Software Distributed Shared Memory Multiprocessors

[From Section 9.3 of Culler, Singh, Gupta]

Previous Next

Why SDSM?

- Hardware DSM is hard to design
    - Must have tightly integrated communication assist and NI
    - The CA should probably be custom designed for performance
    - Expensive in terms of time to market and the amount of custom design in memory system
    - But still want to retain shared memory programming
- Software DSM
    - Provides shared virtual memory (SVM) over message passing programs
    - Just take the commodity nodes, connect them over a commodity high-speed network, augment commodity OS with an SVM kernel, and port your shared memory programs to SVM
    - Coherence granularity is a page

SVM for dummy

- Embed a coherence protocol in the page fault handler
    - On a page fault, figure out if the page is mapped on some other node
    - If yes, get a copy of the page and map it in local memory in some free page frame and return from interrupt
    - If no, swap it in from disk and map it as usual
    - If it was a page fault generated by a load, set only read permission in the PTE; subsequent write will generate another access fault and then you invalidate all copies in the system
    - Multiple nodes are allowed to have a virtual page mapped at different physical frames locally; thus the sharing really happens in the virtual address space and physical address space is private

SVM overheads

- Performance factors
    - Every protocol invocation requires an interrupt and context switch
    - Messages are sent through message passing libraries as opposed to specialized NI
    - The entire protocol runs in software; there is no hardware support
    - Even remote requests interrupt local processes and pollute local caches due to protocol processing
    - The granularity of coherence is too big; causes unnecessary communication and false sharing
    - This last point was the major problem when such systems took off; attempts to limit false sharing and communication volume led to numerous innovations in SDSM coherence protocols

Use of RC

- A good place to make use of relaxed models
    - With SC there is no other choice but to invalidate all sharers and wait for all acknowledgments on every write to a page; immediately the invalidated readers may proceed to bring the page back and performance will degrade sharply

- SDSM systems invariably advertise RC or WO or some other relaxed model, but not SC
- Under WO since all accesses between synchronization points can be re-ordered arbitrarily, the writer can hold back all write notices (i.e. invalidations) until that point
- For RC this needs to be done only at release boundaries
- Note how different the use of RC is from hardware DSM; there RC is used to hide write latency and invalidations are sent immediately; here RC is used to limit communication (close to delayed consistency)

Previous Next

Eager and lazy release

- Propagating invalidations at release is still conservative
  - P1 does not care about the writes from P0 until P1 executes the next acquire; at this point P1 must see all updated values
  - Delay write notices until next acquire of the consumer
  - Let the consumer ask for the updates (on demand)
  - This leads to lazy release consistency (LRC); the conventional release consistency is often called eager release consistency (ERC) in SDSM world
  - In LRC a process executing an acquire obtains all write notices corresponding to all releases that happened in the system since its last acquire (conservative)

Lazy release

- All synchronization operations must be carefully labeled

```
P0:                          P1:
LOCK(L);                     while (!ptr);
ptr = some_non_null_value;   LOCK(L);
UNLOCK(L);                   f(ptr);
                             UNLOCK(L);
```

- Hardware DSM binaries may not work directly in SVM
  - The fence instructions are largely useless here
  - What is more important is a way to tell the SVM library to propagate writes at proper points

