

Lectures on distributed systems

Distributed Shared Memory and Memory Consistency Models

Paul Krzyzanowski

Introduction

With conventional SMP systems, multiple processors execute instructions in a single address space. It is possible to run parts of a program in parallel, generally by using threads to specify such parallelism and using synchronization primitives to prevent race conditions. As the number of processors on an SMP system increases, the shared bus becomes a performance bottleneck. The NUMA architecture attempted to alleviate this bottleneck by locating a certain amount of memory close to each processor. Processors still share the same address space, only access to remote memory is slower. This inflicted the addition burden of locating data within the address space for optimal performance. Nevertheless, with a single address space data can be shared and pointers have meaning on all processors.

Unfortunately, large-scale SMP and NUMA systems are expensive to build and prices are kept high as these are not commodity systems. One motivation for using distributed systems in the first place was that a multicomputer system can be cheaper and more reliable. However, all communication and synchronization has to be accomplished with message passing. Each system has a separate address space, so pointers have no meaning outside of the local address space (as we have seen with remote procedure calls). Programs can no longer use global data and any data that is to be shared must be packed into a network message and unpacked upon receipt.

In many cases, it is still easier to program on a uniprocessor or multiprocessor system with a uniform address space than to deal with message passing. Tasks of dynamic data partitioning and load distribution are greatly simplified in this case. **Distributed shared memory** is a technique for making multicomputers easier to program by simulating a shared address space on them.

Architecture

To construct such a system, the underlying hardware and/or software must move data (memory contents) among processors in a way that provides the illusion of a globally shared address space. To get an insight on designing such systems, we can look at one of the earliest systems. Li in 1986 and Li and Hudak in 1989 configured a number of workstations on a LAN and had them all share a single paged virtual address space.

In the simplest form, each page of memory (of the distributed shared address space) exists on only one machine at a time. References to a memory are made through a processor's memory

Distributed Shared Memory

management unit as in conventional paging systems. If a page table entry for a page is valid, that means the page is currently cached locally and the process is allowed to reference it.

A reference to a shared memory location that is not locally present causes a page fault (as would a reference to a non-resident page on a conventional system). This page fault results in a trap to the operating system, which then consults a central server, known as the **directory**, for the location of the needed page. The directory returns the name of a machine that is currently holding the requested page. That machine is then contacted to transfer the contents of the page over to the machine that needs that portion of the address space, with the directory updating its information on the residence of that page.. When the page is brought into a frame in local memory, the local page table is updated and the faulting instruction is restarted. The program continues running. Figure 1 illustrates this process. In short, all operations take place transparently and the DSM system appears like any other virtual memory system to the programmer.

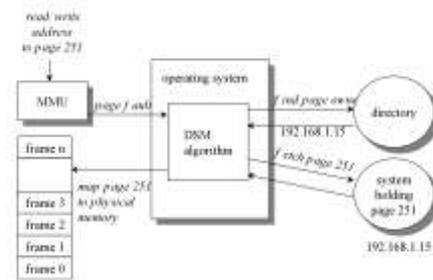


Figure 1. Page fault handling in a DSM system

Clearly, the directory can become a bottleneck, as all non-local requests must be channeled through that central server. An alternate architecture is to implement a **distributed directory**. The directory may be distributed amongst all processors with each processor being responsible for a portion of the address space. When processing a page fault, bits of the page number can be hashed to yield the processor that is responsible for maintaining information about that page. Directories avoid the broadcast that is used in bus-based coherent caches (where each processor's cache would snoop on the address lines). Instead, point-to-point communication is used to the directory.

Design considerations

There are a number of design issues that must be considered when designing a distributed shared memory system. One of these issues is the **granularity** of the data transferred. Since we are building this atop the system's virtual memory system, a restriction is that the data units are multiples of the node's page size. Larger pages are attractive since the cost of migration gets amortized over many localized accesses. However, a larger page size also increases the chance that multiple objects will reside in one page. This is known as **false sharing**. It occurs when process needs to access one memory region and another process needs to access another, non-overlapping, memory region but both memory regions are mapped into the same page. When two nodes need to access the same page, the page will often have to be constantly transferred between the nodes. This leads to **thrashing**.

