CHAPTER 2

# Distributed Shared Memory

Distributed shared memory, DSM, though it may sound like a contradiction in terms, is an attempt to merge the ease of programming of shared memory, with the scalability of distributed memory architectures. Hiding the distribution aspects of an architecture for the programmer eases programming and especially porting existing, applications. DSM systems exist in a large number and with very varying behavior.

Before starting to identify classes of distributed shared memory systems, it is perhaps a good idea to define what shared memory actually means, [Culler 98] states that

> *Writes to a logical shared address by one thread are visible to reads of the other threads.*

This definition is straightforward and probably compatible with the perception most people have of shared memory. It is important however to understand that this definition identifies no condition under which a write is seen by another thread. This problem will be introduced in this chapter, and investigated more thoroughly in the next.

The primary distinction one may make is between DSM systems that are based on hardware, and those that are based on software. While the border is not quite as rigid in reality, since some systems use a combination of the two, this distinction eases the classification of distributed shared memory systems. This chapter will go on to describe some hardware based DSM approaches and systems, and then a set of software based

approaches and systems. This description is meant to help put the PastSet work in perspective.

## 2.1 Non Distributed Shared Memory

Defining what is not distributed shared memory is a good idea before discussing what is distributed shared memory. The trivial case is one CPU with one memory block. If this one CPU runs a multithreaded application the memory will appear shared to the threads. More realistic views of shared memory are several CPUs connected to the same memory block or blocks using either a shared bus or a cross-bar switch. This is sometimes called Symmetrical Multi Processing, SMP, though SMP has come to mean the shared bus approach more specifically. Taken together the two are more often refereed to as Uniform Memory Access, UMA. Uniform because access to a memory address has the same cost independently of the address. This uniform cost may be partly hidden by caching, but this in effect does not change the UMA property.

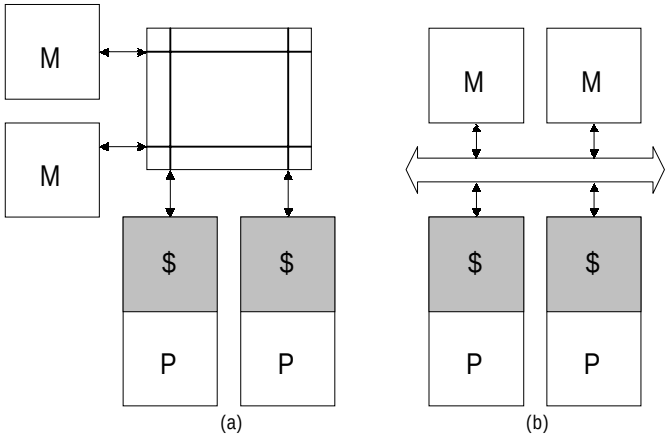


*Figure 1 UMA approaches, (a) cross-bar switch and (b) Shared bus*

The cross-bar switch approach can be found in machines such as the HP Exemplar V Class, while the shared bus approach can be found in machines such as the SGI Power Challenge and the Sun Enterprise Server. The latter may intuitively not look like an UMA machine, as processors and memory modules are collected on the same system boards, which then interface with the shared bus. Non the less this is a standard UMA architecture, since a CPU can access memory in a different system board in the same time it takes to access memory on its own system board. Unfortunately UMA architectures does not scale well [Bianchini 94] .

## 2.2 Hardware based distributed shared memory

Hardware based DSM systems have several advantages on software based systems, mainly that if things can be done transparently in hardware they are likely to run faster. However custom hardware is expensive, and history has shown that custom hardware has a hard time keeping up with technology advances.

There are three main approaches to designing hardware based distributed shared memory. Section 2.3 describes the non-uniform memory access approach, the cache coherent non-uniform memory approach is described in section 2.4 and finally the cache only memory architecture is described in section 2.5 .

## 2.3 Non Uniform Memory Access

The Non Uniform Memory Access architecture, NUMA, approach is the simplest approach to hardware based distributed shared memory. The name gives a fairly good idea as to how this architecture differs from the UMA approach. Rather than maintaining a uniform access to memory, access time and memory behavior differs, depending on the address that is accessed. NUMA machines are usually made up of several nodes with one or more CPUs and a block of memory each. Memory on the same node can then be accessed quickly, as the CPUs memory bus is connected directly to the local memory, while access to memory on a different node is routed via a scalable interconnect and is often not cached. The NUMA architectures utilize that local access, such as the executable and the stack of a process is fast, and that access to shared data can be fast also if it is placed in the processors local memory. Distributing data to nodes, so that most possible accesses are local, need to be considered by the programmer, or to a certain extent the compiler.

