Memory Coherence in Shared Virtual Memory Systems

Yeong Ouk Kim, Hyun Gi Ahn
Contents

• Introduction
• Loosely coupled multiprocessor
• Shared virtual memory
• Memory coherence
• Centralized manager algorithms
• Distributed manager algorithms
• Experiments
• Conclusion
Introduction

• Designing a shared virtual memory for a loosely coupled multiprocessor that deals with the memory coherence problem

• Loosely coupled multiprocessor
• Shared virtual memory
• Memory coherence
Loosely Coupled Multiprocessor

• Processors (Machines) connected via messages (network) to form a cluster
• This paper is not about distributed computing
• The key goal is to allow processes (threads) from a program to execute on different processors in parallel
• Slower (due to message delay and low data rate) but cheaper compared to tightly coupled systems
Shared Virtual Memory

- Virtual address space shared among all processors
- Read-only pages in multiple locations
- Writable pages can only be in one place at a time
- Page fault when the referenced page is not in the current processor’s physical memory

Fig. 1. Shared virtual memory mapping.
Locality

• Frequently referencing the same data
• Virtual Memory with sequential programs → High degree of locality
• Parallel programs, while as a whole show low locality, each individual process is still a sequential program thus should show high locality
Memory Coherence

• Memory is coherent if the value returned by a read operation is always the same as the most recent write operation to the same address.

• Architecture with one memory access is free from this problem but it is not sufficient for high performance demands.

• Loosely coupled multiprocessor → communication between processors is costly → less communication more work.
Two Design Choices for SVM

- Granularity
- Strategy for maintaining coherence
Granularity

• AKA page size
• The size doesn’t really effect the communication cost
  • Small packet ≒ Large packet
• However, larger page size can bring more contention on the same page → relatively small page size is preferred
• Rule of thumb → choose a typical page size used in a conventional virtual memory → the paper suggests 1KB
Memory Coherence Strategies

• Page synchronization
  • Invalidation
  • Write-broadcast

• Page ownership
  • Fixed
  • Dynamic
    • Centralized
    • Distributed
      • Fixed
      • Dynamic
Page synchronization

- Invalidation: one owner per page
- Write-broadcast: writes to every copy when a write happens
Invalidation (page synchronization #1)

- Write fault (processor $Q$ and page $p$)
  - Invalidate all copies of $p$
  - Changes the access of $p$ to write
  - Copy $p$ to $Q$ if $Q$ doesn’t already have one
  - Return to the faulting instruction

- Read fault
  - Changes the access of $p$ to read on the processor that has write access to $p$
  - Move a copy of $p$ to $Q$ and set its access to read
  - Return to the faulting instruction
Write-broadcast (page synchronization #2)

• Write-fault
  • Writes to all copies of the page
  • Return to the faulting instruction

• Read-fault
  • Identical to invalidation
Page ownership

• Fixed

• Dynamic
  • Centralized
  • Distributed
    • Fixed
    • Dynamic
Fixed Ownership

• A page is always owned by the same processor
• Other processors will never get full write access to the page
  • Negotiate with the owner → generate write-fault every time
• Impossible for invalidation and expensive for write-broadcast

→ Not considered
Solutions to Memory Coherence Problem

<table>
<thead>
<tr>
<th>Page synchronization method</th>
<th>Page ownership strategy</th>
<th>Dynamic</th>
<th>Distributed manager</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Fixed</td>
<td>Centralized manager</td>
<td>Fixed</td>
</tr>
<tr>
<td>Invalidation</td>
<td>Not allowed</td>
<td>Okay</td>
<td>Good</td>
</tr>
<tr>
<td>Write-broadcast</td>
<td>Very expensive</td>
<td>Very expensive</td>
<td>Very expensive</td>
</tr>
</tbody>
</table>
Data Structure (page table)

• Access: indicates the accessibility to the page
  • i.e. Read or Write

• Copy set: list of processors that own a read copy of the page
  • i.e. {1, 3}

• Lock: Synchronizing multiple page faults by different processes on the same processor and synchronizing remote page requests
Remote operations