To avoid thrashing, it is attractive to allow copies of shared data to reside at multiple nodes simultaneously. Replication reduces the average cost of read operations since simultaneous reads can be executed locally at multiple hosts. However, *write* operations become more expensive since each cached copy needs to be invalidated or updated. If the ratio of reads over writes is sufficiently large, this might still be a worthwhile tradeoff. A basic design for replication, similar to DFS file caching, is to allow only one host to have a read/write copy of a page or allow

multiple hosts to have read/only copies of a page. This is known as **multiple readers/single writer replication**. On a *read* operation, if a block is not local, the system will acquire a read-only copy of the block. By doing so, any node that has the block cached with write permissions enabled will be downgraded to read-only permissions. On a *write* operation, if the block is not local or no write permissions exist, the DSM system will invalidate all copies of the block at other nodes and get an exclusive copy of the block. The DSM system must keep track of all nodes that have a copy of the data block. Read/write replication can be extended to **full replication**, where multiple hosts can have read/write access to shared data using a multiple readers/multiple writers protocol. Access to shared data must be carefully controlled to maintain consistency, making efficient operation a challenging task.

Consistency models

The most serious problem with Li's architecture is that only one system may own a memory page at a time. If two processes on two processors are frequently accessing the same page, a highly inefficient ping-ponging of that page between the two processors results. However, to avoid this may create problems in the way memory appears to behave.

A **consistency model** is the definition of when modifications to data may be seen at a given processor. It defines *how memory will appear to a programmer* by placing restrictions on the values that can be returned by a *read* of a memory location. A consistency model must be well understood. It determines how a programmer reasons about the correctness of programs and determines what hardware and compiler optimizations may take place.

Intuition and uniprocessor experience tells us that a *read* returns the data placed by the last write to that memory location. This is known as **strict consistency**. This does not happen in multicomputer systems (or many multiprocessor systems as well).

A slight relaxation of the rules of strict consistency is **sequential consistency**. Lamport defined sequential consistency as:

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 program.

Sequential consistency requires:

1. all memory operations to execute one at a time.
2. all operations of a single processor to appear to execute in program order.

Any interleaving among processors is possible and acceptable. For example, suppose we have two processes with the following code (assume all values are 0 until set):

process 0	process 1
a=1 print(a,b)	b=1 print(a,b)

Distributed Shared Memory

Some possible results may be

sequence 0	sequence 1	sequence 2
a=1 print(a,b) b=1 print(a,b)	a=1 b=1 print(a,b) print(a,b)	a=1 print(a,b) print(a,b) b=1
results	results	results
1 0 1 1	1 1 1 1	1 0 1 0
consistent?	consistent?	consistent?
yes	yes	no

As another example, consider Dekker's algorithm for mutual exclusion running on two processes:

process 0	process 1
flag1 = 1 if (flag2 == 0) <i>critical section</i>	flag2 = 1 if (flag1 == 0) <i>critical section</i>

A sequentially consistent system *cannot* allow an outcome where both `flag1` and `flag2` are zero so both processes can enter the critical section.

Sequential consistency with uniprocessor systems

Most high-level languages present simple sequential semantics for memory operations. The programmer assumes that all memory operations will occur one at a time in the sequential order given by the program (program order).

This illusion can be supported quite easily and efficiently on uniprocessor systems. The system (compiler as well as the memory and CPU hardware) needs to only maintain data and control dependencies. Operations need to be executed in program order only when one modifies a memory location used by another or where one instruction controls the execution of another (for example, a conditional branch).

The compiler or the hardware can freely reorder all operations that do not have these dependencies. For the compiler, these optimizations include register allocation, loop unraveling, and code reordering. For the CPU hardware, these optimizations include pipelining, multiple issue, and lockup-free caches.

Sequential consistency with multiprocessor/multicomputer systems

Sequential consistency is not trivial to achieve in multiprocessor systems and problems may arise if not designed carefully. This section surveys some of these problems.

Cacheless architectures: write buffers

A multiprocessor system without local caches will have to block for any read requests since it cannot proceed without that data. However, it need not wait for write operations to complete, so a write queue can be employed to optimize writes. All read operations will go to memory unless the address is in the write queue, in which case the data will be fetched from there.

Sequential consistency can be violated since a write to different locations may be pending in both processors. Each processor, however, reads the other's data from main memory and not from the other's write buffer (it cannot do so). Consider Dekker's algorithm (page 4). In this case process 1 may have set `flag2 = 1` but the update is still in the write queue for that processor so process 0 reads the old value (0) from main memory. Sequential consistency is violated, both processes see a value of 0 for the other's `flag` and enter the critical section concurrently.