Multiple writers

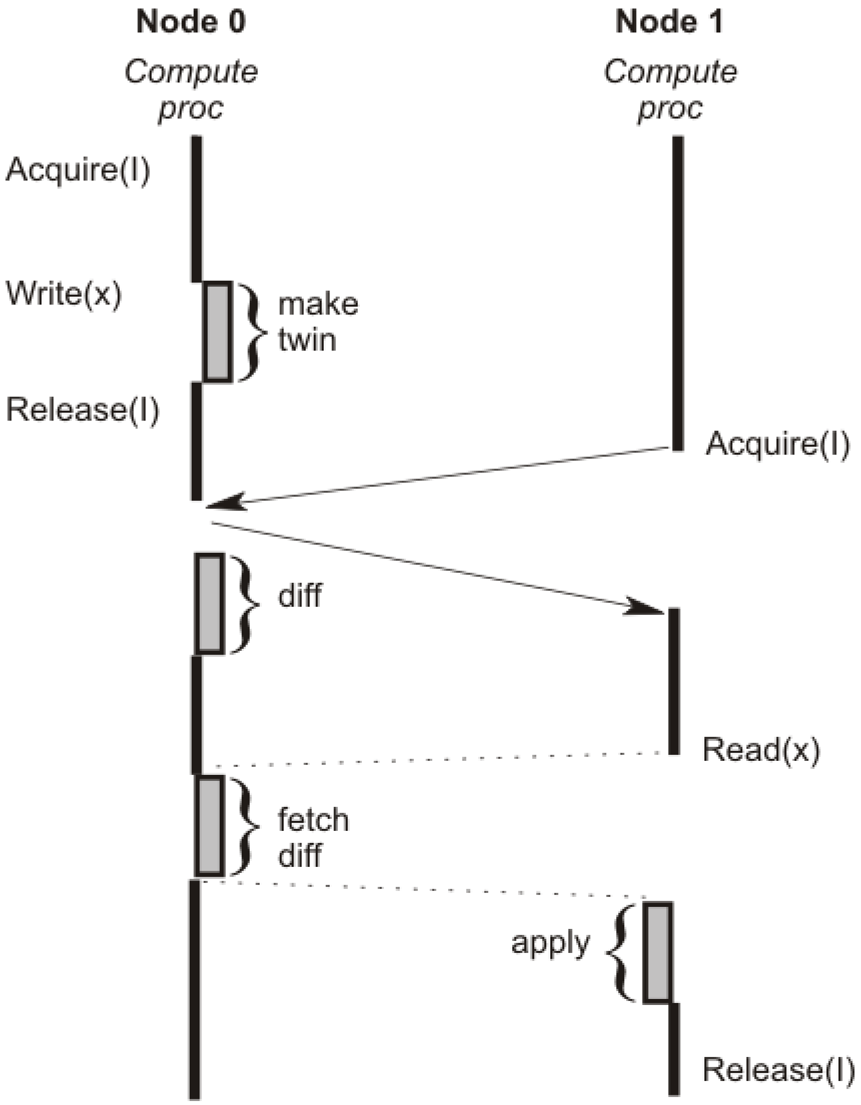
- Thus far we have silently assumed only one writer
  - With multiple writers if the coherence protocol only allows a single modified page at a time, ownership must be transferred every time a new writer arrives
  - Clearly, under release consistency there is no problem in having multiple writers; you just need to pretend as if all the writes from one processor happened before all the writes from another even though they actually interleaved (assume that none of these writes are part of a release)
  - So we just need to design a multiple writer protocol which allows multiple writers to co-exist between two consecutive synchronization points, allows pages to be modified locally and become inconsistent
  - The main design concern of this protocol is: what happens when a process reaches acquire? How to collect all write notices?
- Multiple writer protocol (from TreadMarks SVM)
  - When a page is brought in, the PTE is marked to have only read permission
  - On the first write to the page an access fault handler is invoked and the handler makes a copy of the page (called twin); also at this point the PTE is set to have RW
  - Now the process can write to the page as many times as it wishes
  - At release boundary (for ERC) or at the time of an incoming acquire request (for LRC), the page is compared with the twin and a diff is created (containing just the

modifications)

- Finally, the diff is propagated to the requester
- The requester collects all the diffs and merges them into its own copies

