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 \(\rightarrow\) 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)
Cache Coherence
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)?
Cache Coherence

- Basic question: If multiple processors cache the same block, how do they ensure they all see a consistent state?
The Cache Coherence Problem

P1

Interconnection Network

P2

ld r2, x

Main Memory

1000

x

1000
The Cache Coherence Problem

P1

P2

Id r2, x

1000

Interconnection Network

Id r2, x

1000

Main Memory

x 1000
The Cache Coherence Problem

ld r2, x
add r1, r2, r4
st x, r1
The Cache Coherence Problem

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

ld r2, x
ld r5, x

Should NOT load 1000

ld r2, x

Interconnection Network

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:

- PrRd/--
- PrWr / BusWr
- PrRd / BusRd
- BusWr
- 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
(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 bandwidth 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-reaquire (broadcast update pattern)
  - If data is rewritten without intervening reads by other cores, updates were useless
  - Write-through cache policy → bus becomes bottleneck

- **Invalidate**
  + After invalidation broadcast, 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-reaquire 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 invalidation 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

$p = 4$

$\begin{array}{cccccc}
0 & 0 & 0 & 0 & 0
\end{array}$

Exclusive bit

No cache has the block

$p_4$ takes a read miss to block A

$\begin{array}{cccccc}
0 & 0 & 0 & 0 & 0
\end{array} \rightarrow \begin{array}{cccccc}
0 & 1 & 0 & 0 & 0
\end{array}$

$p_3$ takes a read miss

$\begin{array}{cccccc}
0 & 1 & 0 & 1 & 1
\end{array}$
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 director.
- P2 needs to have a bit in its cache indicating it can perform exclusive updates to that block.
- Private/exclusive bit per cache block

P3 takes a write miss

- Mem controller requests block from P2
- Mem controller gives block to P3
- P2 invalidates its copy

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 1000-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 acting to a block:
- PR (Proc. Read)
- RW (Proc. Write)

Bus acting to a block:
- BR (Bus Read)
- BW (Bus Write)
  or BRx (Bus Read Exclusive)

coheren state bits in tag size (e.g., MESI)
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**(odified): cache line is the only cached copy and is dirty
  - **S**(hared): cache line is potentially one of several cached copies and it is clean
  - **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 re-reading data from memory (via *Invalidations*)
MSI State Machine

ObservedEvent/Action

[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 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* → *Modified* is possible on write!

- MESI 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

**Notes**

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

Slide credit: Yale Patt
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 transfer
  - 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: necessary?
  - 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
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 → 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

\[ P = 4 \]

\[ \begin{array}{cc}
1 & 0 \\
2 & 0 \\
3 & 0 \\
4 & 0 \\
5 & 0 \\
\end{array} \]

\[ \text{Exclusive bit} \]

No cache has the block

\[ P_1 \text{ takes a read miss to block A} \]

\[ \begin{array}{cc}
1 & 0 \\
2 & 0 \\
3 & 0 \\
4 & 0 \\
5 & 0 \\
\end{array} \rightarrow \begin{array}{cc}
1 & 0 \\
2 & 1 \\
3 & 0 \\
4 & 0 \\
5 & 0 \\
\end{array} \]

\[ P_3 \text{ takes a read miss} \]

\[ \begin{array}{cc}
1 & 0 \\
2 & 1 \\
3 & 0 \\
4 & 0 \\
5 & 0 \\
\end{array} \rightarrow \begin{array}{cc}
1 & 0 \\
2 & 1 \\
3 & 1 \\
4 & 0 \\
5 & 0 \\
\end{array} \]
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)
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
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

Diagram:

- P0
- Home
- Owner

Arrows indicate the flow of processes.
Contention Resolution (for Write)

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

1b. RdEx
2b. NACK
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 ($2^{35}$ 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
  - \((32\text{GB address space} \div 128\text{ bytes per block}) \div 32\text{ nodes}\)
  - \(2^{(35-7-5)} = 2^{23}\)

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

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

- 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, Kevin Hsieh, Krishna T. Malladi, Hongzhong Zheng, and Onur Mutlu,

"CoNDA: Efficient Cache Coherence Support for Near-Data Accelerators"

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!

