU
CoNDA: Efficient Cache Coherence Support for Near-Data Accelerators

Amirali Boroumand

Saugata Ghose, Minesh Patel, Hasan Hassan, Brandon Lucia, Rachata Ausavarungnirun, Kevin Hsieh, Nastaran Hajinazar, Krishna Malladi, Hongzhong Zheng, Onur Mutlu
Specialized Accelerators

Specialized accelerators are now everywhere!

- GPU
- FPGA
- ASIC

Recent advances in DRAM technology enabled Near-Data Accelerators (NDA)
Coherence For NDAs

Challenge: Coherence between NDAs and CPUs

(1) Large cost of off-chip communication

(2) NDA applications generate a large amount of off-chip data movement

It is impractical to use traditional coherence protocols
**Existing Coherence Mechanisms**

We extensively study existing **NDA coherence mechanisms** and make **three key observations**:

1. These mechanisms **eliminate** a significant portion of **NDA’s benefits**

2. The **majority of off-chip coherence traffic generated** by these mechanisms is **unnecessary**

3. Much of the **off-chip traffic** can be **eliminated** if the **coherence mechanism** has **insight** into the **memory accesses**
An Optimistic Approach

We find that an optimistic approach to coherence can address the challenges related to NDA coherence.

1. Gain insights before any coherence checks happen.
2. Perform only the necessary coherence requests.

We propose CoNDA, a coherence mechanism that lets an NDA optimistically execute an NDA kernel.

Optimistic execution enables CoNDA to identify and avoid unnecessary coherence requests.

CoNDA comes within 10.4% and 4.4% of performance and energy of an ideal NDA coherence mechanism.
Outline

• Introduction
• **Background**
• Motivation
• CoNDA
• Architecture Support
• Evaluation
• Conclusion
Background

- **Near-Data Processing (NDP)**
  - A potential solution to *reduce data movement*
  - **Idea:** move computation close to data

- ✓ Reduces data movement
- ✓ Exploits large in-memory bandwidth
- ✓ Exploits shorter access latency to memory

- **Enabled by recent advances in 3D-stacked memory**
Outline

• Introduction
• Background
• **Motivation**
• CoNDA
• Architecture Support
• Evaluation
• Conclusion
Application Analysis
Sharing Data between NDAs and CPUs

Hybrid Databases (HTAP)  Graph Processing

We find *not all portions* of applications benefit from NDA

1. Memory-intensive portions benefit from NDA
2. Compute-intensive or cache friendly portions should remain on the CPU

1st key observation: CPU threads often concurrently access the same region of data that NDA kernels access which leads to significant data sharing
Shared Data Access Patterns

2nd key observation: CPU threads and NDA kernels typically do not concurrently access the same cache lines.

For Connected Components application, only 5.1% of the CPU accesses collide with NDA accesses.

CPU threads rarely update the same data that an NDA is actively working on.
Analysis of NDA Coherence Mechanisms
We analyze three existing coherence mechanisms:

1. **Non-cacheable (NC)**
   - Mark the NDA data as non-cacheable

2. **Coarse-Grained Coherence (CG)**
   - Get coherence permission for the entire NDA region

3. **Fine-Grained Coherence (FG)**
   - Traditional coherence protocols
Non-Cacheable (NC) Approach

Mark the **NDA data as non-cacheable**

(1) Generates a **large number** of off-chip accesses

(2) Significantly **hurts** CPU threads performance

**NC fails to provide any energy saving and perform 6.0% worse than CPU-only**
Coarse-Grained (CG) Coherence

Get coherence permission for the entire NDA region

Unnecessarily flushes a large amount of dirty data, especially in pointer-chasing applications

Use coarse-grained locks to provide exclusive access

Blocks CPU threads when they access NDA data regions

CG fails to provide any performance benefit of NDA and performs 0.4% worse than CPU-only
Fine-Grained (FG) Coherence

Using fine-grained coherence has two benefits:

1. Simplifies NDA programming model
2. Allows us to get permissions for only the pieces of data that are actually accessed

FG eliminates 71.8% of the energy benefits of an ideal NDA mechanism
Analysis of Existing Coherence Mechanisms

Poor handling of coherence eliminates much of an NDA’s performance and energy benefits.
Motivation and Goal

1. Poor handling of coherence eliminates much of an NDA’s benefits

2. The majority of off-chip coherence traffic is unnecessary

Our goal is to design a coherence mechanism that:

1. Retains benefits of Ideal NDA
2. Enforces coherence with only the necessary data movement
Outline

• Introduction
• Background
• Motivation
• CoNDA
• Architecture Support
• Evaluation
• Conclusion
Optimistic NDA Execution

We leverage two key observations:

1. Having insight enables us to eliminate much of unnecessary coherence traffic
2. Low rate of collision for CPU threads and NDA kernels

We propose to use optimistic execution for NDAs

NDA executes the kernel:

1. Assumes it has coherence permissions
2. Gains insights into memory accesses

When execution is done:

Performs only the necessary coherence requests
Starts optimistic execution

CPU Thread Execution

Concurrent CPU + NDA Execution

No Coherence Request

Coherence Resolution

Commit or Re-execute

NDA

Optimistic Execution
We propose **CoNDA**, a mechanism that uses optimistic NDA execution to avoid unnecessary coherence traffic.
How do we identify coherence violations?
Necessary Coherence Requests

• Coherence requests are only necessary if:
  – Both NDA and CPU access a cache line
  – At least one of them updates it

We discuss three possible interleaving of accesses to the same cache line:

1. NDA Read and CPU Write (coherence violation)
2. NDA Write and CPU Read (no violation)
3. NDA Write and CPU Write (no violation)
Identifying Coherence Violations

1) NDA Read and CPU Write: **violation**

2) NDA Write and CPU Read: **no violation**

3) NDA Write and CPU Write: **no violation**

任何coherence checks during NDA execution

"C4" and "C5" are ordered before "N5"
Outline

• Introduction
• Background
• Motivation
• CoNDA
• Architecture Support
• Evaluation
• Conclusion
CoNDA: Architecture Support

- CPU
- DRAM
- Shared LLC
- Coherence Resolution
- CPUWriteSet
- NDAReadSet
- NDAWriteSet
The CPU records all writes to the NDA data region in the **CPUWriteSet**

Per-word dirty bit mask to mark all *uncommitted* data updates

The **NDAReadSet** and **NDAWriteSet** are used to track memory accesses from NDA
Bloom filter based signature has two major benefits:

- Allows us to easily perform coherence resolution
- Allows for a large number of addresses to be stored within a fixed-length register
If **conflicts** happens:

If **no conflicts**:

- Any clean cache lines in the CPU that **match** an address in the **NDAWriteSet** are **invalidated**
- NDA **commits** data updates
Outline

- Introduction
- Background
- Motivation
- CoNDA
- Architecture Support

- Evaluation
- Conclusion
Evaluation Methodology

- **Simulator**
  - Gem5 full system simulator

- **System Configuration:**
  - **CPU**
    - 16 cores, 8-wide, 2GHz frequency
    - L1 I/D cache: 64 kB private, 4-way associative, 64 B block
    - L2 cache: 2 MB shared, 8-way associative, 64 B blocks
    - Cache Coherence Protocol: MESI
  - **NDA**
    - 16 cores, 1-wide, 2GHz frequency
    - L1 I/D cache: 64 kB private, 4-way associative, 64 B Block
    - Cache coherence protocol: MESI
  - **3D-stacked Memory**
    - One 4GB Cube, 16 Vaults per cube
Applications

• **Ligra**
  – Lightweight multithreaded graph processing
  – We used three *Ligra* graph applications
    • *PageRank* (PR)
    • *Radii*
    • *Connected Components* (CC)
  – Real-world Input graphs:
    • Enron
    • arXiv
    • Gnutella25

• **Hybrid Database (HTAP)**
  – In-house prototype of an in-memory database
  – Capable of running both *transactional* and *analytical* queries on the *same* database (*HTAP workload*)
  – 32K transactions, 128/256 analytical queries
CoNDA consistently retains most of Ideal-NDA’s benefits, coming within 10.4% of the Ideal-NDA performance.
CoNDA significantly reduces energy consumption and comes within 4.4% of Ideal-NDA
Effect of Multiple Memory Stacks

![Graph showing speedup with different stack configurations.](chart.png)

- CPU-only
- NC
- CG
- FG
- CoNDA
- Ideal-NDA

Speedup

1 Stack 2 Stacks 4 Stacks

3.3x
Other Results in the Paper

- **Results for larger data sets**
  - $8.4x$ over CPU-only
  - $7.7x$ over NDA-only
  - $38.3\%$ over the best prior coherence mechanism

- **Sensitivity analysis**
  - Multiple memory stacks
  - Effect of optimistic execution duration
  - Effect of signature size
  - Effect of data sharing characteristics

- **Hardware overhead analysis**
  - 512 B NDA signature, 2 kB CPU signature, 1 bit per page table, 1 bit per TLB entry, 1.6\% increase in NDA L1 cache
Outline

• Introduction
• Background
• Motivation
• CoNDA
• Architecture Support
• Evaluation

• Conclusion
Conclusion

• Coherence is a major system challenge for NDA
  – Efficient handling of coherence is critical to retain NDA benefits