Previous Next

Twin and diff

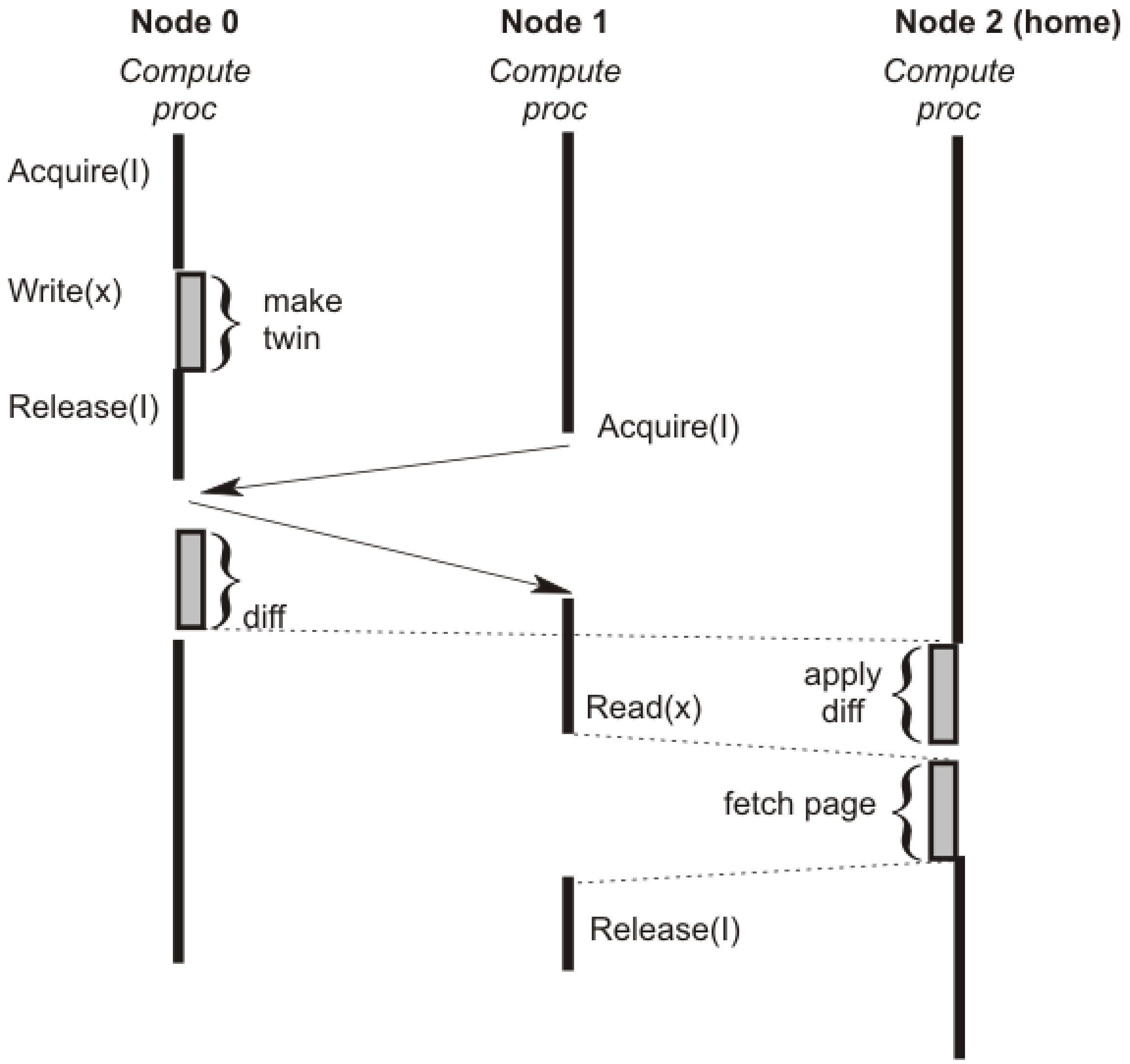


- Fits well with ERC
    - Can free storage of twin at release
    - In LRC must hold back diff until requested, and until it is guaranteed that no process will request in future
    - Garbage collection becomes necessary in LRC forcing diffs to be propagated; this is very complex due to the fact that a page may have distributed diffs across many nodes and all of them must be collected before propagating
    - Solution: home-based LRC (HLRC)
    - Let every page have a home node (may be necessary anyway even in standard LRC to find out writers/readers unless the OS kernel has a shared page table; in a commodity cluster usually each node has a separate OS which is much simpler than a distributed NUMA OS)
    - At release send the diff to the home node which merges the received diffs into the “master copy”

Previous Next

HLRC

- Home-based LRC
    - A process performing acquire obtains write notices from previous releaser
    - But on getting a page fault it asks the home node to send the entire page (of course, with already merged diffs)
    - Note that this protocol not only provides space advantage, but also leads to two-hop page transfer from home to acquirer (as opposed to multiple two hops corresponding to multiple previous writers)
    - Also, home node never suffers from page fault; here also you see a notion of local vs. remote access faults
    - However, here the whole page (as opposed to diffs) is communicated every time from the home leading to wastage of BW



Twin and diff overhead

- Diff processing is expensive
    - Twin creation, diff computation and diff application take up precious CPU cycles (but still faster than network)
    - Some hardware support (especially for fine-grain write propagation) in NI would definitely help
    - Writes can be detected by snooping the memory bus if the caches are write-through
    - Otherwise, every write can be instrumented in software
    - Thus writes can be automatically propagated to home
    - Problem is all writes are propagated wasting BW
    - All-software approach to avoid diff is also proposed: maintain a dirty bit in software per memory block or memory word; dirty bit is cleared at synchronization points and only the words with dirty bits set are propagated to acquirer

Previous Next

Performance factors

- Where does SDSM stand?
    - HLRC and multiple writer protocols do improve performance dramatically
    - But SDSM is still lagging behind its hardware counterpart by a considerable margin
    - The main bottlenecks are: false sharing, cost of protocol processing, time spent in taking page faults i.e. the interrupt overhead
    - As a result, coarse-grain sharing is very well suited
    - Also, synchronization does not scale well on SDSM because all primitives must be implemented with explicit messages
    - Suggestions: hardware support for diff processing in memory controller (e.g., page copy engine)? Dedicated hardware thread for protocol processing and capability to deliver interrupt from a protocol thread to kernel (partitioned contexts?)?

