

Shared Virtual Memory

Arvind Krishnamurthy

Programming Models

- Shared memory model
 - Collection of “threads”
 - Sharing the same address space
 - Reads/writes on shared address space visible to all other threads immediately
- Message passing model
 - Collection of “processes”
 - Each with a separate address space
 - Coordination happens through message sends and receives
 - Message passing requires “cooperating” processes

CS518

Arvind Krishnamurthy

Shared Memory vs. Message Passing Programming

- Shared memory is just like programming with threads

Shared memory program: $x = x + 1$

- Message passing is more convoluted, requires marshalling and message anticipation

Message Passing Program

Processor 1

```
send fetch request
receive into tmp
tmp = tmp + 1
send tmp
```

Processor 2

```
receive fetch request
send x
receive into x
```

CS518

Arvind Krishnamurthy

Comparison

- Message Passing
 - + Controlled execution: no race conditions, protection
 - + Explicit control makes performance issues clear
 - Difficult to program, especially for irregular, dynamic programs
- Shared Address Space
 - + Easier to program, graceful migration path, hence attractive
 - + Works well on moderate-scale tightly-coupled systems
 - Race conditions, synchronization hazards, fault intolerance

CS518

Arvind Krishnamurthy



Parallel Architectures & Prog. Models

- Bus-based shared memory multiprocessors
 - Hardware support for coherent shared memory
 - Can run both shared memory & message passing programs
 - Scalable to 10's of nodes
- Distributed memory machines/clusters of workstations
 - Provides message passing interface
 - Uses message passing
 - Scalable up to 1000 nodes
 - Cheap! Economies of scale: benefit from using off-the-shelf equipment



Distributed Shared Memory

- Radical idea: let us not have the hardware dictate what programming model we can use
- Provide a shared address space abstraction even on clusters
- Various approaches:
 - Operating System Support: Ivy, Treadmarks, Munin
 - Compiler Support: Shasta
 - minimize overhead through compiler analysis
 - object granularity as opposed to byte granularity
 - notions of immutable data, sharing patterns
 - Hardware support: Wisconsin Wind Tunnel, DEC Memory Channel
 - allow fine-grained control, efficient execution



Shared Virtual Memory: Ivy

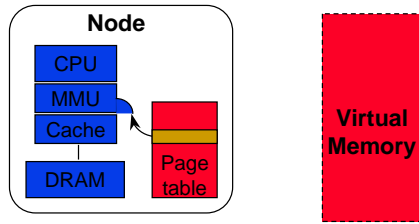
- Seminal system: sparked the entire field of Distributed Shared Memory (DSM)
- Alan Perlis' vision
 - Sharing things on a network
 - An "embassy" system to support a network file system between PDP-10 TOPS-20 and VAX VMS
- Parallel scheme idea
 - Lisp and Scheme use list data structures
- Focus: parallel computing and not distributed computing
 - not interested in request-reply paradigms, fault-tolerance; can assume cooperating programs



Ivy System Overview

- Uses threads ("lightweight process" in the thesis)
- Prototype: Apollo workstations, SVM + thread migration
 - Zoo in 1980's! Network of workstations (100 Apollo's)
- Documented a collection of ideas and conjectures
 - Process migration
 - Page table compaction to minimize memory
 - Paging among memories for space (optimize disk paging)
 - Program transformations to reduce false sharing
 - Distributed state, hints, amortized analysis, ...

Traditional Virtual Memory



- Page Table entry:

Virt. page #	physical page #	valid
--------------	-----------------	-------

- If "valid", translation exists
- If "not valid", traps into the kernel, gets the page, re-executes trapped instruction
- Check is made for every access; TLB serves as a cache for the page table entries

CS518

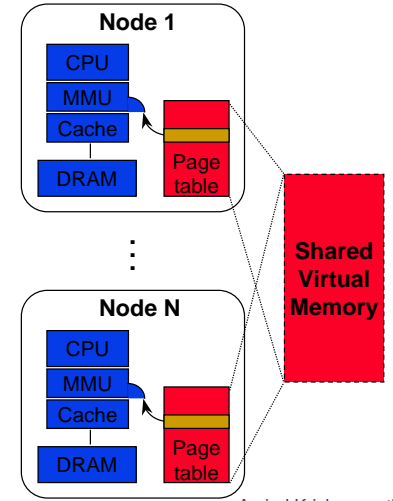
Arvind Krishnamurthy

Shared Virtual Memory

- Pool of "shared pages": if not local, page is not mapped
- Page table entry access bits

