

AN INTRODUCTION TO DISTRIBUTED SHARED MEMORY SYSTEMS

Lecture notes for Fall 1998

1 INTRODUCTION

- *Distributed shared memory* (Kai Li 1986. Kai Li and P. Hudak 1989) is a software abstraction allowing a set of workstations connected by a LAN to share a *single paged virtual address space*.

In other words, a DSM simulates by software the existence of a shared memory among machines that are physically disjoint. It hides all message passing operations from the programmer who can program the workstations as if it were a single multiprocessor. Hence we can have the functional equivalent of a multiprocessor architecture at a much lower price.

- In its essence a DSM is quite similar to a conventional virtual memory system: whenever a process attempts to access data that do not reside in its physical memory, a *trap* occurs and the process remains blocked until the data have been fetched.

The difference is that the data must now be taken away from another node. Hence a fault will often result in another fault. In the worst case, we may have a *Ping-Pong* effect where some data are always in transit between two nodes.

- The key issue in a DSM is that of its *performance*. To achieve it, we will *replicate* all data that are needed on more than one node and develop sophisticated strategies for minimizing the cost of maintaining these data in a *consistent state*.

From this viewpoint, the management of a DSM is very similar to that of managing memory caches in a multiprocessor architecture.

- DSM is not a panacea: remote procedure calls are better suited to distributed applications that conform to the client/server model.

2 DSM AND CACHE MANAGEMENT

- Multiprocessors and multicomputer architectures belong to a continuum of architectures. We can see the same continuum in memory management solutions:

— *Hardware controlled caching:*

MMU transfers cache *blocks*

Single bus multiprocessors: Sequent, Firefly

Switched multiprocessors: Dash, Alewife

— *Software controlled caching:*

(a) *managed by the O.S.:*

NUMA architectures: CM*, Butterfly
(remote access is done by *hardware*)

Page-Based DSM: IVY, Mirage
(NORMA: no hardware remote access)

OS. transfers whole *pages*

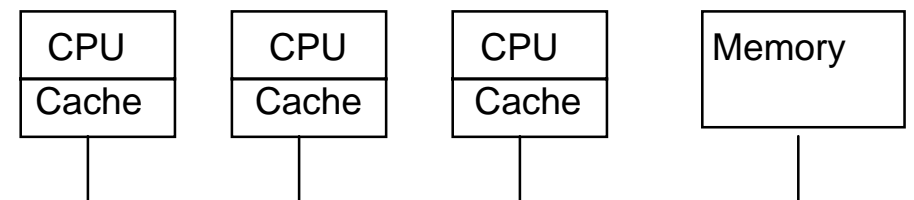
(b) *managed by a run-time system:*

Shared-variable DSM: Munin, Midway
transfers *data structures*

Object-based DSM: Linda. Orca
transfers or objects

- Note that the conventional dividing line between multiprocessor architectures and distributed systems is somewhere between switched multiprocessors and page-base DSM: C_m^* is normally considered to be a distributed system and the Butterfly to be a multiprocessor.

2.1 Cache management in single bus MP's



- Since there is a single bus, the MMU's of all processors can watch the bus and learn which data blocks are fetched or updated (*snoopy caches*).
- The simplest protocol is **write-through** with **cache invalidation**:
 - whenever a processor updates a data entry in its cache, the data block containing the entry is immediately written into the main memory;
 - other caches listen to the bus, notice the block transfer and *invalidate* their copy of the block if they have one.
- A better solution takes advantage of the locality of writes (**ownership protocol**):
 - data blocks fetched in a cache are initially marked **CLEAN**;
 - when a processor updates for the first time a data entry in its cache, it marks the whole data block **DIRTY** and broadcasts the news to let the other caches invalidate

their copy of the data block but it ***does not update the main memory***,

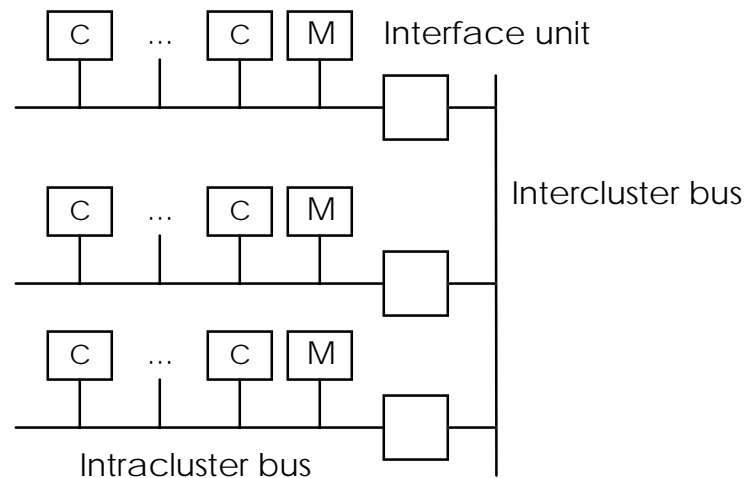
- the processor is now the **owner** of the data block: it can continue to update it without having to notify the other caches; the main memory will not be updated until the data block is flushed from the cache;
 - whenever another processor needs to access the data block, it sends a request to the bus; this request will be intercepted by the cache of the processor owning the data block, which will mark its copy ***INVALID*** and send the block to the cache of the other processor
- The correctness of both *write-through* and *ownership* protocols are based on the fact that all caches do bus snooping
 - Both protocols are implemented in the MMU and take less than a memory cycle.
 - Both protocols prefer to invalidate rather than to update as it takes less bus bandwidth.