- Any remote operation will require two messages
  - Request and reply
- Invalidation for $m$ copies on an $N$ processor system

<table>
<thead>
<tr>
<th></th>
<th>Point-to-point</th>
<th>Broadcast</th>
<th>multicast</th>
</tr>
</thead>
<tbody>
<tr>
<td>Requests</td>
<td>$m$</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>Replies</td>
<td>$m$</td>
<td>$m$</td>
<td>$m$</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>Point-to-point</th>
<th>Broadcast</th>
<th>multicast</th>
</tr>
</thead>
<tbody>
<tr>
<td>Send</td>
<td>$2m$</td>
<td>$m+1$</td>
<td>$m+1$</td>
</tr>
<tr>
<td>Receive</td>
<td>$2m$</td>
<td>$N+m-1$</td>
<td>$2m$</td>
</tr>
</tbody>
</table>

$N-1$ in parallel $\rightarrow m+1$ in best case
$m$ in parallel $\rightarrow m+1$ in best case
**Centralized Manager**

- Centralized manager (monitor) resides on a single processor
- Every page-fault needs to consult with this monitor

<table>
<thead>
<tr>
<th>Manager</th>
<th>P1</th>
</tr>
</thead>
<tbody>
<tr>
<td>Info</td>
<td></td>
</tr>
<tr>
<td>Owner</td>
<td>Processor that owns the page</td>
</tr>
<tr>
<td>Copy set</td>
<td>Enables non-broadcast invalidation</td>
</tr>
<tr>
<td>Lock</td>
<td>Synchronize request to the page</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Each Processor</th>
<th>P1</th>
<th>P2</th>
<th>P3</th>
</tr>
</thead>
<tbody>
<tr>
<td>PTable</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Access</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Lock</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Synchronization within the processor</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Demo (centralized manager)

Read page fault in P2

---

<table>
<thead>
<tr>
<th>Info</th>
<th>Owner</th>
<th>Copy set</th>
<th>Lock</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>P3</td>
<td>{P1}</td>
<td>0</td>
</tr>
</tbody>
</table>

---

<table>
<thead>
<tr>
<th>PTable</th>
<th>Access</th>
<th>Lock</th>
</tr>
</thead>
<tbody>
<tr>
<td>P1</td>
<td>R</td>
<td>0</td>
</tr>
</tbody>
</table>

---

<table>
<thead>
<tr>
<th>PTable</th>
<th>Access</th>
<th>Lock</th>
</tr>
</thead>
<tbody>
<tr>
<td>P2</td>
<td></td>
<td>0</td>
</tr>
<tr>
<td>P3</td>
<td>R</td>
<td>0</td>
</tr>
</tbody>
</table>
Demo (centralized manager)

Read page fault in P2

1. Request
Demo (centralized manager)

Read page fault in P2

1. Request
Demo (centralized manager)

Read page fault in P2

1. Request
2. Ask to send copy to P2
Demo (centralized manager)

Read page fault in P2

1. Request
2. Ask to send copy to P2
3. Send Copy to P2
Demo (centralized manager)

Read page fault in P2

1. Request
2. Ask to send copy to P2
3. Send Copy to P2
4. Confirmation

<table>
<thead>
<tr>
<th>Info</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Owner</td>
<td>P3</td>
</tr>
<tr>
<td>Copy set</td>
<td>{P1, P2}</td>
</tr>
<tr>
<td>Lock</td>
<td>1</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>PTable</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Access</td>
<td>R</td>
</tr>
<tr>
<td>Lock</td>
<td>0</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>PTable</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Access</td>
<td>R</td>
</tr>
<tr>
<td>Lock</td>
<td>0</td>
</tr>
</tbody>
</table>
Demo (centralized manager)

