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 \( \rightarrow \) 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

![Diagram](image)

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**
Many parallel programs communicate through *shared memory*. Proc 0 writes to an address, followed by Proc 1 reading. This implies communication between the two. Each read should receive the value last written by anyone, which requires synchronization. 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?

Diagram:

- P1
- P2
- Interconnection Network
- Main Memory
- Value: 1000
- Value: x
The Cache Coherence Problem

ld r2, x
The Cache Coherence Problem

ld r2, x

P1

1000

P2

1000

ld r2, x

Interconnection Network

Main Memory

x 1000
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
The Cache Coherence Problem

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

```
ld r2, x  
ld r5, x
```

P1

```
2000
```

P2

```
1000
```

Interconnection Network

```
1000
```

Main Memory
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
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 store 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 do we want?**
  - Write frequency and sharing behavior are critical

- **Update**
  - If sharer set is constant and updates are infrequent, avoids the cost of invalidate-reatquire (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-reatquire 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

\[ p+1 \text{ bits: for block } A \]

\[ P=4 \]

\[ \begin{array}{c}
A \\
000000
\end{array} \]

Exclusive bit

No cache has the block

1. \( P_1 \) takes a read miss to block A

\[ \begin{array}{c}
000000 \rightarrow 010000
\end{array} \]

2. \( P_3 \) takes a read miss

\[ \begin{array}{c}
01010
\end{array} \]
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 disk
   - 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 actions on a block:
- PR (Proc. Read)
- RW (Proc. Write)

Bus actions on a block:
- BR (Bus Read)
- BW (Bus Write)
- or BRx (Bus Read Exclusive)
A Simple Snoopy Cache Coherence Protocol

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

```
PrRd/--  PrWr / BusWr
\     /   \\
|   |     |
|   |     |
|   |     |
V   V     V

PrRd / BusRd

Valid  BusWr

Invalid

PrWr / BusWr
```

- **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**(odified): cache line is the only cached copy and is dirty
  - **S**(hared): cache line is one of potentially several cached copies and it is clean (i.e., at least one clean cached copy)
  - **I**(nvalid): 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*)
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 Solution: MESI

- 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!

- MESI is also called the *Illinois protocol*
Papamarcos & Patel, ISCA 1984

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
MESIS 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 Optional Lab 5
Intel Pentium Pro Coherence Protocol

- **INV**
- **SHARED (Clean)**
- **MODIFIED**
- **EXCLUSIVE (Clean)**

- **Write Allocate**
- **L1 can have data not in L2**
- **Hit**: Someone has it
  - **Clean**: Someone has it clean
  - **Dirty**: 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 MASI

- 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.
Tradeoffs in Sophisticated Cache Coherence Protocols

- 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 \(\rightarrow\) 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 \(\rightarrow\) dir. \(\rightarrow\) 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

$P = 4$

$000000$  

Exclusive bit

No cache has the block

1. $P_1$ takes a read miss to block A

$000000 \rightarrow 010000$

2. $P_3$ takes a read miss

$010100$
Directory-Based Protocols

- Required when scaling past the capacity of a single bus
- Distributed:
  - Coherence still requires single point of serialization (for write serialization)
  - Serialization location can be different for every block (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 buses)
### 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>Address</th>
<th>State</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x00</td>
<td>Shared: {P0, P1, P2}</td>
</tr>
<tr>
<td>0x04</td>
<td>---</td>
</tr>
<tr>
<td>0x08</td>
<td>Exclusive: P2</td>
</tr>
<tr>
<td>0x0C</td>
<td>---</td>
</tr>
<tr>
<td>...</td>
<td>---</td>
</tr>
</tbody>
</table>
Directory: Basic Operations

- Follow *semantics* of 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)
RdEx with Former Owner

1. RdEx
2. Invl
3a. Rev
3b. DatEx
Contestion Resolution (for Write)

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

1b. RdEx
3. RdEx
2b. NACK

5b. DatEx
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
  - (32GB address space / 128 bytes per block) / 32 nodes
  - \(2^{(35-7-5)} = 2^{23}\)

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

- 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
  - \(P+1 = 200 \Rightarrow P = 199\)
Cache Coherence:
A Recent Example
Automatic Data Coherence Support for PIM

- 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"


LazyPIM: An Efficient Cache Coherence Mechanism for Processing-in-Memory

Amirali Boroumand†, Saugata Ghose†, Minesh Patel†, Hasan Hassan†§, Brandon Lucia†,
Kevin Hsieh†, Krishna T. Malladi*, Hongzhong Zheng*, and Onur Mutlu‡†

†Carnegie Mellon University  *Samsung Semiconductor, Inc.  §TOBB ETÜ  ‡ETH Zürich
Automatic Data Coherence Support for PIM

- 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)]

CoNDA: Efficient Cache Coherence Support for Near-Data Accelerators

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

†Carnegie Mellon University
‡Carnegie Mellon University
○Simon Fraser University
§Samsung Semiconductor, Inc.
*ETH Zürich
‡‡Carnegie Mellon University
§§KMUTNB
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 are now everywhere!