2.2 Cache management in switched MP's

- A switched MP is organized as a set of *clusters* containing several processors and memory unit(s) on the same *local bus*; clusters are linked by *intercluster buses* and can be grouped into *superclusters* linked in turn by *supercluster buses*.

The switched MP *Dash* consists of 16 clusters each containing a bus, 4 CPU's, 16 MB of memory and some I/O devices. Each CPU can snoop on its local bus but not on the intercluster busses. The intercluster bus uses wormhole routing to achieve very high speed.

The total Dash address space is $16 \times 16 \text{ MB} = 256 \text{ MB}$.



- Dash cache management policy is a combination of *bus snooping* (within each cluster) and directories (between processors on different clusters).

The caching and data transfer unit is a 16-byte block.

Each cluster has a directory that keeps track of the current ownership of the 2^{20} data blocks within the 16 MB of cluster memory.

Each directory consists of a 16-bit *bitmap*¹ identifying which clusters have a copy of the block and 2 bits representing the state of the block (*UNCACHED*, *CLEAN* or *DIRTY*)

A block is said to be *UNCACHED* if the only copy of the block is in main memory.

It is said to be *CLEAN* if it may be also present in one or more caches but the main memory is still up-to-date.

It is said to be *DIRTY* otherwise.

- Each data block has a home cluster. This home cluster is the cluster whose memory always has a copy of the data block. It

¹ This is not the only possible solution: the Alewife system developed at MIT stores in each directory entry a list with the ID's of the clusters having copies of the block.

can be identified by looking at the first four bits of any address within the data block.

At every moment, each block is also *owned* by a single cluster. Blocks that are in the UNCACHED or CLEAN states are owned by their home clusters; DIRTY blocks are owned by the cluster having the one and only up-to-date copy of the block.

When a CPU wants to access some data that are not in its cache, it issues a request on the intracluster bus:

- a) if the data block is already present in one of the cluster caches, a cache to cache transfer takes place; *moreover, if the block was DIRTY, a copy of it is sent to its home cluster and the block returns to the CLEAN state;*
 - b) if this procedure fail, the directory hardware forwards the request to the home cluster of the data block (which can be or not be the same cluster):
 - if the current block state is UNCACHED or CLEAN, the directory hardware of the home cluster fetches it from the cluster main memory and forwards it to the requesting cluster; the new state of the data block will always be CLEAN;
 - if the current block state is DIRTY, the directory hardware of the home cluster forwards it to the current owner of the block; the directory hardware of that cluster fetches the data block from one of the cluster caches and forwards it to the home and the requesting cluster; as a result the data block becomes CLEAN since the home cluster main memory has been brought up to date.
- Before a CPU can perform a write, it must invalidate all other copies of the affected data block:

- a) if the data block is already **DIRTY** and present in the CPU's cache, the Write can proceed;
- b) if the data block is already **DIRTY** and can be found in one of the cluster caches, a cache to cache transfer will take place and the block copy in the other cache will be invalidated;
- c) if the data block was not in the **DIRTY** state, all other copies must be invalidated first by the home cluster:
 - if the current block state is UNCACHED, the directory hardware of the home cluster fetches it from the cluster main memory and forwards it to the requesting cluster; the new state of the data block will always be DIRTY;
 - if the current block state is CLEAN or DIRTY, the directory hardware of the home cluster must send an invalidation request to all clusters having copies of the block.

2.3 Cache management in NUMA MP's

- Like other multiprocessor architectures, Non-Uniform Memory Access multiprocessors have a single virtual address space that is shared by all CPU's. The difference between NUMA and UMA architectures is that accessing a remote memory address is much slower than accessing a local address, which was not the case for other UMA architectures.
- **Examples:**
 - a) CM*: it used the same two level architecture as the later DASH did but the had a much slower intraccluster bus and microcoded MMU's instead of hardware-based directories
 - b) BBN'S Butterfly

- When a process starts on a NUMA, it may or may not have some of its pages prepositioned in its local memory.

In either cases, any reference to a page that is not mapped into its current address space will cause a **page fault** since remote accesses are controlled by software.

The page fault handler can:

- map the page onto the remote memory where it currently resides: this solution leaves the page where it was located but forces the process to access the page remotely;
- *replicate* or *migrate* the page into the local memory of the process depending whether the page is *read-only* or *read-write*; this solution a higher initial overhead but makes subsequent accesses to the page much faster.

In either cases, subsequent accesses to the page will be handled by the hardware without any further intervention from the OS.