• We extensively analyze NDA applications and existing coherence mechanisms. Major Observations:
  – There is a significant amount of data sharing between CPU threads and NDAs
  – A majority of off-chip coherence traffic is unnecessary
  – A significant portion of off-chip traffic can be eliminated if the mechanism has insight into NDA memory accesses

• We propose CoNDA, a mechanism that uses optimistic NDA execution to avoid unnecessary coherence traffic

• CoNDA comes within 10.4% and 4.4% of performance and energy of an ideal NDA coherence mechanism
CoNDA: Efficient Cache Coherence Support for Near-Data Accelerators

Amirali Boroumand

Saugata Ghose, Minesh Patel, Hasan Hassan, Brandon Lucia, Rachata Ausavarungnirun, Kevin Hsieh, Nastaran Hajinazar, Krishna Malladi, Hongzhong Zheng, Onur Mutlu
More on CoNDA…

- Amirali Boroumand, Saugata Ghose, Minesh Patel, Hasan Hassan, Brandon Lucia, Rachata Ausavarungnirun, Kevin Hsieh, Nastaran Hajinazar, Krishna T. Malladi, Hongzhong Zheng, and Onur Mutlu,

"CoNDA: Efficient Cache Coherence Support for Near-Data Accelerators"
[Slides (pptx) (pdf)]
[Lightning Talk Slides (pptx) (pdf)]
[Poster (pptx) (pdf)]
[Lightning Talk Video (4 minutes)]
Transactional Memory: Architectural Support for Lock-Free Data Structures

Maurice Herlihy
Digital Equipment Corporation
Cambridge Research Laboratory
Cambridge MA 02139
herlihy@crl.dec.com

J. Eliot B. Moss
Dept. of Computer Science
University of Massachusetts
Amherst, MA 01003
moss@cs.umass.edu

Transactional Memory

- **Idea:** Programmer specifies code to be executed atomically as transactions. Hardware/software guarantees atomicity for transactions.

- Motivated by difficulty of lock-based programming
- Motivated by lack of concurrency (performance issues) in blocking synchronization (or “pessimistic concurrency”)
Locks vs. Transactions

Lock issues:
- Under-locking → data races
- Deadlock due to lock ordering
- Blocking synchronization
- Conservative serialization

How transactions help:
+ Simpler interface/reasoning
+ No ordering
+ Nonblocking (Abort on conflict)
+ Serialization only on conflicts

- Locks → pessimistic concurrency
- Transactions → optimistic concurrency
Transactional Memory

- Transactional Memory (TM) allows arbitrary multiple memory locations to be updated atomically (all or none)

- Basic Mechanisms:
  - **Isolation and conflict management**: Track read/writes per transaction, detect when a conflict occurs between transactions
  - **Version management**: Record new/old values (where?)
  - **Atomicity**: Commit new values or abort back to old values → all or none semantics of a transaction

- Issues the same as other speculative parallelization schemes
  - Logging/buffering
  - Conflict detection
  - Abort/rollback or commit
Lecture on Transactional Memory

Transactional Memory

- Idea: Programmer specifies code to be executed atomically as transactions. Hardware/software guarantees atomicity for transactions.

- Motivated by difficulty of lock-based programming
- Motivated by lack of concurrency (performance issues) in blocking synchronization (or “pessimistic concurrency”)

https://www.youtube.com/watch?v=McMyefc8CCE
Serialization of threads due to critical sections is a fundamental bottleneck to achieving high performance in multithreaded programs. Dynamically, such serialization may be unnecessary because these critical sections could have safely executed concurrently without locks. Current processors cannot fully exploit such parallelism because they do not have mechanisms to dynamically detect such false inter-thread dependences.
Speculative Lock Elision

- Many programs use locks for synchronization
- Many locks are not necessary
  - Stores occur infrequently during execution
  - Updates can occur to disjoint parts of the data structure

- Idea:
  - Speculatively assume lock is not necessary and execute critical section without acquiring the lock
  - Check for conflicts within the critical section
  - Roll back if assumption is incorrect

Dynamically Unnecessary Synchronization

Figure 1. Two examples of potential parallelism masked by dynamically unnecessary synchronization.
Speculative Lock Elision: Enabling Highly Concurrent Multithreaded Execution

Ravi Rajwar and James R. Goodman
Computer Sciences Department
University of Wisconsin-Madison
Madison, WI 53706 USA
{rajwar, goodman}@cs.wisc.edu

We propose Speculative Lock Elision (SLE), a novel micro-architectural technique to remove dynamically unnecessary lock-induced serialization and enable highly concurrent multithreaded execution. The key insight is that locks do not always have to be acquired for a correct execution. Synchronization instructions are predicted as being unnecessary and elided. This allows multiple threads to concurrently execute critical sections protected by the same lock. Misspeculation due to inter-thread data conflicts is detected using existing cache mechanisms and rollback is used for recovery. Successful speculative elision is validated and committed without acquiring the lock.
Speculative Lock Elision: Enabling Highly Concurrent Multithreaded Execution