Virt. page #	physical page #	valid	access
--------------	-----------------	-------	--------

- H/w detects read access to invalid page
 - read faults
- H/w detects writes to mapped memory with no write access
 - write faults
- OS maintains consistency at VM page level
 - copying data
 - setting access bits

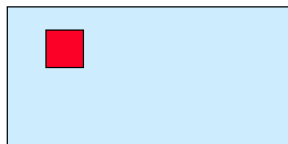


CS518

Arvind Krishnamurthy

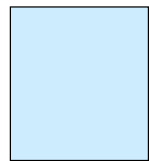
Shared Virtual Memory

Shared Virtual Address Space

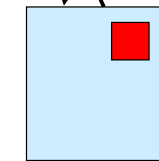


Library keeps track of current "ownership"

Shared Virtual Memory Library



Local Memory



w

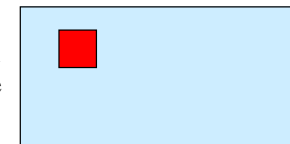
P2

CS518

Arvind Krishnamurthy

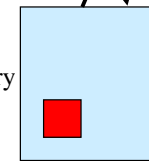
Read Miss to a Page

Shared Virtual Address Space



Shared Virtual Memory Library

Local Memory



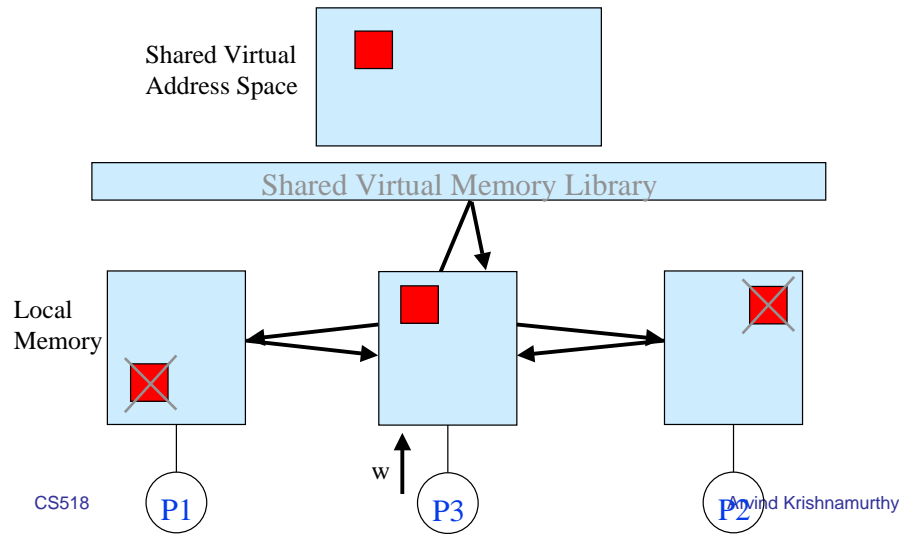
r

P1

CS518

Arvind Krishnamurthy

Write Miss to a Page



Implementation: Base Version

- Single centralized manager (processor) maintains sharing information:
 - Owner(p): who owns p
 - most recent processor to have write access
 - could also be any one of the readers
 - Copyset(p): who has copies of p
- Implementation of copyset(p)
 - Guided by space consumption issues
 - Bit vector
 - Dynamic processor list
 - Bit vector for “neighbors” + dynamic processor list