Arbitrary grain

- Why not let the user specify which variables (or formally called “objects”) should be kept coherent
    - To each synchronization point attach the “objects” (nothing to do with OOP) for which write notices must be propagated (leads to “shared object space programs”)
    - If nothing is attached to a synchronization point just fall back to release consistency
    - The big advantage is that false sharing may disappear completely
    - Disadvantages: a careful analysis of the program is needed, an efficient run-time library must intercept all synchronization events and manage the attached objects
    - This is known as entry consistency
    - Same philosophy has been applied to page-based SVM also leading to scope consistency

Implementing ERC

- Single writer
    - Simple scheme: maintain sharer list at the owner and transfer it with ownership to the next writer; at release send write notices to all sharers for all pages that the writer has written to since its last release
    - Problem1: Multiple invalidations to the same node
    - Solution1: Maintain a directory entry per page and store the sharer list there; releaser first consults the directory and then sends invalidations
    - Problem2: Invalidating copies more recent than the releaser’s copy (not a correctness issue, just a performance problem)
    - Solution2: Attach version number to each copy; increment version number on write; receiver applies invalidation only if its version number is lower than releaser’s; is it better with directory?
- Single writer
    - When to collect the invalidation acknowledgments?
    - Conservative: wait for all acknowledgments immediately at release
    - Observation: following the same argument as LRC we can push the time to collect all acknowledgments until the next incoming acquire (the acquire will come to the last releaser because it probably has the dirty page with the synchronization variable)

This optimization allows the releaser to proceed past release while the acknowledgments are collected in background; again without hardware support, collection of each acknowledgment may need an interrupt

    - Under heavy contention the next acquire may immediately follow the release

- Multiple writers
    - Doesn’t make sense to talk about sharing list unless the sharing list is kept coherent across all writers (this may require broadcasting read access faults to all owners)
    - Two ways to communicate write notices: broadcast write notices at release or use a directory to find sharers
    - How does a faulting processor obtain the diffs?
    - Two solutions: use a home node and releaser sends diffs to the home node or visit all “causal” releasers and apply diffs in appropriate order; order of diffs is very hard to decide and therefore, multiple writer ERC systems use updates instead of invalidations if no home (diffs are sent at release and are not demand-based); is the order okay now? Non-deterministic if not race-free
    - Update-based multiple writer ERC protocol is used in Munin
    - What about version numbers? Not helpful

Previous Next

Implementing LRC

- Single writer
    - The invalidations may be sent at any time between release and the next incoming acquire
    - Sending invalidations at acquire time puts the entire inval/ack round-trip time in the critical path of acquire
    - On the receiving side, a processor may or may not choose to apply the invalidations immediately; it is perfectly fine to defer application until next acquire of the processor
    - Note that invalidations are sent to all sharers not just the acquirer (why?)
        - Otherwise, the last releaser not only sends the invalidations for pages it has written to, but also the invalidations it itself has received; optimally acquirer wants to receive invalidations for releaser's writes and others that it hasn't yet seen, but not all
- Single writer
    - Version numbers don't help in limiting invalidation traffic (note the difference with single writer ERC)
    - To reduce unnecessary invalidation traffic every node maintains a vector time stamp
    - The vector has P elements if the system has P nodes
    - The execution on each node is divided into intervals such that every acquire or release starts a new one and each interval has a vector time stamp per process
    - Vector element i of the vector in node k records the last logical interval of node i that sent a notice to node k
    - When a node sends an acquire request to the last releaser it also sends the vector time stamp so that the releaser can compare it with its own and reply with the appropriate invalidations
- Multiple writers
    - Essentially same as single writer as far as invalidations are concerned
    - Diffs introduce new storage overhead
    - HLRC seems to offer the best solution: home node can maintain the diffs (already discussed)
    - Without home, vector time stamps can be used for determining correct diff order
    - Even in the presence of home node, vector time stamps can be used to reduce invalidations as in single writer case
    - One of the main design goal is to keep diff storage low so that large problems can be run

Previous Next