- Selecting to access a page remotely or locally is a difficult decision: most NUMA systems have a **page scanner** that runs in the background, gathers every few seconds page usage statistics and uses these data to rearrange page mappings.

The page scanner can also **freeze** the location of a page for a few seconds whenever it detects a page is been moved too much.

- Numerous algorithms have been proposed but none seems the best for all machine architectures, remote to local access times ratios and patterns of process behavior.

3 CONSISTENCY MODELS

- A consistency model is essentially a **contract** between the software and the distributed memory:
- It contains *rules* that the software must follow in order to be *guaranteed* that the memory will work correctly.
- As it can be expected, the contracts with the least restrictions will not perform as well as those with major restrictions.

3.1 Strict Consistency

- Strict consistency stipulates that:

*Any read to a memory location X will always return the value stored by the **most recent write** operation to X .*

- Strict consistency forces the distributed memory to act as if it was a conventional memory

It is the hardest to implement due to the unavoidable transmission delays: how can we guarantee that a read immediately following a remote write to the same memory address will see the outcome of that write if the time interval between the two events is less than the signal transmission delay without enforcing mutual exclusion?

3.2 Sequential Consistency

- Sequential consistency stipulates that:

*The result of any execution is the same as if the operations of all processors were executed in **some sequential order** and the operations of each individual processor appear in this sequence in the order specified by the software.*

- Sequential consistency forces the all non-local accesses to be serialized possible by a centralized sequencer but allow these operations to be serialized according to the order they arrive

at the sequencer rather than the order they were issued as long as all requests from the same processor remain ordered in the order they were issued.

3.3 Causal Consistency

- Causal consistency weakens sequential consistency by distinguishing between events that are *potentially causally related* (the outcome of one could maybe affect the outcome of another and those that are not (they are said to be concurrent)).
- It stipulates that:
Writes that are potentially causally related must be seen by all processes in the same order. Concurrent writes may be seen in different order on different machines.
- Causal consistency forces the distributed memory to keep track of which processes have seen which writes and building some dependency graph.

3.4 PRAM Consistency and Processor Consistency

- PRAM (for Pipelined RAM) consistency stipulates that:
Writes done by single process must be seen in the order they were issued but writes from different processes may be seen in different order on different machines.
- Strict consistency forces the distributed memory to act as if it was a conventional memory
- Processor consistency is equivalent to PRAM consistency but also requires a *global agreement* among the processors on the order of writes to any *specific memory location X*.

3.5 Weak Consistency

- Good programming practice dictates that most accesses to shared variables will be done through critical sections. Since critical sections enforce mutual exclusion the exact order of execution of write operations by one process within a critical section is of no interest to the other processes. We can therefore delay the writes until the end of the critical section.
- Weak consistency introduces a new type of shared variables called ***synchronization variables***. Any access to a synchronization variable forces all previous writes to complete everywhere:
 1. *Accesses to synchronization variables are sequentially consistent* (that is all processors see them in the same order).
 2. *No access to a synchronization variable is allowed to be performed until all previous writes have completed everywhere.*
 3. *No read or write is allowed to proceed until all previous accesses to synchronization variables have completed.*
- As a consequence, every process that accesses a synchronization variable ***before*** accessing any shared data is guaranteed to get the most recent values of these data.
- Similarly, accessing a synchronization variable ***after*** having one or more write operations guarantees that all other processors will receive the newly modified values of the shared variables.

3.6 Release Consistency

- Weak consistency is still too restrictive because it forces all writes to complete every time a synchronization variable is accessed.

- Release consistency goes one step further in trying to implement these minimal requirements for the correct implementation of critical section semantics:
 - When a process enters a critical section, it should always get the most recent values of all shared variables in that critical section.
 - When a process leaves a critical section, it should let it known that new values of the shared variables in that critical section are now available.
- In other words:
 1. *Any access to a shared variable must wait the completion of all previous acquires issued by the process.*
 2. *Any acquire must wait the completion of all previous read and writes issued by the process.*
 3. *Acquire and release must be processor consistent.*
- Release consistency is the most efficient consistency model. The price to pay is that programmers must bracket their accesses to the shared data by library procedures calls such as *enter_critical_region()* and *leave_critical_region()* or *request()* and *release()* unless they are not concerned about maintained the accessed data in a consistent state.
- Release consistency also provides a global mechanism for synchronizing processors:
 - Whenever a process executes a *barrier*, it must wait until all processes have reached that barrier.
 - At that moment all pending writes are propagated everywhere and the processes can proceed having all received the most recent values of all shared data.

- There are two possible semantics for the *release* operation:
 - *Eager release* immediately forwards the modified data to all the other processors.
 - *Lazy release* waits until the data are requested by another processor.