Ravi Rajwar and James R. Goodman  
Computer Sciences Department  
University of Wisconsin-Madison  
Madison, WI 53706 USA  
{rajwar, goodman}@cs.wisc.edu

SLE can be implemented entirely in microarchitecture without instruction set support and without system-level modifications, is transparent to programmers, and requires only trivial additional hardware support. SLE can provide programmers a fast path to writing correct high-performance multithreaded programs.
Speculative Parallelization Concepts

- **Idea:** Execute threads unsafely in parallel
  - Threads can be from a sequential or parallel application

- **Hardware or software monitors for data dependence violations**

- If data dependence ordering is violated
  - Offending thread is squashed and restarted

- If data dependences are not violated
  - Thread commits
  - If threads are from a sequential order, the sequential order needs to be preserved → threads commit one by one in order
Speculative Lock Elision: Issues

- Either the entire critical section is committed or none of it
- How to detect the lock
- How to keep track of dependencies and conflicts in a critical section
  - Read set and write set
- How to buffer speculative state
- How to check if “atomicity” is violated
  - Dependence violations with another thread
- How to support commit and rollback

https://www.youtube.com/watch?v=McMyefc8CCE
Lecture on Thread-Level Speculation

Module 2.5 - Speculation (2 of 4) - 740: Computer Architecture 2013 - Carnegie Mellon - Onur Mutlu

https://www.youtube.com/watch?v=FqHk4bxrI8Y
Backup Slides
Breakdown of Performance Overhead

- **CoNDA’s execution time consist of three major parts:**
  - (1) NDA kernel execution
  - (2) Coherence resolution overhead (3.3% of execution time)
  - (3) Re-execution overhead (8.4% of execution time)

- **Coherence resolution overhead is low**
  - CPU-threads do not stall during resolution
  - NDAWriteSet contains only a small number of addresses (6)
  - Resolution mainly involves sending signatures and checking necessary coherence

- **Overhead of re-execution is low**
  - The collision rate is low for our applications → 13.4%
  - Re-execution is significantly faster than original execution
Memory System Energy

- **NC** suffers greatly from the *large number of accesses to DRAM*
- **Interconnect** and **DRAM** energy increase by *3.1x* and *4.5x*

**CG and FG** loses a significant portion of benefits because of
- large number of writebacks and off-chip coherence messages

**CoNDA** significantly reduces energy consumption
- and comes within 4.4% of Ideal-NDA
Speedup

- **CPU-only**
- **NDA-only**
- **NC**
- **CG**
- **FG**
- **CoNDA**
- **Ideal-NDA**

CoNDA consistently retains most of Ideal-NDA’s benefits, coming within 10.4% of the Ideal-NDA performance.

**NDA-only** eliminates 82.2% of Ideal-NDA’s improvement.

FG loses a significant portion of Ideal-NDA’s improvement.

The GMEAN across CC, Radii, PR, and NDA-only shows that NDA-only eliminates 82.2% of Ideal-NDA’s improvement.
Effect of Multiple Memory Stacks

![Graph showing the effect of multiple memory stacks on speedup. The graph compares different scenarios: CPU-only, NC, CG, FG, CoNDA, and Ideal-NDA. The x-axis represents 1 Stack, 2 Stacks, and 4 Stacks, while the y-axis represents speedup. The chart indicates a 3.3x speedup for 4 Stacks compared to the CPU-only scenario.](image)
Effect of Optimistic Execution Duration

<table>
<thead>
<tr>
<th>CC: Enron</th>
<th>HTAP-128</th>
</tr>
</thead>
<tbody>
<tr>
<td>150</td>
<td>150</td>
</tr>
<tr>
<td>250</td>
<td>250</td>
</tr>
<tr>
<td>350</td>
<td>350</td>
</tr>
</tbody>
</table>

- Normalized Execution Time
- Normalized Off-Chip Traffic
Effect of Signature Size

<table>
<thead>
<tr>
<th></th>
<th>Normalized Execution Time</th>
<th>Normalized Off-Chip Traffic</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>256</td>
<td>512</td>
</tr>
<tr>
<td>CC: Enron</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>256</td>
<td>512</td>
</tr>
<tr>
<td>HTAP-128</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Identifying Coherence Violations

1) NDA Read and CPU Write: **violation**

2) NDA Write and CPU Read: **no violation**

3) NDA Write and CPU Write: **no violation**
Example: Hybrid Database (HTAP)

Hybrid Database (HTAP)

Transactions

Analytics

Data Sharing

Transactions

Analytics

CPU

NDA