Read page fault in P2

1. Request
2. Ask to send copy to P2
3. Send Copy to P2
4. Confirmation
Demo (centralized manager)

Write page fault in P2

<table>
<thead>
<tr>
<th>Info</th>
<th>P1</th>
</tr>
</thead>
<tbody>
<tr>
<td>Owner</td>
<td>P3</td>
</tr>
<tr>
<td>Copy set</td>
<td>{P1, P2}</td>
</tr>
<tr>
<td>Lock</td>
<td>0</td>
</tr>
</tbody>
</table>

PTable

Access | R
Lock   | 0

P2

PTable

Access | R
Lock   | 0

P3

PTable

Access | R
Lock   | 0
Demo (centralized manager)

Write page fault in P2

1. Request

```
<table>
<thead>
<tr>
<th>Info</th>
<th>Owner</th>
<th>Copy set</th>
<th>Lock</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>P3</td>
<td>{P1, P2}</td>
<td>0</td>
</tr>
</tbody>
</table>

P1

<table>
<thead>
<tr>
<th>PTable</th>
<th>Access</th>
<th>Lock</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>R</td>
<td>0</td>
</tr>
</tbody>
</table>

P3

<table>
<thead>
<tr>
<th>PTable</th>
<th>Access</th>
<th>Lock</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>R</td>
<td>0</td>
</tr>
</tbody>
</table>

1. Request

P2

<table>
<thead>
<tr>
<th>PTable</th>
<th>Access</th>
<th>Lock</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>R</td>
<td>1</td>
</tr>
</tbody>
</table>
```
Demo (centralized manager)

Write page fault in P2

1. Request
2. Invalidation
Demo (centralized manager)

Write page fault in P2

1. Request
2. Invalidation
3. Ask to send page to P2
Demo (centralized manager)

Write page fault in P2

1. Request
2. Invalidation
3. Ask to send page to P2
4. Send Copy to P2
Demo (centralized manager)

Write page fault in P2

1. Request
2. Invalidation
3. Ask to send page to P2
4. Send Copy to P2
5. Confirmation

<table>
<thead>
<tr>
<th>Info</th>
<th>Owner</th>
<th>Copy set</th>
<th>Lock</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>P3</td>
<td>{}</td>
<td>1</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>PTable</th>
<th>Access</th>
<th>Lock</th>
</tr>
</thead>
<tbody>
<tr>
<td>P1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>P2</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>P3</td>
<td>0</td>
<td></td>
</tr>
</tbody>
</table>
Demo (centralized manager)

Write page fault in P2

1. Request
2. Invalidation
3. Ask to send page to P2
4. Send Copy to P2
5. Confirmation
Overview (centralized manager)

• Locking is handled separately (local, remote)
• Traffic bottleneck at the at the manager
  • Worse as N gets bigger

• Worst-case number of messages to locate a page is two (manager → owner)
• This is considerably low but the confirmation message is required every time there is a fault

→ Eliminate confirmation messages for improvement
Improved Centralized Manager

• Synchronization of page ownership has been moved to the individual owners → eliminating the confirmation operation to the manager

• Lock in each processor now also deals with synchronizing remote requests (while still dealing with multiple local requests)

• Manager still answers where the page owner is
Improved Centralized Manager

• Modify the data structure to accommodate
• Copy set is only valid iff the processor is the owner of the page

<table>
<thead>
<tr>
<th>Manager</th>
<th>P1</th>
</tr>
</thead>
<tbody>
<tr>
<td>Owner</td>
<td></td>
</tr>
<tr>
<td>Owner</td>
<td>Processor that owns the page</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Each Processor</th>
<th>P1</th>
<th>P2</th>
<th>P3</th>
</tr>
</thead>
<tbody>
<tr>
<td>PTable</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Access</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Lock</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Now deals with local and remote requests</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Copy set</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Enables non-broadcast invalidation</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Improved Centralized Manager

Read server:

\[\text{Lock}(\text{PTable}[p].\text{lock})\];
IF I am owner THEN BEGIN
  PTable[ p ].access := read;
send copy of p;
END;
Unlock( PTable[ p ].lock );