3.7 Entry Consistency

- Entry consistency is a variant of release consistency that requires each ordinary shared variable to be associated with a *synchronization variable*, say, a lock or a barrier. When an *acquire* is performed, the shared variables guarded by the synchronization variable that was accessed are the only ones to be synchronized.
- As a result, entry consistency is much cheaper than *release consistency*, which requires the whole shared address space to be synchronized. The price to pay is the added complexity of specifying the shared variables guarded by each synchronization variable and the added risk of programming errors.
- Entry consistency allow to request exclusive access or non-exclusive access to the guarded variables.
- The semantics of entry consistency can be specified as follows:
 1. *An acquire access to a synchronization variable must wait for the completion of all previous updates to the shared data guarded by that synchronization variable.*
 2. *An exclusive mode access to a synchronization variable cannot proceed unless no other process holds the synchronization variable either in exclusive or in non-exclusive mode.*

3. *After an exclusive mode access to a synchronization variable has been performed by a process, any non-exclusive mode access to the synchronization variable will not be allowed to be performed until it has been performed with respect to the process holding the synchronization variable.*

3.8 Summary

- One can distinguish two classes of consistency models:

1. those using synchronization operations:

these are weak consistency, release consistency and entry consistency; we have:

weak > release > entry

2. those not using synchronization operations:

these are strict consistency, sequential consistency, causal consistency, processor consistency and PRAM consistency; we have

strict > sequential > processor > PRAM

4 PAGE-BASED DSM

- Page-based DSM implements a shared address space on the top of a NORMA (that is *No Remote Access Architecture*) system which can consist of a cluster of workstations.
- The original objective was to allow programs written for a multiprocessor architecture to run on a cluster of workstations without modifications: these programs are often referred to as *dusty decks*.
- For this reason, the first DSM systems implemented a sequentially consistent memory; it was later found that this solution was not particularly efficient.

4.1 Classical DSM Systems

- Their main objective was to allow programs written for multiprocessor systems to run *unchanged on a multicomputer architecture*
- Since these programs normally assume *sequential consistency*, systems like IVY or Mirage implemented the *same* model of data consistency.
- The general idea is simple: the shared memory is divided into fixed-size pages. At any given time, each page resides on one of the processors. If another processor attempts to access a non-local page, a *page fault* will occur and the missing page will be migrated to the processor's memory.
- A first possible optimization is to *replicate* missing pages instead of migrating them. **Replication** performs better than migration because it allows several processors to share read access to the same data. Two replication strategies are possible:
 - (a) replicate **read-only** pages such as those containing program code, read-only constants and so forth; the benefits of replication might be quite limited but this is the *easiest strategy to implement*;
 - (b) replicate **all** pages; problems may arise every time a processor writes into its copy of a replicated page; the DSM must have a mechanism to prevent inconsistencies among the replicas.
- The lowest feasible level of granularity of a page-based DSM is a single page. One could however prefer to transfer larger data units, say clusters of 2, 4 or 8 contiguous pages.
 - On a typical LAN, it does not take much more to transfer one kilobyte than 512 bytes and we might transfer data that will be used later.

- Larger data transfer units also mean more *false sharing*, that is two unrelated variables that are located in the same transfer unit and are accessed at the same time.
- Smart compilers can reduce false sharing but cannot eliminate it.

4.2 Implementing sequential consistency

- This issue only applies to DSMs that share read/write pages
- Whenever one process modifies its copy of a replicated page, the DSM can either
 - (a) *update* all other replicas of the page, or
 - (b) *invalidate* them
- Page-based DSM typically use an invalidation strategy as it is simpler to implement.

One possible solution consists of associating with each page:

- (a) an *owner*, that is, the last processor that wrote to the page
- (b) a state that can be either R (for *read-only* permission) or RW (for *read/write* permission)

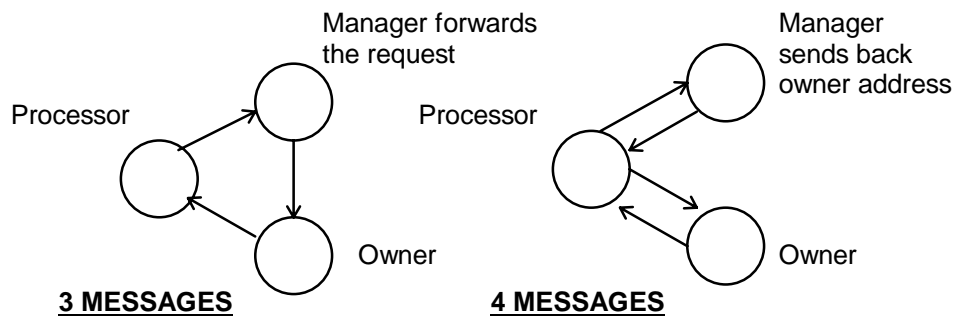
Only pages in read-only states can be replicated:

- (a) whenever a processor wants to write into a page, it must obtain a replica of the page and change its state to read/write, which will automatically *invalidate* all other replicas of the page;
- (b) whenever a processor has a replica of a page with read/write permission and another processor wants to read the page, a copy of the page will be sent to the second process and the state of the page will be changed to read-only;

- (c) whenever a processor has a replica of a page with read/write permission and another process wants to write into the page, the page will be taken away from the first processor and migrated to the second.