### 2.3.1 The Cray T3E

Non uniform memory access can be found in the Cray T3E machine [Scott 96]. The T3E is scalable up to 2048 processors. The T3E follows the above generic description for a NUMA architecture fairly closely. The machine is made up of processing nodes and IO nodes. Each processing node consists of a Compaq Alpha processor, cache, local memory, a memory controller and a network interconnect. Each node is connected to six neighbors, constituting a three dimensional cube architecture. Any processor can, without preparation, read and write the memory that is placed on the same board, this memory can also be cached. If a process wishes to access memory on a remote node it must first set up addressing of the node on the memory controller, this is a do once action. Once the memory controller know of the remote memory, access to this memory is automatically caught by the memory controller and routed to the corresponding node of the memory, where the local memory controller will perform the actual memory access on behalf of the issuing processor. Remote memory is usually not cached, but if it is there is no mechanism available for invalidating the entry if the data changes. In fact the memory access is only guarantied to be coherent until it leaves the node of the memory. Since the interconnect is a three dimensional cube, routing a message through the machine takes time and during this delay the data may change in the actual memory.

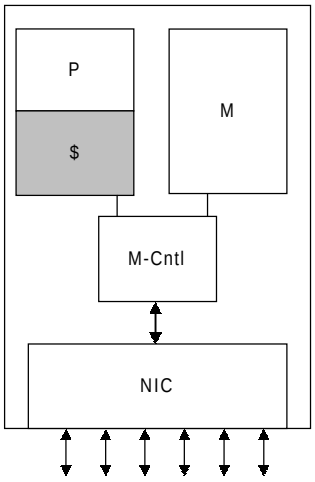


*Figure 2 Node layout of the Cray T3E NUMA machine*

## 2.4 Cache Coherent Non Uniform Memory Architecture

While NUMA architectures has several advantages, primarily simplicity and scalability, the lack of memory coherence makes programming more difficult and solving certain problems thus requires more memory accesses which in turn slows execution down. To achieve memory coherence one need to add a cache coherence mechanism to the non-uniform architecture, which then becomes a Cache Coherent Non Uniform Memory Access architecture, CC-NUMA. CC-NUMA architectures are basically an approach to achieve the scalability of the NUMA architectures and maintain the programmability of UMA architectures.

CC-NUMA machines are made up much like the NUMA machines, i.e. one or more processors are placed on the same node as a memory block and different nodes are then interconnected. But while the NUMA architecture has no coherence of memory, which is placed on remote nodes, and thus usually does not cache it, CC-NUMA architectures do cache remote memory locations and keep them coherent, e.g. invalidate all cached copies on a write if a looser consistency model is not defined. This invalidation requires that the system keeps track of all cached copies of a cache line, and to maintain sequential consistency that all copies are invalidated before a write operation returns. Consistency models will be further explained in the next chapter. Maintaining a picture of which addresses are mapped where, requires significant amounts of extra memory. The extra memory to track cache lines favors larger cache lines, because larger cache lines results in fewer entries to keep track of, however larger cache lines also increases the possibility of false sharing, which is a phenomena that occurs when two addresses are not actually shared, but the sharing granularity is so large that they appear to be shared anyway. False sharing degrades performance and because of this identifying the optimal coherence granularity is an important issue when designing CC-NUMA architectures.

### 2.4.1 SGI Origin 2000

There are several commercially available CC-NUMA architectures on the market today. One of the most widespread of these is the SGI Origin 2000. Just as the Cray T3E, the SGI Origin 2000 is made up of interconnected system boards which in this machine holds two MIPS R10000 CPUs or one MIPS R12000 CPU, and a part of the global memory. The CPUs have both a level one and a level two cache, which are kept coherent. Access to memory is done via a memory controller named the HUB. The HUB handles memory mapping, access to remote memory and cache coherence. Cache coherence is achieved via a directory scheme [Lenoski 90], and kept at cache line size, which is 128 bytes. The cache copy information, the directory information, is kept in main memory for small systems, up until 32 processors, and in a special memory block for larger systems. A cache entry can be marked as copied in one processor, at zero or more nodes or at some set or sets of nodes if the system is larger than 64 nodes. Directory information is looked up in parallel with the actual memory access. Each box has a fixed number of interconnections, so the boards are connected via a hyper-cube, the order of which decrease as the number of boards increase. Each box can hold 64 processors and more boxes can be interconnected in what is called a fat cube architecture. The interconnection has a total bandwidth of 1.56GB/s per node, and supports prioritized virtual channels to solve various problems.