IF I am manager THEN BEGIN
  Lock( Info[ p ].lock );
  Info[ p ].copyset := Info[ p ].copyset \cup \{\text{RequestNode}\};
  ask Info[ p ].owner to send copy of p to RequestNode;
  receive confirmation from RequestNode;
  Unlock( Info[ p ].lock );
END;

Read server:

\[\text{Lock}(\text{PTable}[p].\text{lock})\];
IF I am owner THEN BEGIN
  PTable[ p ].copyset := PTable[ p ].copyset \cup \{\text{RequestNode}\};
  PTable[ p ].access := read;
send p;
END

ELSE IF I am manager THEN BEGIN
  Lock( ManagerLock );
  forward request to owner[ p ];
  Unlock( ManagerLock );
END;
Unlock( PTable[ p ].lock );
Improved Centralized Manager

• Compared to previous implementation
• Improves performance by decentralizing the synchronization
  • Read page faults $\rightarrow$ for all processors saves one send and one receive
• But on large $N$ $\rightarrow$ bottleneck still exists since the manager must respond to every page fault

$\rightarrow$ We must distribute the ownership management duty
A Fixed Distributed Manager Algorithm

• Each processor is given a predetermined subset of pages to manage

• Mapping pages to appropriate processors is a difficult task and uses a hash function

• Page fault handling applies the hash function to get the manager for the page and follows the same procedure as centralized manager

• It is difficult to find a good fixed distribution function that fits all application well
A Broadcast Distributed Manager Algorithm (1)

Read page fault

1. requestor: lock PTable[page#]
2. requestor: broadcast read access request
3. owner: lock PTable[page#]
4. owner: add requestor to PTable[page#].copy-set
5. owner: PTable[page#].access = read
6. requestor: receive the page from the owner
7. requestor: PTable[page#].access = read
8. requestor: unlock PTable[page#]
9. owner: send the copy to requestor
10. owner: unlock PTable[page#]
A Broadcast Distributed Manager Algorithm (2)

Write page fault

<table>
<thead>
<tr>
<th>Requestor</th>
<th>Owner</th>
</tr>
</thead>
<tbody>
<tr>
<td>P1</td>
<td>P2</td>
</tr>
</tbody>
</table>

1. requestor: lock PTable[page#]
2. requestor: broadcast write access request
3. owner: lock PTable[page#]
4. owner: send the page and copy-set to requestor
5. requestor: receive the page & copy-set from the owner
6. requestor: invalidate(page, PTable[page#].copy-set)
7. requestor: PTable[page#].access = write
8. requestor: PTable[page#].copy-set = empty
9. requestor: PTable[page#].owner = myself
10. requestor: unlock PTable[page#]

(2) owner: PTable[page#].access = null
(6) owner: unlock PTable[page#]
A Dynamic Distributed Manager Algorithm (1)

Read page fault

(1) requestor: lock PTable[page#]
(2) requestor: send read access request to PTable[page#].probOwner
(3) probOwner: lock PTable[page#]
(4) probOwner: forward the request to PTable[page#].probOwner, if I am not the real owner
(5) probOwner: unlock PTable[page#]
(5) owner: lock PTable[page#]
(6) owner: add requestor to PTable[page#].copy-set
(7) owner: PTable[page#].access = read
(8) owner: send the page to the requestor
(9) owner: unlock PTable[page#]
(8) requestor: receive the page from the owner
(9) requestor: PTable[page#].probOwner = owner
(10) requestor: PTable[page#].access = read
(11) requestor: unlock PTable[page#]
A Dynamic Distributed Manager Algorithm (2)