4.3 Locating the owner of a page:

- The simplest solution is to broadcast the request message to all processors but every broadcast will result in two context switches on every processors.
- Li and Hudak have proposed several strategies:
 - (a) have one **page manager** that would keep track of the ownership of every page: requesting a page would take between three and four messages, that is, three messages if the page manager forwards the request to the page owner and four messages otherwise;



- (b) have several page managers, each sharing a given number of pages (as in DASH)
- (c) let each processor keep track of the *probable owner* of each page: if the probable owner is not the owner it will forward the request to the site it believes to be the owner of the page

⇒ the search may take several steps

4.4 Locating the copies of a page

- The copies of a page must be invalidated every time the state of a page is changed from read-only to read-write
- The simplest solution is to broadcast an invalidation message with the *page number* to all processors: it only works if we have a *reliable broadcast* mechanism
- The *owner* or the *page manager* could also maintain a copyset listing the processors that currently hold a replica.

4.5 Interactions with the virtual memory system

- The virtual memory manager should be notified of page invalidations so that it knows these pages can be expelled
- Pages that are read-shared with other processors are *clean*: they can be expelled from main-memory without being written back to disk and the virtual memory manager may want to expel them first

4.6 Fighting false sharing

- Mirage guarantees that any page that has been migrated to a processor will stay at least for a given quantum of time ΔT before being released to another processor.
- If other machines want to access it before the quantum expires, their requests are queued for the duration of the quantum

4.7 Spinlocks

- Any processor doing a *busy wait* on a lock variable will compete for the page holding the lock variable with the process holding the lock and *all* other processors waiting on the lock
- A better solution is to use a *lock manager* and a message-based synchronization mechanism.

4 SHARED-VARIABLE DSM

- Sharing specific variables or data structures will reduce false sharing.
- While page-based DSM were always using an invalidation algorithm for managing replicated pages, update algorithms can also be used in shared-variable DSM provided that
 - (a) writes to the individual shared variables can be isolated, and
 - (b) the processor issuing the update is not likely to continue to update the variable.

4.1 Munin

- Munin is based on software objects (variables) but uses the processor's virtual memory to detect access to the shared objects.
- Munin offers eager release consistency and allow the users to annotate their shared variables and, by doing so, inform Munin of their expected behavior.
- Shared variables that are not annotated are treated as in a conventional DSM:
 - (a) read-only pages are *replicated*
 - (b) read/write pages are *migrated* but *never replicated*.
- The price to pay for not annotating pages is a much slower performance of the DSM.
- Incorrect annotations may result in inefficient performance or in *runtime errors* that will be detected by the *runtime system* but ***not in incorrect results***.

- Munin includes:
 - (a) a *preprocessor* that reads the user annotations and converts them into compiler directives controlling the placement of the shared data in virtual memory
 - (b) a *run-time* system that maintains an object directory and a delayed update queue.
- ***Release consistency:***
 - Munin is based on a software implementation of *release consistency*
 - It distinguishes three kinds of shared variables:
 1. ***ordinary variables:*** they are not shared and can only be accessed by the process that created them;
 2. ***shared data variables:*** they are visible to all Munin processes and will appear *sequentially consistent* as long as they are always accessed *from within critical regions*;
 3. ***synchronization variables:*** these can be locks, barriers or condition variables; they must be accessed through special library procedures such as `lock()` and `unlock()` for locks.
 - When a processor modifies shared data inside a critical region, all update messages are *buffered* and delayed until the processor leaves the critical region.
 - This consistency model is known as *eager release* because the update messages are forwarded at release time to all the other processors.

- ***Multiple consistency protocols:***

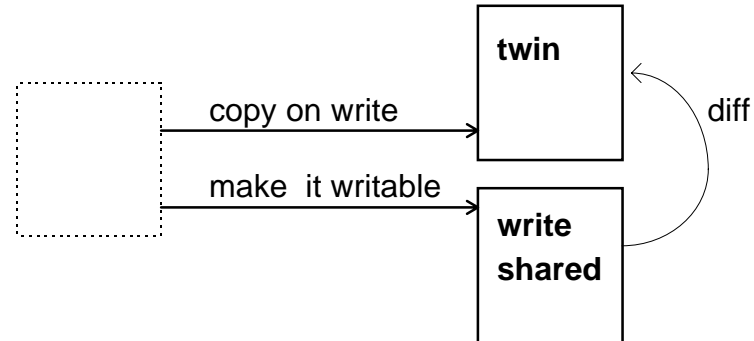
- Munin offers several consistency protocols aimed at supporting different types of access patterns:

4. ***conventional shared variables:*** they are *replicated on demand* and kept consistent using an *invalidation-based protocol* requiring a writer to be the sole owner of the data it wants to modify;
5. ***read-only variables:*** they cannot be modified once they have been initialized and are *replicated on demand*; any attempt to modify them will result in a runtime error;
6. ***migratory variables:*** they are migrated among the processors accessing them: every process accessing a migratory variable will always get full read and write access even if it only requested read access;
7. ***write-shared variables:*** they can be updated concurrently without intervening synchronization because the programmer knows that different portions of the data are accessed

- Since Munin relies on the virtual memory to control all accesses to the DSM, the implementation of its write-shared protocol is based on a *copy-on-write* mechanism:

Whenever a processor is granted access to write-shared data, the page containing these data is marked *copy-on-write*. Hence the first attempt to modify the contents of the page will result in the creation of a copy of the page modified (the *twin*).