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

**1st key observation:** CPU threads often concurrently access the same region of data that NDA kernels access which leads to significant data sharing.

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

**Hybrid Databases (HTAP)**

**Graph Processing**
2\textsuperscript{nd} key observation: CPU threads and NDA kernels typically \textit{do not} concurrently access the same cache lines.

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

CPU threads \textit{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

- High amount of off-chip coherence traffic
  - Memory-intensive
  - Poor locality

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.

Loses a significant portion of the performance and energy benefits.

CPU-only
NC
CG
FG
Ideal-NDA

Speedup

Normalized Energy

GMEAN

arXiv

CC
Radii
PR
CC
Radii
PR

0.0
0.5
1.0
1.5
2.0

0.0
0.5
1.0
1.5
2.0

NC suffers from a large number of off-chip accesses from CPU threads.

CG unnecessarily flushes a large amount of dirty data.

FG suffers from high amount of unnecessary off-chip coherence traffic.

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

High-Level Overview of Optimistic Execution Model

CPU Thread Execution

Concurrent CPU + NDA Execution

No Coherence Request

Coherence Resolution

Commit or Re-execute

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**

Outline

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

CPU

Shared LLC

Coherence Resolution

CPUWriteSet

DRAM

LI

NDA Core

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**
Coherence Resolution

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

![Bar chart showing speedup with different stack configurations: 1 Stack, 2 Stacks, 4 Stacks. The chart compares CPU-only, NC, CG, FG, CoNDA, and Ideal-NDA scenarios. The speedup is marked up to 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)]
An Example Parallel Problem: Task Assignment to Processors
Static versus Dynamic Scheduling

- **Static**: Done at compile time or parallel task creation time
  - Schedule does not change based on runtime information

- **Dynamic**: Done at run time (e.g., after tasks are created)
  - Schedule changes based on runtime information

- **Example**: Instruction scheduling
  - Why would you like to do dynamic scheduling?
  - What pieces of information are not available to the static scheduler?
Parallel Task Assignment: Tradeoffs

- Problem: N tasks, P processors, N>P. Do we assign tasks to processors statically (fixed) or dynamically (adaptive)?

- Static assignment
  + Simpler: No movement of tasks.
  - Inefficient: Underutilizes resources when load is not balanced
    *When can load not be balanced?*

- Dynamic assignment
  + Efficient: Better utilizes processors when load is not balanced
  - More complex: Need to move tasks to balance processor load
  - Higher overhead: Task movement takes time, can disrupt locality
Parallel Task Assignment: Example

- Compute histogram of a large set of values
- Parallelization:
  - Divide the values across $T$ tasks
  - Each task computes a local histogram for its value set
  - Local histograms merged with global histograms in the end

```cpp
getPageHistogram(Page *P)
    For each thread: {
        /* Parallel part of the function */
        UpdateLocalHistogram(Fraction of Page)
        /* Serial part of the function */
        Critical Section:
        Add local histogram to global histogram
        Barrier
    }
    Return global histogram
```
Parallel Task Assignment: Example (II)

- How to schedule tasks updating local histograms?
  - Static: Assign equal number of tasks to each processor
  - Dynamic: Assign tasks to a processor that is available
  - When does static work as well as dynamic?

- Implementation of Dynamic Assignment with Task Queues

![Diagram of task assignment and queuing](image)

(a) Distributed Task Stealing
(b) Hierarchical Task Queuing
Software Task Queues

- What are the advantages and disadvantages of each?
  - Centralized
  - Distributed
  - Hierarchical

(a) Distributed Task Stealing
(b) Hierarchical Task Queuing
Task Stealing

- **Idea:** When a processor’s task queue is empty it steals a task from another processor’s task queue
  - Whom to steal from? (Randomized stealing works well)
  - How many tasks to steal?

+ Dynamic balancing of computation load

- Additional communication/synchronization overhead between processors
- Need to stop stealing if no tasks to steal
Parallel Task Assignment: Tradeoffs

- Who does the assignment? Hardware versus software?

- Software
  + Better scope
  - More time overhead
  - Slow to adapt to dynamic events (e.g., a processor becoming idle)

- Hardware
  + Low time overhead
  + Can adjust to dynamic events faster
  - Requires hardware changes (area and possibly energy overhead)
How Can the Hardware Help?

- Managing task queues in software has overhead
  - Especially high when task sizes are small

- An idea: Hardware Task Queues
  - Each processor has a dedicated task queue
  - Software fills the task queues (on demand)
  - Hardware manages movement of tasks from queue to queue
  - There can be a global task queue as well → hierarchical tasking in hardware

  - Optional reading
Dynamic Task Generation

- Does static task assignment work in this case?