Recent advancement in 3D-stacked 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
Analysis of Existing Coherence Mechanism

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
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 permission**
2. Gains **insights** into memory accesses

When execution is done:

Perform **only the necessary** coherence requests
High-Level Overview of Optimistic Execution Model

- **CPU**
  - Starts optimistic execution
  - Concurrent CPU + NDA Execution
  - Coherence Resolution
    - Commit or Re-execute

- **NDA**
  - Optimistic Execution

**Time**

- CPU Thread Execution
- No Coherence Request
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)
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**
  - **NDA Core**
    - NDAReadSet
    - NDAWriteSet

- **Li**
Optimistic Mode Execution

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 dealing with conflicts happens:

- The CPU flushes the dirty cache lines that match addresses in the NDAReadSet.
- NDA invalidates all uncommitted cache lines.
- Signatures are erased and NDA restarts execution.

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.
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, Kevin Hsieh, Krishna T. Malladi, Hongzhong Zheng, and Onur Mutlu,

"CoNDA: Efficient Cache Coherence Support for Near-Data Accelerators"

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
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

Flush dirty data

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 cache and performs 0.4% worse than CPU-only cache
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

(1) Memory-intensive
(2) Poor locality

High amount of off-chip coherence Traffic

FG eliminates 71.8% of the energy benefits of an ideal NDA mechanism
**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*

**Normalized Energy**

- **CoNDA** significantly reduces energy consumption and comes within 4.4% of Ideal-NDA.
- **CG** and **FG** lose a significant portion of benefits because of large number of writebacks and off-chip coherence messages.

---

**GMEAN**

- **Ideal-NDA**
**Speedup**

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

---

- **FG** loses a significant portion of Ideal-NDA's improvement.
- **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.
Effect of Multiple Memory Stacks

![Speedup Chart]

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

Speedup

1 Stack | 2 Stacks | 4 Stacks

3.3x
Effect of Optimistic Execution Duration

![Graph showing normalized execution time and normalized off-chip traffic for CC: Enron and HTAP-128 with data points for 150, 250, and 350.]
Effect of Signature Size

Normalized Execution Time vs. Normalized Off-Chip Traffic

- CC: Enron
- HTAP-128

<table>
<thead>
<tr>
<th></th>
<th>256</th>
<th>512</th>
<th>1024</th>
</tr>
</thead>
<tbody>
<tr>
<td>CC: Enron</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>HTAP-128</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

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

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

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

We leverage two key observations:

1. Majority of coherence
2. Enforce coherence with only the necessary data movement

We propose to use optimistic execution for NDAs.

When executing in optimistic mode:

- An NDA gains insight into its memory accesses without issuing any coherence requests.

When optimistic mode is done:

- The NDA uses the tracking information to perform necessary coherence requests.
Example: Hybrid Database (HTAP)
Application Analysis Wrap up

1. There is a significant amount of data sharing between CPU threads and NDAs.

2. CPU threads and NDAs often do not access the same cache lines concurrently.

3. CPU threads rarely update the same data that NDAs are actively working on.
Background

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

- Enabled by recent advances in 3D-stacked memory
Specialized accelerators are now everywhere!

On-chip Accelerators:
- GPU
- FPGA
- ASIC

Off-chip Accelerators:
- Near-Data Accelerator
Applications

• **Ligra**
  - Lightweight multithreaded graph processing for shared memory system
  - We used three Ligra graph applications
    • PageRank (PR)
    • Radii
    • Connected Components (CC)
  - Input graphs constructed from real-world network datasets:
    • Enron email communication network (36K nodes, 183K edges)
    • arXiv General Relativity (5K nodes, 14K edges)
    • peer-to-peer Gnutella25 (22K nodes, 54K edges).

• **IMDB**
  - In-house prototype of an in-memory database (IMDB)
  - Capable of running both transactional queries and analytical queries on the same database tables (HTAP workload)
  - 32K transactions, 128/256 analytical queries
Optimistic NDA Execution

We leverage two key observations:

1. Eliminate much of unnecessary coherence traffic by having insight into memory accesses
2. CPU threads and NDA kernels typically do not concurrently access the same cache lines

We propose to use optimistic execution for NDAs

NDA executes the kernel:

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

When execution is done:

Performs only the necessary coherence requests
Analysis of Existing Coherence Mechanisms

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

Suffers from a large number of off-chip accesses.

Unnecessarily flushes a large amount of dirty data.

Blocks CPU threads when they access NDA data regions.