At release time, the DSM will perform a word by word comparison of the page and its twin, store the diff in the space used by the twin pages and notify all processors having a copy of the shared data of the update.



— A runtime switch can be set to check for conflicting updates to write-shared data.

- ***Update Time-out Mechanism:***

— To reduce the cost of sending update messages, Munin does not send updates to processors holding *stale* replicas.

— Anytime a processor receives an update for a page for which it does not have a *twin*, the page is marked supervisor only and the time of receipt of the update is recorded.

The first local access to the page will cause a trap that will remove the restriction.

When a processor receives an update for a page that is still marked supervisor only, it checks the timestamp of the previous updates. If more than 50 ms have elapsed it notifies the originator of the update not to send more updates and invalidates the page.

- ***Conclusion:***

The strong point of Munin is its excellent performance, typically within 5 to 33% of that of message passing versions of the same programs. Unfortunately these results can only be achieved after the programmer has painstakingly annotated his or her parallel program.

4.2 Midway

- Midway differs from Munin in two basic ways:
 - (a) it implements a weaker consistency model, namely *entry consistency*;
 - (b) it includes its own compiler, which allows it to detect directly all accesses to the shared data.
- Midway implements three consistency models, among which the programmer can chose:
 - *processor consistency*, a variant of sequential consistency where writes from different processors are not guaranteed to be observed in the same order everywhere,
 - *release consistency*, but without the multiple implementations found in Munin, and
 - *entry consistency*.
- Midway does not attempt to detect inconsistent accesses to the shared data, which may introduce ***incorrect results***.
- ***Entry consistency***.
 - Entry consistency requires all shared variables to be:
 - (a) declared with the keyword *shared*;
 - (b) associated with at least *one specific synchronization object* (lock, barrier, and so forth); this association is dynamic and may change during the execution of a program;
 - (c) only accessed *within critical sections*.
 - Midway uses a *lazy release* protocol: the shared variables associated with a synchronization object are only brought up to date when a critical section for that object is entered.

The values of the other shared variables are left unchanged.

This is clearly a minimalist approach: Midway only updates the variables that the programmer requested and does it “just in time.”

- Unlike Munin, Midway does not rely on the virtual memory to detect which data have been updated. The Midway compiler produces special code that detects which individual variables have been modified.

The major advantage of this approach is its finer granularity: the virtual memory can only detect which pages have been modified (which explains why Munin has to rely on twin pages).

- To minimize message traffic a processor entering a critical section will only receive the updates for all data that were modified since its last critical section.

The times of the updates are kept consistent among all processors through a *logical clock protocol*.

4.3 Treadmarks

- Treadmarks was developed by the group Munin who had . It keeps the same general philosophy (variable-orientation, release consistency and shared write protocol):

- (a) It is entirely implemented at user-level and does not require any changes tot the kernel.

This does not mean that Treadmarks does not use the kernel facilities for detecting: it only means that it does not require any kernel modifications as Munin did.

- (b) It uses lazy release instead of eager release.
- (c) It runs on several commonly available UNIX systems.

Threadmarks was also designed to run on the top of commonly available local area networks, including the rather slow 10-Mbps Ethernet. Hence, it does more than other systems to minimize communication costs

- ***Lazy Release Consistency:***

- Under lazy release consistency, the propagation of updates to other processes is delayed until these processors do an acquire. At that time each process must first find out which updates it needs to see, that is, which updates were made before the last release of each process.
- To achieve that goal, Treadmarks divides the execution of each process into *intervals*, each having its own interval index. New intervals start every time a process does an *acquire* or a *release*.
- Intervals on the same processor are *totally ordered* while intervals on different processors are only partially ordered: We can only say that interval i on processor p will happen before interval j on processor p if j starts with an acquire corresponding to the release that ended i .
- Treadmarks represents this partial order by a *vector timestamp* that has one entry for each processor: entry P in the timestamp vector of processor p will contain its own interval index i while the entries corresponding to *other* processors, say $Q \neq P$ will contain the most recent interval of Q that happened before i .
- Release consistency requires that a processor p performing an acquire must receive “the updates of all intervals with a smaller timestamp.” To achieve this P will send its vector timestamp to each previous processor Q that has done a previous release. When Q receives this message, it sends back to P update notices for all intervals in its current timestamp that were not in the time stamp it received from P .

We could think of interval indexes as logical times. The vector timestamp of a process will contain its current time and the times of the latest updates it received from the other processors. The message that P sends to Q essentially asks to Q if Q did modify any variables since the last time they communicated (at time t_Q). Q replies by sending write notices for all updates after t_Q and informs P that its current logical time is t_Q' .

— Note that write notices do not contain the new values of the modified variables: Treadmarks uses an invalidation protocol and will only update each variable when an attempt to access it causes an address exception.