CS518

Arvind Krishnamurthy

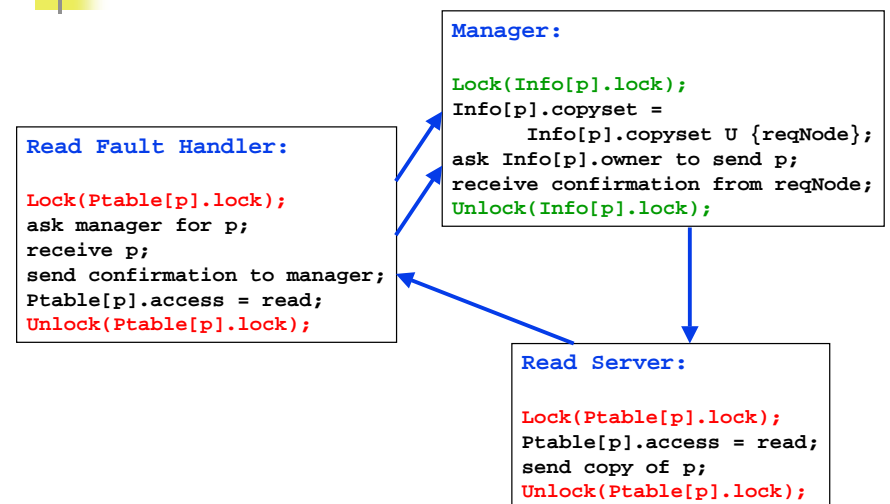
Read fault: High level description

- Handler
 - Ask **manager**
 - **Manager** forwards request to **owner**
 - **Owner** sends the page
 - Requester sends an acknowledgement to the **manager** (so it can update the copyset)
 - 4 messages

CS518

Arvind Krishnamurthy

Centralized Manager Implementation



CS518

Arvind Krishnamurthy

Write Faults: High level description

- Handling includes invalidations:
 - Make request to **manager**
 - Copies are invalidated (hardware drives this issue)
 - Individual messages: simplest, possibly not making use of h/w
 - Broadcast: wasteful communication, unnecessary interruptions
 - Multicast invalidations: middle ground, efficient if sufficient h/w
 - **Manager** forwards request to **owner**
 - **Owner** relinquishes page to **requester**
 - **Requester** sends an acknowledgement to the **owner**
 - Number of messages: 4 + 2*invalidations

Centralized Manager Implementation

Write Fault Handler:

```

Lock(Ptable[p].lock);
ask manager for p;
receive p;
send confirmation to manager;
Ptable[p].access = write;
Unlock(Ptable[p].lock);
    
```

Manager:

```

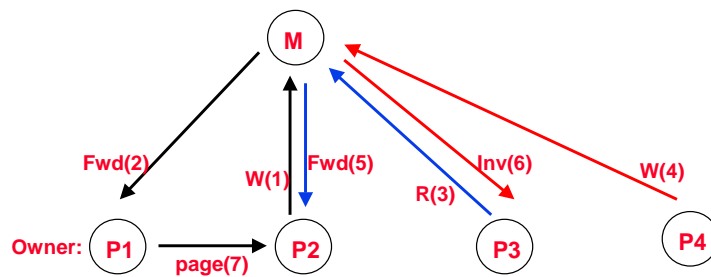
Lock(Info[p].lock);
Invalid(p, Info[p].copyset);
Info[p].copyset = {};
ask Info[p].owner to send p;
receive confirmation from reqNode;
Unlock(Info[p].lock);
    
```

Write Server:

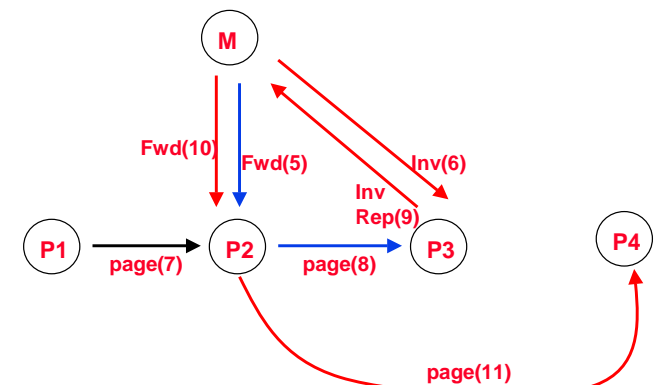
```

Lock(Ptable[p].lock);
Ptable[p].access = nil;
send copy of p;
Unlock(Ptable[p].lock);
    
```

Effect of eliminating confirmation



Effect of eliminating confirmation



Improved Centralized Manager

Read Fault Handler:

```
Lock(Ptable[p].lock);
ask manager for p;
receive p;
Ptable[p].access = read;
Unlock(Ptable[p].lock);
```

Read Server:

```
Lock(Ptable[p].lock);
If I am owner {
    Ptable[p].access = read;
    Ptable[p].copyset =
        Ptable[p].copyset U {reqNode};
    send copy of p;
}
If I am manager {
    Lock(ManagerLock);
    forward request to owner[p];
    Unlock(ManagerLock);
}
Unlock(Ptable[p].lock);
```

Improved Centralized Manager

Write Fault Handler:

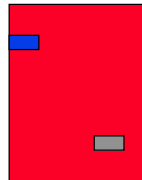
```
Lock(Ptable[p].lock);
If I am manager {
    receive p from owner[p];
} else {
    ask manager for p &
    p's copyset;
    receive p;
}
Invalid(p, Ptable[p].copyset);
Ptable[p].access = write;
Ptable[p].copyset = {};
Unlock(Ptable[p].lock);
```

Write Server:

```
Lock(Ptable[p].lock);
If I am owner {
    Ptable[p].access = nil;
    send p & p's copyset to reqNode;
}
If I am manager {
    Lock(ManagerLock);
    forward request to owner[p];
    owner[p] = reqNode;
    Unlock(ManagerLock);
}
Unlock(Ptable[p].lock);
```

Granularity

- Key leverage point: use existing hardware support
 - TLBs, page fault mechanism
 - tied to page size → granularity
- Advantages
 - Prefetching when spatial locality is high
 - Latency vs. bandwidth argument
- Disadvantages
 - False sharing
 - Fetch unnecessary stuff
- Solution 1: restructure program
- Solution 2: let concurrent updates take place, reconcile later



Issues

- Write update vs. write invalidate
 - Question: can we support a write invalidate system?
- Centralized vs. distributed managers
 - Scalability
 - Load balance
 - Number of messages
- Common goal of any mechanism:
 - Need to know owner(p) and copyset(p)
 - Need to send invalidation messages to members of copyset(p)

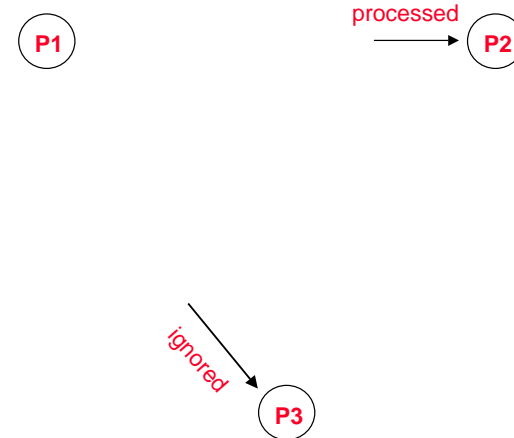
Distributed Manager

- Fixed distribution scheme:
 - Ask manager(p) who is the owner(p)
 - $\text{Manager}(p) = p \% \text{NUM_PROCS}$ or $(p/b) \% \text{NUM_PROCS}$
 - Requires good “load balance”
- Broadcast scheme:
 - Broadcast query for owner(p)
 - Owner(p) responds, rest of the processors remain silent
 - Owner(p) maintains copyset(p)
 - Correctness issue: P1 and P2 making simultaneous broadcasts for a page owned by P3
 - Requests should not slip through the cracks
 - Requests should not be satisfied multiple times

CS518

Arvind Krishnamurthy

Non-Atomic Broadcasts



CS518

Arvind Krishnamurthy

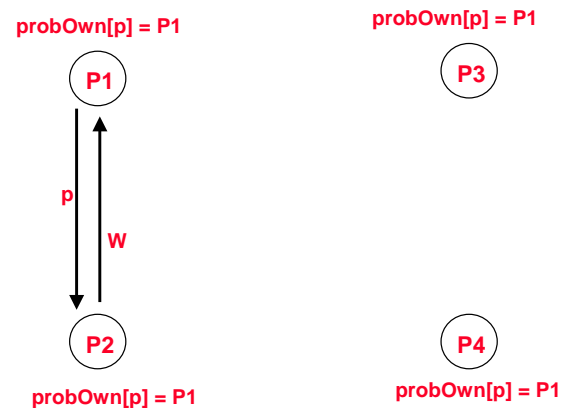
Dynamic non-broadcast scheme

- Try to contact the owner directly by maintaining a **probOwner**
 - Use $\text{probOwner}(p)$ as an hint
 - Every node has the probOwner table
 - Forward request if not owner
 - Update hint if required: for example, if forwarding a write request
 - Cost of finding the owner increases, but avoids bottleneck
- $\text{probOwner}(p)$ gets updated:
 - When a processor owns a page p and another processor makes a read or write request
 - When a page is invalidated
 - When a page request is forwarded
 - No need to change probOwner for read requests