Write page fault

(1) requestor: lock PTable[page#]
(2) requestor: send write access request to PTable[page#].probOwner
(3) probOwner: lock PTable[page#]
(4) probOwner: forward the request to PTable[page#].probOwner, if not the real owner
(5) probOwner: PTable[page#].probOwner = requestor
(6) probOwner: unlock PTable[page#]
(5) owner: lock PTable[page#]
(6) owner: PTable[page#].access = null
(7) owner: send the page and copy-set to the requestor
(8) owner: PTable[page#].probOwner = requestor
(9) owner: unlock PTable[page#]
(7) requestor: receive the page & copy-set from the owner
(8) requestor: invalidate(page, PTable[page#].copy-set)
(9) requestor: PTable[page#].access = write
(10) requestor: PTable[page#].copy-set = empty
(11) requestor: PTable[page#].probOwner = myself
(12) requestor: unlock PTable[page#]
(8) copy holders: receive invalidate
(9) copy holders: lock PTable[page#]
(10) copy holders: PTable[page#].access = null
(11) copy holders: PTable[page#].probOwner = requestor
(12) copy holders: unlock PTable[page#]
An Improvement by Using Fewer Broadcasts

• By periodically broadcasting the real owner, the number of steps to find the owner can be reduced
Distribution of Copy Sets (1)

• So far only the copy-set of the owner is correct and complete. This requires that:
  • the page should be copied from the owner to maintain the copy-set correct
  • invalidation should be broadcasted from the owner to every copy-holders

• Key Ideas
  • copy the page from any of copy holder
  • maintain copy-holders in a tree (with the owner as the root) and propagate invalidation from the owner
Distribution of Copy Sets (2)

Read page fault

1. requestor: lock PTable[page#]
2. requestor: send read access request to PTable[page#].probOwner

3. probOwner: lock PTable[page#]
4. probOwner: forward the request to PTable[page#].probOwner, if it does not have the copy
5. probOwner: unlock PTable[page#]

6. copy-holder: lock PTable[page#]
7. copy-holder: add requestor to PTable[page#].copy-set
8. copy-holder: PTable[page#].access = read
9. owner: unlock PTable[page#]

8. requestor: receive the page from the owner
9. requestor: PTable[page#].probOwner = copy-holder
10. requestor: PTable[page#].access = read
11. requestor: unlock PTable[page#]
Distribution of Copy Sets (3)

Write page fault

(1) requestor: lock PTable[page#]
(2) requestor: send write access request to PTable[page#].probOwner
(3) probOwner: lock PTable[page#]
(4) probOwner: forward the request to PTable[page#].probOwner, if not the owner
(5) probOwner: PTable[page#].probOwner = requestor
(6) probOwner: unlock PTable[page#]
(5) owner: lock PTable[page#]
(6) owner: PTable[page#].access = null
(7) owner: send the page and copy-set to the requestor
(8) owner: PTable[page#].probOwner = requestor
(9) owner: unlock PTable[page#]
(7) requestor: receive the page & copy-set from the owner
(8) requestor: invalidate(page, PTable[page#].copy-set)
(9) requestor: PTable[page#].access = write
(10) requestor: PTable[page#].copy-set = empty
(11) requestor: PTable[page#].probOwner = myself
(12) requestor: unlock PTable[page#]
(8) copy holders: receive invalidate
(9) copy holders: lock PTable[page#]
(10) copy holders: propagate invalidation to its copy-set
(11) copy holders: PTable[page#].access = null
(12) copy holders: PTable[page#].probOwner = requestor
(13) copy holders: PTable[page#].copy-set = empty
(14) copy holders: unlock PTable[page#]
Experiments

• Compare to memory coherence algorithms
  • All algorithms have similar number of page faults. Due to the limitation on the number of processors.
  • The number of forwarding requests was used as a criterion for comparing algorithms. \(\text{Overhead} = \text{Messages} = \text{Requests}\)

Both algorithms need a forwarding request to locate the owner of the page for almost every page fault that occurs on a nonmanager processor.

ProbOwner fields usually give correct hints.

Fig. 11. Forwarding requests.
Conclusion

• This paper studied two general classes of algorithms for solving the memory coherence problem
  • Centralized manager
    • Straightforward and easy to implement
    • Traffic bottleneck at the central manager
  • Distributed Manager
    • Fixed dist. Manager
      • Alleviates the bottleneck
      • On average, still need to spend about two message to locate an owner
    • Dynamic dist. Manager
      • The most desirable overall features

• This paper gives possibility of using a shared virtual memory system to construct a large-scale shared memory multiprocessor system