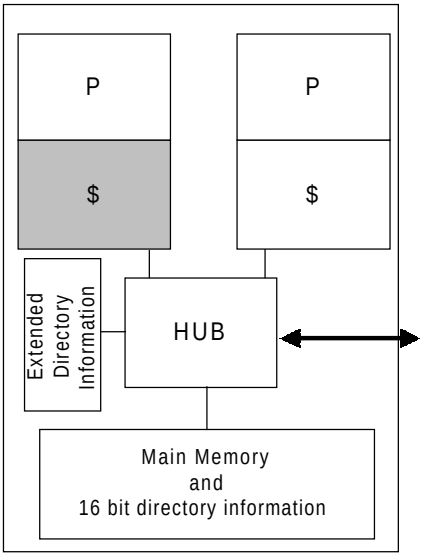


*Figure 3 Layout of an Origin 2000 node*

## 2.5 Cache Only Memory Architecture

A drastically different approach to maintaining cache coherence in distributed memory systems, is to eliminate the concept of main memory entirely. Instead of working with the concept of physical main memory, the system provides only cache memory [Saulsbury 95]. This cache memory is associated with a virtual memory space, rather than a physical memory, and thus the concept of main memory disappears. Instead memory blocks are

always tagged, and like the cache entries on an SMP system, the cache can be replicated and migrated freely among the nodes in the system. When a processor needs a memory block, the required block is attracted to the node and placed in what is otherwise thought of as the main memory, which in COMA architectures is denoted attraction memory. Like ordinary cache entries other attraction blocks can replace cached attraction blocks when the capacity of the cache is exhausted. However, in COMA machines the system cannot allow the last copy of a block to be flushed, since there is no actual memory where a coherent version can be stored.

### 2.5.1 KSR1

The Kendall Square Research machine KSR1 [Frank 93], is the only commercial machine ever build on the COMA principle up until now. The KSR1 is made up of system boards, which consist of one processor and up to 32 MB attraction memory, called ALLCACHE in KSR terminology. The KSR1 uses a hierarchical directory structure for maintaining coherence. Memory is made up of 16 KB pages each divided into 128 sub-pages of 128 bytes. While memory is allocated in blocks of 16 KB coherence is kept at the sub-page granularity. When a processor references an address that is not available in the local portion of the ALLCACHE an ALLCACHE ENGINE traps the access and locates the nearest copy of the data and copies it into the local ALLCACHE memory. If the operation is a write, all other copies of the sub-page are invalidated, in order to maintain sequential consistency. A sub-page in the KSR1 can be in one of four states:

- **Exclusive** (owner), which indicates that this is the only valid copy of the sub-page in the system and can be both read and written.
- **Copy**, which means that the sub-page is a read-only copy.
- **Non-exclusive** (owner), which is also a read only copy of the sub-page, but this copy cannot be flushed without transferring ownership to another copy.
- **Invalid**, meaning that the copy has been invalidated by a write on a different copy.

*Figure 4 The four possible states of a sub-page copy in the KSR1*

The ALLCACHE ENGINE can service 32 nodes; if more than 32 nodes exist in the system, the ALLCACHE ENGINEs of each set of 32 nodes is connected to a higher level ALLCACHE ENGINE. The ALLCACHE ENGINE uses a matrix to keep track of sub-page copies. Each sub-page has a column in the matrix and each node a row. Each entry in the matrix is two bits, enough to reflect the four possible states of a sub-page. Locating one or more copies of a sub-page is then simply a matter of traversing the entire column for the corresponding sub-page. This approach is rather wasteful in terms of memory, in a system with 256 nodes, the memory required for controlling a sub-page is the same as the sub-page itself, and thus a system with 1K nodes requires four times the memory that is actually available to the applications. This problem is addressed in the KSR1 by letting the ALLCACHE ENGINE maintain the sub-page matrix in a compressed form. A later

version, the KSR2, was also build, but except for doubling the processors speed and the bandwidth of the interconnect the two machines are identical.

## 2.6 Software based distributed shared memory

While the hardware based Distributed Shared Memory all provide high performance scalable solutions, they all carry the inherent penalty for hardware based systems. The custom hardware is expensive and upgrading to new processors with new features is complex and often requires modifications to the custom hardware. To address these problems, some or all of the DSM functionality may be migrated to software. Software solutions are cheaper to implement and easier to modify, and will often be able to run on new processors without modifications. Thus software based distributed shared memory reduces the cost over hardware based systems significantly, but is a change to runtime efficiency. The following is a brief description of the concepts in software based DSM, and some system examples. In the literature software based DSM systems are often divided into only two groups, page-based and object-based DSM systems. The philosophy behind this coarse definition is that the page-based systems use partial hardware support and have a fixed coherence granularity, namely a page, while the object based systems are usually completely software based and work on software defines coherence granularities, e.g. objects. In the following the object based systems has been divided into a set of more finely defined groups, and while most systems include elements from other models, they share the overall group characteristics.