CS518

Arvind Krishnamurthy

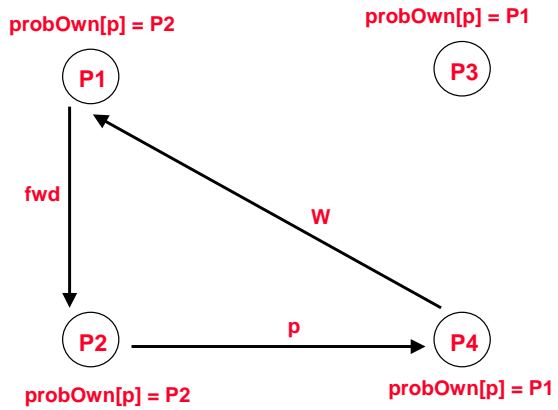
Example



CS518

Arvind Krishnamurthy

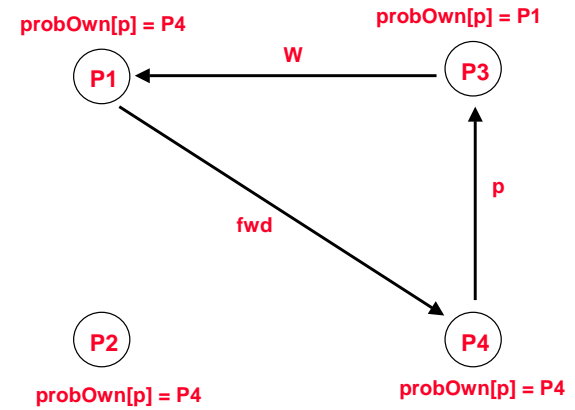
Example (contd.)



CS518

Arvind Krishnamurthy

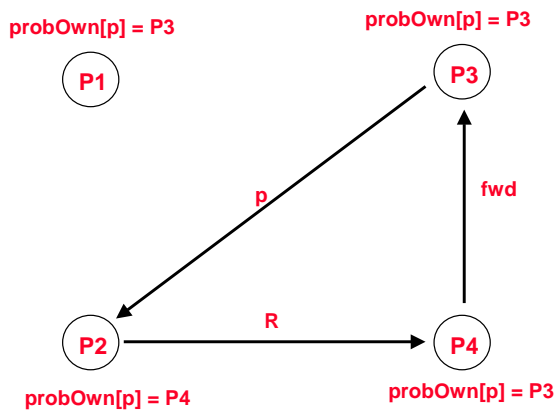
Example (contd.)



CS518

Arvind Krishnamurthy

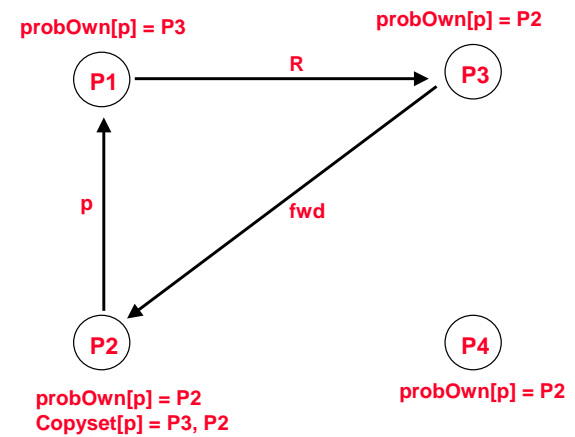
Example (contd.)



CS518

Arvind Krishnamurthy

Example (contd.)



CS518

Arvind Krishnamurthy

Example (contd.)

probOwn[p] = P1
Copyset[p] = P3, P2, P1

P1

probOwn[p] = P1

P3

P2

probOwn[p] = P1

P4

probOwn[p] = P2

Further refinements

- Do not change probOwner field for read requests
- Also, instead of forwarding read requests blindly:
 - Satisfy them locally if processor has the page
 - Implication: owner does not have the entire copyset
 - Copyset(p) = union of copyset(p) values of all processors caching the page
 - Invalidation: tree based invalidation scheme
 - Distributed divide-and-conquer approach
 - Individual nodes synchronize on invalidation replies

Example

probOwn[p] = P2

P1

probOwn[p] = P3

P3

p

fwd

R

P2

probOwn[p] = P4

P4

probOwn[p] = P3

Example (contd.)