- Problem: Searching the exit of a maze


```plaintext
while(problem not solved)
  SubProblem = PriorityQ.remove()
  Solve(SubProblem)
  if(solved)
    break
  NewSubProblems = Partition(SubProblem)
  PriorityQ.insert(NewSubProblems)
```
Programming Model vs. Hardware Execution Model
Programming Models vs. Architectures

- Five major models
  - (Sequential)
  - Shared memory
  - Message passing
  - Data parallel (SIMD)
  - Dataflow
  - Systolic

- Hybrid models?
Shared Memory vs. Message Passing

- Are these programming models or execution models supported by the hardware architecture?

- Does a multiprocessor that is programmed by “shared memory programming model” have to support a shared address space between processors?

- Does a multiprocessor that is programmed by “message passing programming model” have to have no shared address space between processors?
Programming Models: Message Passing vs. Shared Memory

- **Difference:** how communication is achieved between tasks

  - **Message passing programming model**
    - Explicit communication via messages
    - Loose coupling of program components
    - Analogy: telephone call or letter, no shared location accessible to all

  - **Shared memory programming model**
    - Implicit communication via memory operations (load/store)
    - Tight coupling of program components
    - Analogy: bulletin board, post information at a shared space

- **Suitability of the programming model** depends on the problem to be solved. Issues affected by the model include:
  - Overhead, scalability, ease of programming, bugs, match to underlying hardware, ...
Message Passing vs. Shared Memory Hardware

- **Difference:** how task communication is supported in hardware

- **Shared memory hardware (or machine model):**
  - All processors see a global shared address space
    - Ability to access all memory from each processor
  - A write to a location is visible to the reads of other processors

- **Message passing hardware (machine model):**
  - No global shared address space
  - Send and receive variants are the only method of communication between processors (much like networks of workstations today, i.e. clusters)

- Suitability of the hardware depends on the problem to be solved as well as the programming model.
Most of parallel computing history, there was no separation between programming model and hardware

- Message passing: Caltech Cosmic Cube, Intel Hypercube, Intel Paragon
- Shared memory: CMU C.mmp, Sequent Balance, SGI Origin
- SIMD: ILLIAC IV, CM-1

However, any hardware can really support any programming model

Why?

- Application $\rightarrow$ compiler/library $\rightarrow$ OS services $\rightarrow$ hardware
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 \( \rightarrow 13.4\% \)
  - 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**

- **NDA-only** eliminates 82.2% of Ideal-NDA’s improvement
- **FG** loses a significant portion of Ideal-NDA’s benefits, coming within 10.4% of the Ideal-NDA performance
- **CoNDA** consistently retains most of Ideal-NDA’s benefits, coming within 11.4% of the Ideal-NDA performance
- **CPU-only** shows the baseline performance
Effect of Multiple Memory Stacks

![Graph showing the speedup for different configurations of memory stacks. The x-axis represents the number of stacks (1 Stack, 2 Stacks, 4 Stacks), and the y-axis represents the speedup. The graph compares various configurations: CPU-only, NC, CG, FG, CoNDA, and Ideal-NDA. The speedup for 4 Stacks is marked as 3.3x.]
Effect of Optimistic Execution Duration

<table>
<thead>
<tr>
<th></th>
<th>Normalized Execution Time</th>
<th>Normalized Off-Chip Traffic</th>
</tr>
</thead>
<tbody>
<tr>
<td>150</td>
<td></td>
<td></td>
</tr>
<tr>
<td>250</td>
<td></td>
<td></td>
</tr>
<tr>
<td>350</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

CC: Enron

HTAP-128

SAFARI
Effect of Signature Size

- Normalized Execution Time
- Normalized Off-Chip Traffic

**CC: Enron**
- 256: 0.6
- 512: 0.6
- 1024: 0.6

**HTAP-128**
- 256: 0.6
- 512: 0.6
- 1024: 0.6
## Identifying Coherence Violations

<table>
<thead>
<tr>
<th>Time</th>
<th>CPU</th>
<th>NDA</th>
<th>Effective Ordering</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Cl. Wr Z</td>
<td>Cl. Wr Z</td>
<td>Cl. Wr Z</td>
</tr>
<tr>
<td>C1.</td>
<td>No. Rd X</td>
<td>N5. Wr Y</td>
<td>N4. Rd X</td>
</tr>
<tr>
<td>C2.</td>
<td>No. Rd Y</td>
<td>N3. Wr Y</td>
<td>N5. Wr Y</td>
</tr>
<tr>
<td></td>
<td>Cl. Wr X</td>
<td>C6. Wr X</td>
<td>C6. Wr X</td>
</tr>
<tr>
<td></td>
<td>C5. Rd Y</td>
<td>N2. Wr Y</td>
<td></td>
</tr>
</tbody>
</table>

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

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

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

---

### 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)