## 2.7 Variable Based Distributed Shared Memory

Variable based DSM systems model CC-NUMA systems in software, thus keeping cache coherence on a very small granularity level, even smaller than cache line size if this is convenient. The major challenge in these systems is to ensure cache coherence, since the hardware provides no support for keeping the coherence. To this end all memory operations to shared regions are instrumented, which means that an instruction block that checks whether the data is available or not will precede i.e. a load from an address. This instrumentation of all accesses to shared memory regions is very costly and several techniques have been developed to reduce this cost. However this approach has never become really popular.

The variable based DSM systems can be divided into two major blocks, actual variable based DSM and region based DSM. The default variable based DSM system has a close integration with the compiler [Chiueh 95] and often the language. Variables may be defined as local or shared, and only access to shared data is instrumented, and the instrumentation is added when the compiler does code generation. This technique can utilize a large amount of semantic information that can be derived from the source code, and may keep coherence on the variable level as it knows the type of the variable when the code is generated.

Alternatively, the instrumentation can take place after the code has been generated, as it is done in the region based DSM systems [Johnson 95]. This approach allows applications to be written in any language and compiled with any compiler. The DSM system then analyzes the executable and attempts to differentiate between local and shared data, e.g. information on the stack is local and all other is shared. The post-compiler then inserts instrumentation code around the operations that access shared data.

### 2.7.1 Midway

Midway [Bershad 93] is a variable based distributed shared memory system, based on a compile time identification of shared variables and synchronization variables. Coherence is kept on an arbitrary granularity level, where it is maintained on a set of variables that are associated with a given critical section. This allows the system to support entry consistency, which will be described in the next chapter.

Midway is a combined compiler and runtime system that requires the programmer to explicitly consider distribution and consistency aspects, however the entry consistency model is very natural to the way of writing parallel applications. Shared variables must be declared as shared in the program, and must be associated with at least one synchronization variable. A set of one or more shared variables are then made coherent before an entry into a critical section is completed, i.e. if a variable *x* if associated with a semaphore *S* then *x* will be made coherent at the time the program issues a p(*S*), accessing *x* outside a critical section will return a value that may not be coherent.

### 2.7.2 Shasta

Shasta [Scales 96] [Scales 97] is a region based distributed shared memory system for Compaq Alpha processors. The system takes an SMP capable executable and instruments it. A set of optimization techniques has been developed for the Shasta system:

1. Careful layout of the address space
2. Instruction scheduling of the checking code with respect to the application code
3. Using a novel technique to do coherence checks based on the actual loaded value
4. Applying a method to reduce extra cache misses caused by the checking code
5. “Batching” together checks for multiple loads and stores

*Figure 5 Shasta software CC-NUMA optimizations techniques*

1. is self explanatory, by manipulating the memory map, Shasta is able to optimize the availability checking. Issue 2 utilizes knowledge of the out-of-order and data-dependency issues in the Alpha processor to schedule checking instructions in slots where the CPU would otherwise place a bubble, e.g. insert no-ops in the pipeline. Issue 3 is a technique that attempts to reduce the checking cost of loads by applying a ‘make the common case fast’ approach, instead of instrumenting a presence check before the load, the result of the

load is compared to a fixed invalid-flag. If the result of the load is not the invalid flag the application can continue, otherwise a miss-handler function is called, which first checks whether the miss is actual or false, e.g. if the loaded value indeed by chance is the same as the miss-flag. The fourth issue attempts to minimize the instrumentation codes interference with the cache and issue five targets the option of checking several read or write operations in sequence in one checking operation.

By applying these techniques for code checking, the Shasta system is able to reduce the cost of instrumentation from 50-150% to 35% on a set of applications taken from the SPLASH-2 benchmark [Woo 95].

Shasta has a default coherence granularity of 64 bytes, which can be changed to a different size specified by the user.

### 2.7.3 Munin

Munin [Carter 91] is a variable based system, that supports release consistency (described in the next chapter) amongst others, and supports a large number of special annotations by which the programmer can tell the run-time system how access to a variable behaves. Coherence is kept on variable granularity, where a variable can also be a structure compromising two or more sub-variables. If a variable is more than a page size, 8 KB on the hardware used by Munin, it is spilt into two or more coherence blocks of at most one page in size. As in Midway, variables in Munin must be explicitly declared as shared. The supported annotation declarations are:

- **Read-only**, once a Read-only variable is initialized it is only read
- **Migratory**, variables will be used in bursts including writes by one thread, and should be kept in one copy only.
- **Write-shared**, variables are written by a number of threads before they are assembled and made coherent in all instances.
- **Producer-consumer**, variables are written by one thread and read by one or more.
- **Reduction**, variables are optimized for reductions e.g. global sum etc.
- **Result**, variables are shared write variables but are not made coherent after a write, only the thread that issues a read receives a coherent copy.
- **Conventional**, variables are read replicated and write exclusive.

*Figure 6 Variable annotations in Munin*

Munin does not instrument variable access but traps access to the variables by utilizing the memory management hardware, like the shared virtual memory systems described below.