probOwn[p] = P2

P1

probOwn[p] = P3
Copyset[p] = P2, P3

P3

R

p

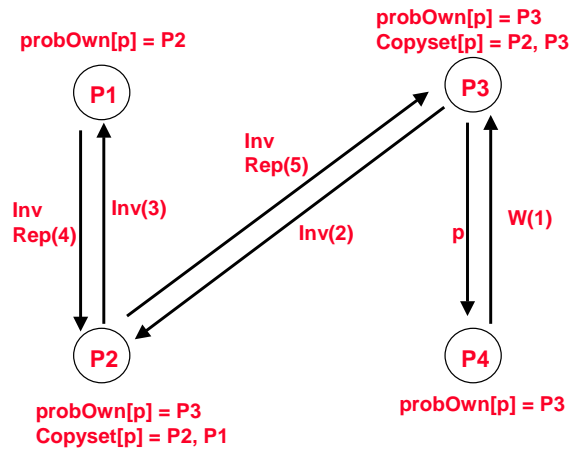
P2

probOwn[p] = P3

P4

probOwn[p] = P3

Example (contd.)



Ideas from the SVM paper

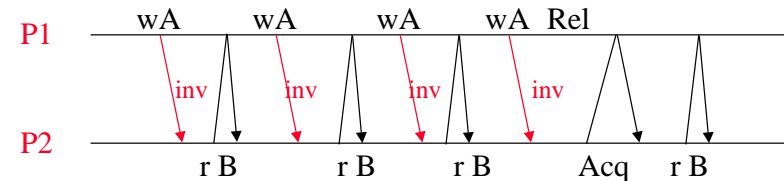
- A classic paper because of the ideas put forth in it
 - Process migration
 - Paging among memories for space (optimize disk paging)
 - Page table compaction to minimize memory
 - Program transformations to reduce false sharing
 - Distributed state, hints, amortized analysis, ...

Where do we go from here?

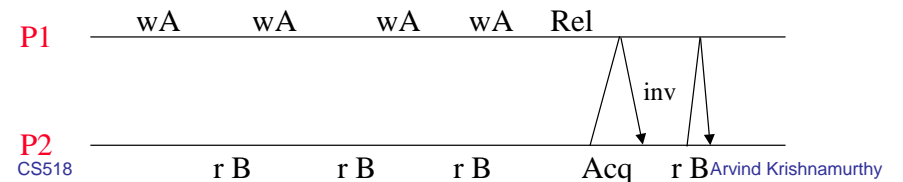
- Performance of shared virtual memory could be bad:
 - Page granularity, false sharing, and thrashing
- Trade programming ease for performance
 - Implicit programming model: sequential consistency
 - Equivalent to a single processor executing instructions from different instruction streams
- Include hardware support
- Build smarter compilers and static analysis tools
 - object granularity as opposed to byte granularity
 - notions of immutable data, sharing patterns
 - minimize overhead through compiler analysis