Cacheless architectures: overlapping reads and writes

If memory access is non uniform for all processors and all locations (e.g., a NUMA architecture), the order in which *write* operations reach memory may differ program order. Suppose one processor added a new value to a linked list. It first writes the data and then updates the pointer to the head of the list:

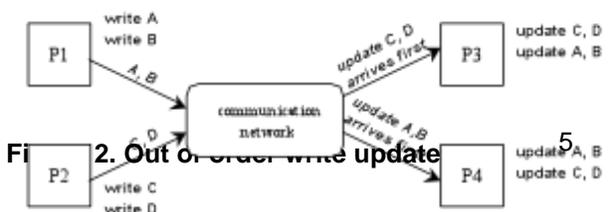
```
*new = data
*head = new
```

If the path to `head` is quicker than that to `new` causing `head` is set first, sequential consistency is violated. Another processor can read the new value of `head` and dereference it before the new value of `data` is written, thus obtaining corrupt data. Another danger is that the compiler or CPU may reorder these writes since it is safe to do so in a uniprocessor system but dangerous in multiprocessor or distributed systems.

The same problem can exist with overlapping *read* operations. Processors that issue early non-blocking *read* operations (in the hope of getting the data to the CPU before the instruction is executed) can suffer from the secondary read being executed first and reflecting a state of memory prior to that of the first read.

Cached architectures

The problem of caches was visited earlier with symmetric multiprocessors (SMP) and is present with a distributed shared memory system. Processors need to know when the contents of their cache are no longer valid so they can update or invalidate the cached entries. With SMP systems, this required that caches be write-through and that a snooping mechanism exist. With a distributed system, a processor that issues a write operation needs to be able to detect that the write has completed. It also needs a mechanism to acknowledge the



Distributed Shared Memory

receipt of invalidation or update messages. A write can be considered complete only after all such acknowledgements are received. The reason for these requirements is that sequential consistency requires that memory operations appear to be atomic (instantaneous at all points) but propagating changes to multiple cache copies is a non-atomic operation.

Even if processors execute memory operations in program order and one at a time, it is still possible to violate sequential consistency if updates of writes by two processors reach the other processors in a different order (Figure 2). To avoid this violation, a condition has to be imposed that all writes to the *same location* be serialized. All processors will then see writes to the same location in the same order. This can be easily achieved if all writes originate from a single point (e.g., a coordinator) and message ordering is preserved on the network. Alternatively, we can delay an *update* or *invalidate* from being sent until any updates that have been issued on behalf of a previous write to the same location have been acknowledged.

Serialized writes are not enough, however. Sequential consistency may still be violated if a processor returns a value of a write to a location before another processor has seen the update. For instance, consider the following sequence of events:

suppose location *A* contains *w*.
P₁ writes *x* to location *A*.
P₂ reads *x* from location *A*.
P₂ modifies location *B* based on *A*: $B \leftarrow y$ where $y = f(x)$.
P₃ receives update of *B*
P₃ receives update of *B*
P₃ reads *B* and *A*, getting the new value of *B* (*y*) and the old value of *A* (*w*).
Sequential consistency has been violated!
P₃ receives update of *A*.

To prevent situations like this, a read cannot be allowed to return a newly written value until all cached copies have acknowledged the receipt of the invalidation (or update) messages generated by the write to that location.

Compilers

Compilers can be a problem when dealing with shared memory. Compiler-generated reordering of shared memory operations can violate sequential consistency. Unless a complex and detailed analysis is performed, program order should be preserved among shared memory operations. In addition, register allocation can eliminate shared memory operations, hence violating sequential consistency.

On the positive side, a number of compiler algorithms have been developed to detect when memory operations can be reordered without violating sequential consistency.

Summary of sequential consistency

A processor must ensure that its previous memory operation is complete before proceeding with the next one. This is called the **program order requirement**. Determining completion of a write typically requires an acknowledgement from the memory system. If caches are present, the write must generate *update* or *invalidate* messages that *must* be acknowledged by all processors that hold cached copies.

In cache-based systems, writes to the same location must be visible in the same order by all processors. The value of a write will not be returned by a read until all updates (or invalidates) are acknowledged. This is the **write atomicity requirement**.