## 2.8 Shared Virtual Memory

While most other models for software based distributed shared memory require the programmer or the compiler to adopt to a DSM model, one model need not make such requirements. This model is the **shared virtual memory, SVM** [Li 86]. The idea is to use the systems memory management unit MMU to trap access to distributed shared memory, very similar to demand paging systems. As the MMU already monitors the presence of pages, in order to support demand paging, shared virtual memory may easily be added to the paging handler, so that it supports paging with remote memory in addition to paging to disk. Examples of SVM systems are IVY [Li 88], Treadmarks [Keleher 94] [Amza 96] and CVM [Keleher 96].

The concept of SVM is very simple. Processes or threads share the same virtual address space on all nodes in the system, but the physical pages are not available at all nodes. When a process accesses a page that is not at the node where it is executing a *page not present* trap will occur and the SVM system can step in, get the page and return to the faulting process to try again. As the MMU control precedes the cache lookup in a CPU, SVM has no immediate problem with maintaining cache coherence. One feature of SVM systems that is not available in any of the other DSM memory models is the fact that the text section of a process can also be handled by the DSM system.

SVM systems can use any of the consistency models described in the next chapter, with more or less requirements placed on the programmer or compiler. Replication can be done at various levels; the simplest is that pages can be marked read only in which case replication is trivial. If a process then attempts a write to the page and that write is otherwise OK, i.e., not in the text segment, the DSM system must invalidate the page on all other nodes at which it is present, and then give the writing process exclusive access to the page. Alternatively any write to a read-only page can be propagated to all nodes that have replicates, this can be done in various ways depending on the consistency model the system complies to. Full replication can be achieved by entering exclusive write mode and create a write diff. This diff is then used to reassemble the page when it returns to read state again.

While SVM systems may model a SMP machine completely, and thus require no software modifications, performance is dependent on some sort of relaxed consistency model. In addition, SVM must use the page size that is inherent to the CPU as the coherence granularity. As page sizes are decided on as a tradeoff between page table size and potential thrashing problems when doing demand paging for pure virtual memory purposes, the page size on most machines is large, typically 4KB. This granularity is quite large for SVM purposes, and often triggers false sharing as well as requiring more bandwidth for the communication than necessary, especially synchronization variables are often only one byte, but will still result in passing 4KB over the network. Another hidden cost in SVM systems is that when a page is migrated from one node to another, all CPUs on the node that the page leaves need to be interrupted and have the Translation Look-aside Buffer, TLB entry of the page cleared if this is supported by the CPU, or the complete TLB flushed otherwise.

### 2.8.1 IVY

IVY [Li 88] was the first SVM system to be presented. IVY is a user level implementation of SVM, build on top of a modified version of the Aegis operating system. IVY ran on a set of Apollo Domain workstations connected by a 12 Mb/sec token-ring network. The page size on the Apollo workstations were 1 KB, which made the problem of false sharing, or contention, smaller in IVY than in more modern systems. IVY supports true coherence that is sequential consistency, by invalidating all read replicas of a page before allowing a write to perform.

IVY implemented two approaches to data location a fixed distributed manager system, where each node in the system was assigned a set of pages, the location of which it has to maintain. The other approach, the dynamic distributed manager system, keeps track of the ownership of all the pages in each node's page-table. Each page is then marked as true owner, which means that the node is guarantied to be the owner of the page, or probable owner, this way it is easier to obtain a copy of a page, but harder to invalidate all copies.

Parallel execution in IVY is performed via threads that run in the same virtual memory space. Memory can be allocated as shared or private, to allow programmers to optimize programs by identifying data that need not be part of the SVM system. The text memory of each thread is private and need therefore not be maintained by the SVM system. The stack of each thread is allocated as shared data. The threads in IVY can be marked as migratable or non-migratable; as the stack is mapped in the SVM part of the memory, migrating a thread to another node becomes simple.

IVY was meant as a platform for experimenting with SVM, and as such did not have a particular high performance.

### 2.8.2 TreadMarks

TreadMarks [Keleher 94] is a user level SVM system, which has been implemented on a variety of architectures. TreadMarks introduced the release consistency model and the lazy release consistency model, by supplying the API with a specific set of synchronization mechanisms. The philosophy behind the release consistency model is that correct parallel applications should not contain race-conditions, so race conditions have to be eliminated via synchronizations. A performance advantage can be gained from this, as TreadMarks then defines that memory should only be updated once a process leaves a critical section. This way iterated writes can be updated in one operation and to support this, TreadMarks also includes a shared write mechanism. The concept of shared writes will be covered in the next chapter.