Alleviating False Sharing Effects

Sequential Consistency



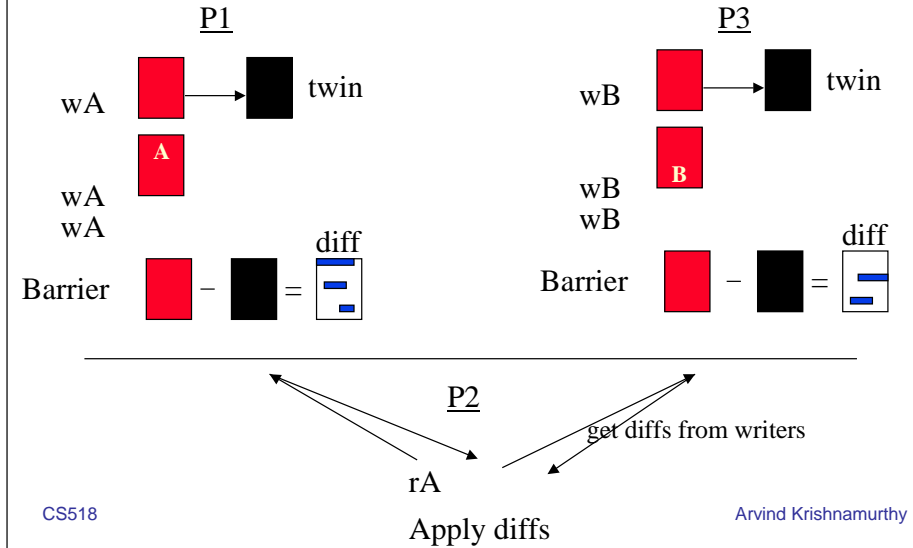
Release Consistency



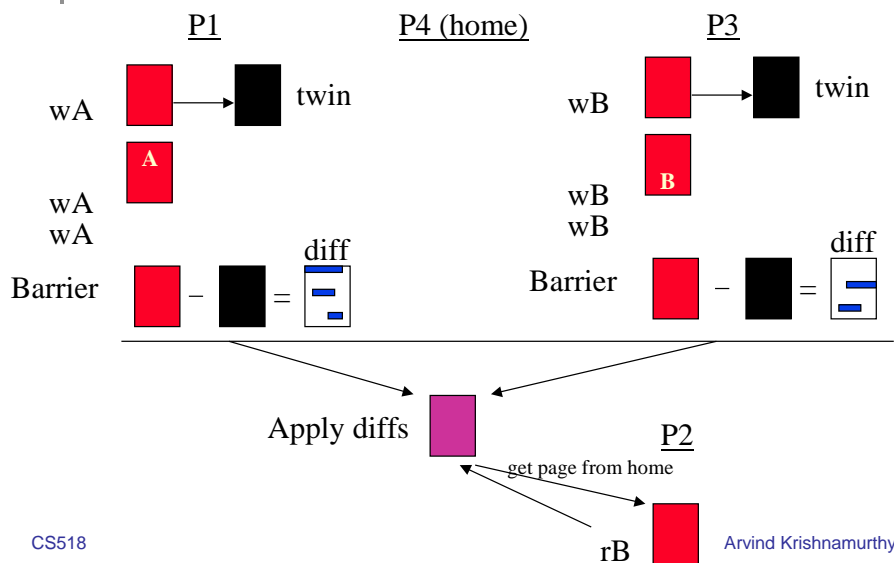
But...

- Increased protocol complexity and overhead
- Increased synchronization cost
- Increased programming complexity compared to sequential consistency

Lazy Release Consistency



Home-based Approach



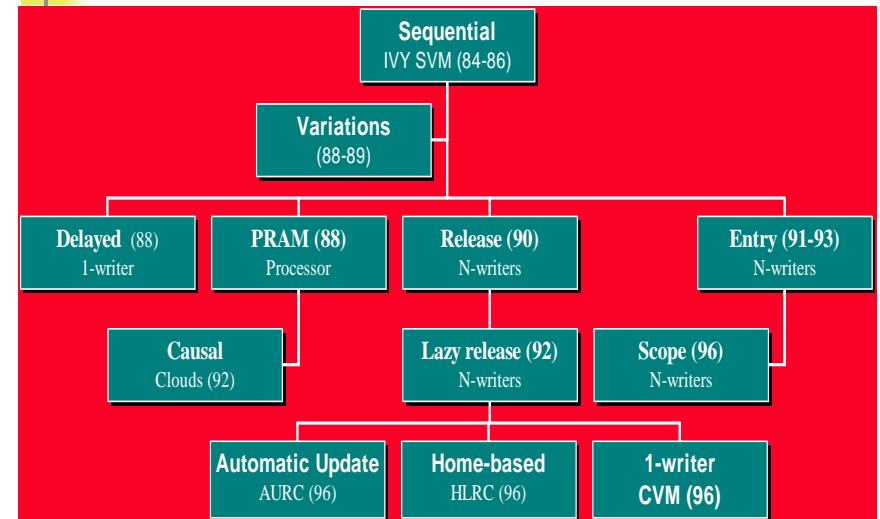
Tradeoffs

- Advantages of home-based
 - Fetch from single location on a page fault: fewer messages in critical path
 - No page faults at the home
 - Less memory overhead for diffs
- Disadvantages of home-based
 - Data distribution is important
 - Potentially higher bandwidth needs

A Host of Choices

- Propagating and Applying Invalidations
 - When: at write, release, acquire or page fault?
 - Eager versus lazy approaches
- Propagating and Applying Changes
 - When: at write, release, acquire or page fault?
 - How: software (diffs, dirty bits), hardware support...
- Mechanisms to Maintain Coherence
 - Directories (information only at home copy)
 - Version numbers (information per page copy)
 - Vector time-stamps (info per page copy per process)

Consistency Models



Summary

- Fun area to do research in
- Not all researchers are informed by the application domain
 - Write parallel programs first, and then build DSM systems
- Did shared virtual memory completely deliver on its promises?
- If not, will it ever?
 - Clusters get further and further apart (relatively speaking)
 - Compilers/languages need to be intimately involved