Hardware techniques for dealing with sequential consistency

A goal in system design is to maximize performance while adhering to the memory consistency model that is used.

Write operations require that remote caches be kept valid. To overcome this overhead, the processor can look ahead in the instruction stream for write operations and issue *prefetch-exclusive* requests for those addresses. These requests invalidate data from other caches. Write operations can then be issued in-order when encountered without the having other processors maintain stale versions. Such a technique only works with cache-based systems that use an invalidation rather than an update protocol.

Remote read operations can be quite slow, so the processor can look ahead in the instruction stream for them and speculatively issue the read requests before they are needed. If the read line gets invalidated or updated, the processor will have to roll back and reissue the necessary read and all subsequent operations that depend on it. This is suitable for dynamically scheduled processors with rollback mechanisms present (which currently are used for dealing with branch mispredictions).

Both of these techniques will be supported by several microprocessors, including the Intel P6 and the MIPS R10000.

Further relaxation

To achieve better memory efficiency (and perhaps complicate life the programmer), the rules of sequential consistency may be relaxed beyond the guarantees provided by sequential consistency. The program order requirement can be relaxed to affect the values of memory contents between successive read/write operations. Relaxing the program order requirement requires us to examine several combinations (read to a following read/write, write to a following read/write), each or all of which can be relaxed. The write atomicity requirement can be relaxed to allow a processor to use a value before others get it. For every relaxation of the rules we must consider that in the end we want a system whose outcome will be deterministic.

One relaxation is relaxing the program order constraints in the case of a write followed by a read to a different location. This seems innocuous but can affect program correctness when the address read depends on the value written. For example, a sequence:

```
write head
read list[head]
```

can produce unexpected results. Two architectures that relax write-read ordering are the IBM 370 and the Sun SPARC V8. To enforce sequential consistency in cases when it is needed, the read can be changed to a read-write combination, writing back the value just read. Since consistency is guaranteed for read-write ordering, the entire sequence will be sequentially consistent. This is the solution that must be taken on a machine such as the Sun SPARC V8. Alternatively, the processor can provide a mechanism allowing the programmer to force a set of operations to be consistent. The IBM 370, for example, provides serialization instructions that may be placed

between two operations (some of these are memory operation instructions such as compare-and-swap and others are branch instructions). Placing a serialization instruction after each write on each processor will provide sequential consistency.

Relaxing the write-write ordering allows the system to eliminate ordering constraints between writes to different locations. For example the SPARC V8's Partial Store Ordering (PSO). As with write-read ordering, there are times when sequential consistency is needed. The SPARC's PSO provides an STBAR instruction for explicitly imposing order between two writes.

Weak consistency

A further relaxation of memory consistency is to relax program order between *all* operations to different locations. In this case any read or write may be reordered with respect to a read or write to a different location. These consistency models are known as **weak consistency** (or *weak ordering*). Consider the case where a process is in a critical section and reading/writing data that nobody else will access until it leaves the critical section. In this there is no need to propagate writes sequentially or at all until the process leaves the critical section.

To accomplish this, we introduce the concept of a **synchronization variable**. Operations on this variable are used to synchronize memory. When the synchronization completes, all writes have been propagated. Think of the synchronization variable as a blocking instruction that does not return until memory is synchronized.

Weak consistency enforces consistency on a *group of operations* rather than individual reads and writes. Its consistency guarantees are:

1. Access to synchronization variables are sequentially consistent (all processes see synchronization variables in the same order).
2. No access to a synchronization variable is allowed until all previous writes have completed. The synchronization variable serves to flush the pipeline of writes.
3. No read or write is permitted until all previous accesses to synchronization variables have been performed. When memory is next accessed, all previous synchronizations have been performed, ensuring that all memory is updated and changes have been propagated.

Release consistency

One problem with weak consistency is that, when accessing a synchronization variable, the hardware does not know if this is because the process has finished memory accesses or is about to start them. During a synchronization the system must always make sure that:

1. all locally initiated writes have completed (propagated).
2. Gather information on updates from other machines.