### 2.8.3 SHRIMP SVM

The SHRIMP SVM system [Rudrajit 98] is a very high performance SVM system designed to utilize physical shared memory where available, SMP, and user level network access to a high performance network, the Myrinet. SHRIMP SVM applies a consistency protocol

called Home based Lazy Release Consistency, HLRC. HLRC uses a software mechanism to detect writes to a page, when leaving a critical section a node will send its diff to the home node of the page, which in turn applies the diff and maintains Lazy Release consistency by using a logical vector clocking mechanism. Reads to modified pages are trapped via a page fault and result in the node retrieving a coherent version of the page. The main attraction of HLRC is that access to a page will never cause page faults on the home node of the page so, when locality can be achieved, a lot of overhead can be eliminated. The home node doesn't have to create a diff either and non-home nodes can always achieve a coherent version of the page by retrieving a copy from the home node.

## 2.9 Distributed Objects

Perhaps the simplest way to implement DSM is the distributed object method. This method lends itself directly to the object-oriented analyses, design and programming philosophy that had a huge growth in the 1980's. The constraint enforced by this method is that shared data must be encapsulated in an object and that these data can only be accessed by methods within this object. Object based DSM systems have been a research topic for quite some time and are still an active research area. Examples of object based DSM systems are Orca [Bal 96] and Emerald [Black 86] from the University of Copenhagen.

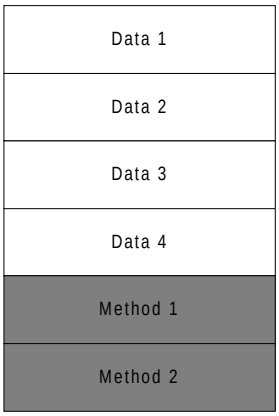


*Figure 7 A generic object.*

The distribution aspects of object based DSM can be divided into three main areas, first the non-distributed method, second the migration method and last the replication method, or a system may employ more of these model, such as Hawk [Ben 96]. Efficiency of execution and complexity of implementation also order these three methods. The simplest implementation is to not distribute at all. When an object is created is stays at the node that created it. When other nodes wish to access the object they transfer the control to the remote node and continue there, this can easily be implemented using e.g., RPC [Birrell 84]. The advantage of this approach is that it can be implemented very easily, and in general won't require a lot of network bandwidth. On the other hand objects need to be distributed in a way that favors concurrency, i.e., if the same object is used intensively by all nodes no real parallelism is achieved, and the rationale behind DSM is lost.

Migrating objects has the advantage that once an object is located at a node, that node can access the object at native speed. Migrating objects takes a little more work than the static approach, but is still fairly simple, and as an object can only be accessed by the node that it is placed on, synchronization problems are solved at no cost. The problem of course is that as objects grow larger the required bandwidth to sustain object migration raises dramatically. Even though the run time system can introduce a latency before releasing an object, to favor object access in tight loops, an object that is often accessed will lead to thrashing, and the time that nodes spend waiting for access to objects will raise.

Replication of objects requires a lot more of the run-time system than the previous two models. By replicating objects all nodes can hold their own copy of the object, and only access control and changes to the object need to be sent on the network. Another advantage of replication of objects is that this approach allows the system to use some of the more relaxed consistency models as described in the next chapter. In fact, by using a replicated object DSM system, one can hide the additional complexity that comes with the most relaxed consistency models. It is intuitive that replicated objects lends themselves directly to a transparent implementation of entry consistency, which would allow concurrent reads on replicated objects, and require only changed data to be transmitted when updating after an exclusive access.

It is obvious that distributed objects is an easy way to implement DSM, and with the use of entry consistent replicated objects, they can be both easy to use and efficient. The problem occurs with applications that do not lend themselves to the object orientated approach, and this includes the classic “dusty-deck” algorithms that are the traditional target for multiprocessors. Widely distributed systems with little competition on shared data, such as large database systems, seems to be interesting targets for object oriented DSM systems.

### 2.9.1 Emerald

Emerald [Jul 88] is an object based language and distributed shared memory system. The philosophy behind Emerald is to model all aspects of a distributed system via a unified object model, including both data objects and code objects. Emerald does not replicate objects, but focuses on distribution and migration of the available objects. Coherence is kept in compliance with sequential consistency, by using monitors to control access to objects. Emerald works with a set of three distinct types of objects:

- Global Objects, which can be migrated between nodes in the system and are globally addressable.
- Local Objects, which are encapsulated in a global object and can not be moved independently of that global object. Local objects are also addressable only within the global object in which they are placed.
- Direct Objects, are primitive objects, e.g. integers. Direct objects are always reduced at compile-time.

*Figure 8 Object types in Emerald*

Direct objects can also be pointers, while most object-based systems use indirect references as pointers, Emerald uses real addresses as pointers, in order to improve local performance. When an object is migrated, all direct pointers in the system are updated to comply with the new location. This update is quite expensive, but Emerald works under the assumption that migration is a rare event. Emerald goes back to 1986, but is still an active project and now also supports execution on heterogeneous systems [Steensgaard 95].