- ***Multiple Writer Protocol:***

— Like Munin, Treadmarks includes a multiple writer protocol. It uses the same twin page mechanism as Munin with the only difference that *diffs* are only created:

- a) when another processor requests a modification to the page, and
- b) when a write notice for that page arrives from another processor.

— This technique is known as *lazy diff creation*.

- ***Implementation:***

— Treadmarks cheats somewhat with its interval mechanism: new intervals are only created when a lock is released to another processor.

— Each lock as a single statically assigned manager. When a processor P want to acquire a lock, it sends its vector timestamp to the processor Q that either owns the lock or did the last release on it. Q replies by sending its current timestamp and pointers to all write notices in the interval

between the vector timestamp of P and its vector timestamp. When P receives this message, it updates its vector timestamp and invalidates all the pager for which it has received write notices.

- All *barriers* have a centralized manager. When a processor reaches a barrier, it looks for the last timestamp vector of the barrier manager and sends to the manager its current timestamp and pointers to all write notices in the interval between the vector timestamp of the manager and its vector timestamp. When all processes have reached the barrier, they all incorporate these write notices.
 - Garbage collection is used to reclaim the space used by write notices. Garbage collection is implemented through a barrier-like mechanism in which all processors incorporate the write notices before they can be deleted.
 - Invalid and write-shared pages are marked with `mprotect ()` system call. Hence any restricted access to the marked page will result in a `SIGSEGV` signal that will be caught by a signal handling routine in the process.
- ***Performance:***
 - Treadmarks generally achieves good speedups with the benchmark tasks.
 - Most of the OS overhead is communication related overhead.
 - A kernel implementation of memory management would have little effect on performance since these tasks only account for 2.2% of the total run time.

- The authors built a modified version of Treadmarks using an *eager* version of release consistency and compared its performance with that of Treadmarks: they found that eager release performed worse than lazy release on all but one of the benchmark programs. That benchmark program had a read access to a global variable that was not bracketed with an acquire/release pair. Eager release consistency guaranteed that the program was always reading the current value of the shared variable while lazy release consistency let it access an older value that resulted in redundant work. Bracketing the read access with an acquire/release pair would have deteriorated further the performance given the large number of times it was executed.

4.4 C Region Library (CRL)

- CRL is an *all-software* distributed shared memory system that requires no special compiler, hardware, or operating system support. Unlike Munin and Treadmarks, it does not use the virtual memory hardware to detect access to invalid data or write accesses to read-shared data.
- CRL runs on the Thinking Machines CM-5 multicomputer and the MIT Alewife machine.
- **Programming Model:**
 - In CRL processes interact through operations on *regions*.
 - Each region is an arbitrarily sized, contiguous area of memory identified by a unique *region identifier*.
 - New regions can be created dynamically by calling `rgn_create(nbytes)`, which creates a region of size `nbytes` and returns a *region identifier*.

- Before accessing a region, a processor must map it into the local address space using `rgn_map(region_id)`, which returns a pointer to the base of the region's data area.
 - A `rgn_unmap()` function can be used to indicate that the processor is done accessing the region,
 - Since successive mappings on the same processor may place the region at different locations in the local address space, applications that need to store a “pointer” to shared data, should always store the corresponding region identifier.
 - CRL requires that programmers group accesses to a region's data area into operations delimited by calls to CRL library functions
 - Read operations must delimited by a `rgn_start_read()/rgn_end_read()` pair and write operations by a `rgn_start_write()/rgn_end_write()` pair.
 - All write operations are serialized with respect to all other operations on the same region.
 - Newly initiated operations that conflict with those already in progress on the same region in question will wait until they can proceed without conflict.
 - CRL provides a flush call that causes the local copy of a region to be flushed back to the home node.
- ***Consistency Model:***
 - CRL considers entire operations on regions as atomic units and provides sequential consistency for these operations.

- Its memory consistency model is thus similar to *entry* or *release consistency*.
- ***Implementations:***
 - Several CRL implementation are available on the WWW.
 - CRL can also be compiled for use with PVM on a network of Sun workstations
- ***Experimenting with the Alewife Implementation:***
 - To estimate the overhead of performing all DSM operations in software, the authors compared the performance of CRL running on the top of Alewife and Alewife's native shared memory system for the three applications.
 - They found that given high-performance communication mechanisms, CRL is capable of delivering performance *competitive* within 10 to 20% with that provided by hardware-supported DSMs.
 - They concluded that the communication performance of the CM-5 current-generation network of workstation technology is not sufficient to allow CRL to compete with hardware-supported DSMs.