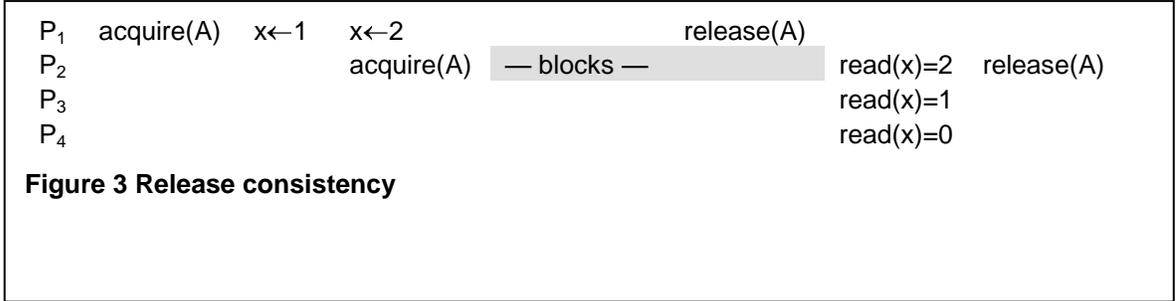
Release consistency alleviates this problem by separating synchronization into two operations:

Acquire access requests entry to the critical section. It acquires changes from the outside.

Release access states that memory operations are complete. Local changes can be propagated.

Although shared memory transparency was lost on a per instruction basis, a more efficient system can be constructed through explicit acquire and release operations. Critical sections are not always necessary. Instead, *barriers* may be used. A **barrier** is a synchronization mechanism that prevents a process from starting phase $n+1$ until *all* processes are finished with phase n . Arrival at a barrier results in a *release* operation. Departure from a barrier (when all other processes have arrived at the barrier and are finished with their respective release operations) results in an *acquire*.

The memory system has no obligation to provide proper values to processes that do not perform the appropriate acquire operation. For example, in Figure 3, processes P_1 and P_2 properly coordinate acquire and release. Processes P_3 and P_4 do not and hence the value of x is undefined for them. Acquire and release operations need not apply to all of memory (the entire address space need not be shared). They can apply to a group of variables. In this case, only that region of memory is considered shared and will be synchronized with acquire and release. This form of release consistency is generally called **eager release consistency (ERC)**.



One inefficiency is that with a *release* operation, changes (invalidations) are propagated to every process, even those not interested in the changes. Instead of propagating updates or invalidations to all processes at *release*, we can wait until some other node performs an *acquire* and then propagate the changes to the acquiring process. For a *release* operation, a node will simply note its changes locally (mark which pages were modified and ensure that they do not get modified after the *release*). On an *acquire*, the acquiring node will contact the node that performed the *release* to get its changes. This variant is known as **Lazy Release Consistency (LRC)**.

Another variant of release consistency requires one node to be the “home” of a specific page. At *release*, differences between the original page and the modified page are computed and transferred to the home node for that page. The node that just did the *release* can now discard these differences and feel free to modify the page. The home node will apply the differences as soon as they arrive. When other nodes perform an *acquire*, they contact the home node for the latest copy of the page. This is known as **Home-based Lazy Release Consistency**.

Entry consistency

Release consistency still has the goal of synchronizing *everything* (all of memory or all protected variables or a shared region). **Entry consistency** is used just like release consistency. The

Distributed Shared Memory

programmer (or, better yet, the compiler) still has to issue acquire and release operations, but now each ordinary variable has a synchronization variable associated with it (even individual elements of arrays may have synchronization variables if they are processed individually).

The goal is to reduce the overhead associated with acquiring and releasing a synchronization variable since only a few shared variables may need to be synchronized at any time. Multiple critical sections with disjoint shared variables can execute in parallel.

A few examples

Compaq's (formerly Digital's) Alpha processor provides two instructions to deal with consistency. The *memory barrier* (MB) instruction maintains program order from any memory operations before the MB to memory operations after the MB. A restricted form of the memory barrier is the *write memory barrier* (WMB), which provides this guarantee only among memory write operations.

Sun's SPARC V9 processor supports *relaxed memory ordering* (RMO) and allows for sequential consistency with the MEMBAR instruction. This instruction can be customized to order a combination of previous read/write operations with respect to future read/write operations (thus serving as a barrier for classes of memory operations). It supports an argument in which a four-bit encoding can specify any combination of the following orderings:

read to read

read to write

write to read

write to write

The Power PC provides a SYNC instruction that is similar to the Alpha's MB with one difference: if a SYNC is present between two reads to the same location, it is possible for the second read to return the value of an older version than the first read (violating program order). To counteract this, a read-write (of the same location) sequence can be used to enforce sequential consistency.