### 2.9.2 Orca

Orca [Bal 88] [Bal 92] is an object based programming language and distributed shared memory system. In Orca objects are in line with the conventional conception of objects, e.g. an object is a set of data and a set of methods. Execution units are processes, not objects as in Emerald. Since the execution units are processes, it also means that the only thing that is shared amongst processes in the system is the objects that are explicitly passed as parameters to new processes.

Orca is based on distributed coherent objects, e.g. Orca does not invalidate objects on write, but propagates the write to all copies of the object. This is done by sending all writes to a primary copy of an object, the node that holds this copy will then update all copies. Coherence is ensured via a 2-phase commitment protocol. The backbone of Orca is a mechanism for reliable broadcast on media that support broadcasts, and a system for reliable multicasts on systems that does not provide physical broadcast options.

Orca is still an active research system, and has proven very good performance [Willem 92] on a set of applications that are not necessarily focused on linear algebra problems.

### 2.9.3 Aurora

Aurora [Lu 97] is a C++ based object oriented distributed shared memory system, that complies with the rules of release consistency. Aurora uses what is called **scope behavior** in order to increase performance on distributed applications. While other release consistency based DSM systems rely on explicit synchronization primitives to control updates of replicated data Aurora uses the scope in the application. This way data will only be made coherent when the source code reaches an '}'.

Aurora is based purely on software so writes have to be detected by software, and it provides primitives to distribute scalar and vector data over a cluster, as well as a set of primitives to describe the single program multiple data, SPMD, type of parallelism. No mechanisms are available for more coarse grained parallelism.

## 2.10 Structured Distributed Shared Memory

Rather than choosing between a distributed object approach that requires data access to be performed via the associated methods, or a distributed variable approach that requires no control over the data, intermediate models have been proposed. These models are known as structured memory, but most often these models have little in common with each other. However, the one thing that structured shared memory models usually have in common is the fact that rather than requiring a new programming language, or living within the constraints of an existing language, they work with the basis of an existing language and expand this language with a few functions. Examples of such structured distributed shared memory models are Linda [Gelernter 85], Global Arrays [Nieplocha 94] and JavaSpaces [Sun 98]. The following describes these structured shared memory models briefly.

### 2.10.1 Linda

Linda is available in a number of varieties, including C-Linda, FORTRAN-Linda and C++ Linda, as well as a set of implementations that modify the Linda model [Kielmann 97]. The Linda aspect of these languages differs only slightly. Linda structures memory into a space called the **tuple space**. A tuple space is shared between all the participating nodes, and as the name indicates, contains tuples. A tuple is an n-vector where each element can be an arbitrary data type, excluding tuples themselves. The first element in a tuple is a string called a **flag**. Examples of tuples are:

```
(“Person”, “Doe”, “John”, 23, 82, BLUE)  // BLUE is an enumerate
(“pi”, 3.141592)
(“grades”, 96,  [Bm, A, Ap, Cp, D, Bp]) //A, Ap… are enumerates
```

*Example 1 Three examples of Linda tuples.*

A task can put a tuple into the tuple space with the **out** command.

```
out(“Person”, “Doe”, “John”, 23, 82, BLUE);
```

*Example 2 Placing a tuple of constants in the tuple space*

Will add the tuple (“Person”, “Doe”, “John”, 23, 82, BLUE) to the tuple space, and

```
out(“Person”, Last_name, First_name, Age, Weight, Eye_color);
```

*Example 3 Placing a tuple with variables in the tuple space*

will add a tuple with the values represented in the variables used. Tuples can be retracted from the tuple space similarly.

```
in(“Person”, ? Last_name, ? First_name, ? Age, ? Weight, ?
Eye_color);
```

*Example 4 Retrieving a tuple from tuple space*

Will retrieve any tuple from the tuple space that matches the template (“Person”, string, string, char, int, EYE_COLOR). Once the tuple is retrieved it is removed from the tuple space, this is done in an atomic operation so that no two processes can retrieve the same tuple. A more selective retrieval can be performed as

```
in(“Person”, ? Last_name, ? First_name, 32, ? Weight, GRAY);.
```

*Example 5 Retrieving a specified tuple from tuple space*

This operation will retrieve any tuple matching the previous template with `Age` set to 32 and `Eye_color` set as GRAY. If no tuple in the tuple space matches the template in an `in` operation the process will be blocked until a matching tuple is available. The syntax of Linda vary, in some implementations ‘?’ is replaced by either a keyword **formal**, or by stating the type of the variable, i.e. `int`.