4.5 Shasta

- Unlike other variable-based DSM's, Shasta allows *transparent execution of binaries*. It achieves this objective by inserting before every instruction accessing a shared variable *in-line checking code* that checks if the access can be performed on a local copy of the data.
- The approach has two major advantages:

- a) Shasta can run commercially available programs, such as an Oracle database management system, without having any access to the source code and
- b) Shasta does not depend on the virtual memory hardware to detect accesses to missing or invalid data and can thus use a much smaller granularity than the system's page size.
- One of the problems the designers of Shasta had to solve is *race conditions* that could happen while executing the in-line checking code.
- ***Organization of Shared Data:***
 - Shared data can be in three basic states at each processor, namely, *invalid*, *shared* (meaning that there are other valid copies of the data) and *exclusive*. A *shared miss* is said to have happened every time a processor attempts to read data that are in the invalid state or to write data that are in either the invalid or the shared state.
 - Shasta divides the shared data into ranges of memory addresses called *blocks* and assumes that all data in the same block are always in the same state. Since block sizes can differ within the same application, Shasta divides all blocks into smaller fixed-size entities called *lines* and maintains state information at the line level.
 - Coherence is managed using a *directory-based invalidation protocol*. Each block has a *home* processor keeping track of the state of the block and the location of its copies.
- ***The In-Line Checking Code:***
 - In-line checks require an average of seven instructions and result in an average overhead of 20%.

- Invalid data are always overwritten with a particular bit pattern. This greatly simplifies checks for read access even though Shasta must also perform a secondary check for the rare case when the bit pattern could be valid application data.
- ***Handling Race Conditions in In-Line Checking Code:***
 - Shasta uses two mechanisms to avoid *race conditions* during execution of the in-line checking code:
 - A first general mechanism is to add synchronization primitives but it results in a large increase in the checking overhead.
 - When Shasta executes on a SMP architecture, it takes advantage of the possibility that each process has to read the state tables of each other process located on the same *node*. This allows each process to send explicit downgrade messages all the processors having valid copies of the block instead of sending the message to all processes having a copy of the block.
- Shasta also offers many features that we will not discuss here such as fully supporting of the DEC Alpha instruction set and *Load Locked* and *Store Conditional* instructions.

5 OBJECT-BASED DSM

- Object based DSMs store objects on which specific access methods are defined.
- All accesses to the shared objects are made by invoking methods on the shared objects.
- The approach has two major advantages:
 - (a) most instances of false sharing disappear

- (b) the implementation can take advantage of the fact that accesses to the shared objects are strictly controlled.

5.1 Linda

- Linda implements immutable shared variables that can be only accessed through a small set of primitive operations that can be added to many existing languages, among which FORTRAN and C.
- The only data recognized by Linda is the *tuple*.

A tuple consists of one or more fields, each of which is a value of one of the basic types supported by the host language:

(999554999, "Doe, Jane", jdoe@cs.uh.edu, 90)

- All tuples stored by Linda are visible to all processes on all machines participating to a Linda system; they are said to form a *tuple space*.
- The three basic primitives defined on tuples are:

- (a) *out(<tuple>*, which stores a new tuple in the tuple space as in:

out(999554999, "Doe, Jane", "jdoe@cs.uh.edu", 90)

- (b) *in(<tuple pattern>*), which removes one matching tuple from the tuple space as in:

in(?ssn, "Doe, Jane", email, grade)

where ?ssn matches any field having the same type as the variable *ssn*.

- (c) *read(<tuple pattern>*), which reads the matching tuple without removing it from the tuple space as in:

read(?ssn, "Doe, Jane", email, grade)

- Both `in(...)` and `read(...)` primitives are *blocking*: if no matching tuples are present, they will wait until some other process creates that tuple.
- ***Implementation issues:***
 - To implement efficiently searches in the tuple space, one can subdivide it into disjoint *subspaces* and use *hash tables*
 - The tuple space can be centralized or decentralized: in the latter case, we can either broadcast all `out(...)` operations (and update immediately all replicas or perform the `out(...)` operations locally and broadcast instead the `in(...)` and the `read(...)` operations. More sophisticated solutions such as partial replication are also possible.

5.1 Orca

- Orca follows much more closely the conventional OO paradigm.
- Orca *objects* are abstract data objects similar to Java classes.

An Orca object encapsulates an internal data representation and a list of operations or methods (again like in Java). Each method consists of a list of pairs:

(<guard>,<block of statements>)

as in:

```

Object implementation semaphore
value: integer;

operation P();
begin
    guard value > 0 do
        value--;
    od;
end;
```

```
operation V();  
begin  
    value++;  
end
```

- When a method is invoked, all of its guards are evaluated in an unspecified order. If all guards are found to be false, the invoking process blocks until one becomes true. Otherwise the block of statements corresponding to one of the guards that were found to be true.
- Orca guarantees that all operations on shared object will be atomic and sequentially consistent and that they will appear in the same order to all processes.

To achieve that Orca executes each method in *mutual exclusion*, implementing indeed the equivalent of a *distributed monitor*.

- Orca objects can either be single copy objects or replicated objects. Replicated objects allow local reads but require a special mechanism for guaranteeing that all updates will reach all replicas in the same order.

Depending on the underlying hardware, Orca uses either:

- (a) a ***reliable broadcast*** mechanism where one central site guarantees the sequencing of the updates.
- (b) a ***primary copy*** mechanism where all updates are always made on a specific copy of the object before being propagated to the other replicas.

5.3 Eden

(to be completed)