The atomic nature of out and in operations supplies an implicit synchronization feature that makes Linda a very simple model for parallel programming, i.e., a bag of tasks approach to parallel programming can easily be implemented by placing tasks in the tuple space. The atomic nature of `in` and `out` comes at the cost of requiring sequential consistency for the tuple space, which again makes Linda mostly suitable for coarse-grained parallelism. Linda exist in many implementations, with variations to both the basic model and to the language that Linda is included in [Kermarrec 94] [Kielmann 97]

### 2.10.2 Global Arrays

Global Arrays [Nieplocha 94], GA, is an attempt at a model that targets distributed data structures for linear algebra operations. GA arrays can be implicitly or explicitly distributed over any number of nodes, most often the number of nodes is very high, i.e., 64-2048 CPUs. The arrays that are distributed are assumed to be quite large. GA then supplies several operations in addition to creation and deletion of arrays, these operations include standard linear algebra operations such as multiplication and dot products. Other features are locking access to a local block of an array to allow the node to access this data directly and thus more efficiently. In all Global Arrays consist of more than 70 functions that may be called by a programmer. Because the semantics of the linear algebra operations is fixed and known, GAs can use a very relaxed memory consistency model. This means that the blocks in the array are generally not synchronized. If parts of an array are required to be synchronized the GA model supplies mechanisms for both synchronization and atomic operations on a remote array block. The GA model is designed with scientific computations in mind, e.g., FORTRAN applications, and is not feasible for applications outside this domain. Global Arrays has become a quite popular choice for applications that

fits the model, and supports a large set of platforms, from the Cray T3E to clusters of PC's running UNIX or Windows NT.

### 2.10.3 JavaSpaces

JavaSpaces [Sun 98] is very similar to Linda, but while Linda supports a set of languages JavaSpaces is closely associated with Java. JavaSpaces supplies programmers with mechanisms to add objects to a JavaSpace or read or retract objects from the space. Instead of working with tuples, like Linda, JavaSpaces work on objects that may contain code in addition to data. Objects in JavaSpaces are strongly typed and may be pattern matched just as tuples in Linda. JavaSpaces supply the programmer with four simple operations to access the JavaSpace:

- **Write**, writes an object into a JavaSpace
- **Read**, gets a copy of an object in a JavaSpace
- **Take**, retrieves an object from a JavaSpace
- **Notify**, attempts to notify an object that an object of a specified type has been written to the JavaSpace

*Figure 9 JavaSpace operations*

A noteworthy difference between JavaSpaces and Linda, apart from the fact that JavaSpaces works on generic objects and not tuples, is that Read and Take in JavaSpaces don't block. If for instance no object matches the template of a Take operation a NULL answer will be returned, while Linda will block until a matching tuple is made available. Instead of blocking input, JavaSpaces allows processes to request a notification once an object that matches a template is available, these notifications may be associated with an expiration time, after which they should not be delivered. In any case JavaSpaces does not guarantee notification delivery, but only states that the best possible effort will be made to deliver the notification. The lack of guaranties are due to the fact that JavaSpaces is intended for work on widely distributed applications as well a local area networks and clusters.

# CHAPTER 3 DSM sub-problems

## 3.1 Distribution Aspects

There are several ways in which one can distribute data in a distributed shared memory system, varying in complexity of implementation and in efficiency of execution. As one may expect, there is a relationship between complexity and efficiency. The following is a brief description of the basic concepts of data distribution.

### 3.1.1 Fixed location

The simplest way to implement distributed shared memory is to assign a fixed location to each data-block at compile or load time. Access to the data is then done in one of two ways, either as an ordinary variable if the data is located at the accessing node, or as a remote procedure call, RPC [Birrell 84], if the data is located on a different node than the one which, performs the access.

The fixed location principle is extremely easy to implement, and requires very little of the runtime system. Naturally the fixed position system is not very efficient if the data is often accessed from remote locations. If however, a high degree of locality is present, and the data is only rarely accessed from a remote node, fixed location allows the DSM system to be implemented rapidly. Debugging fixed position DSM systems is also much easier than debugging more complex DSM system. If a high degree of locality can't be guaranteed a more advanced data distribution model must be developed.

### 3.1.2 Migration

An obvious alternative to fixed location distributed shared memory is to migrate data-blocks as they become needed. This scheme was first implemented in IVY [Li 88]. All data are accessed as is they were local. If the data is not present at the accessing node the access causes a software or hardware trap and the data-block is migrated from its current position to the accessing node, at which point the access is completed and the program continues.

Data-block migration removes the need for locality from start to finish, as with fixed position DSM, however, some degree of both temporal locality and spatial locality is required. If the system does not exhibit temporal locality the data-blocks will migrate constantly between nodes and cause thrashing of the network as previously described. In the event that spatial locality is not present, the problem of false sharing can occur, which as described, also causes network trashing. Pure migration schemes as in IVY have proven to be rather inefficient, as data-blocks that are often migrated causes network trashing, and a high degree of idle nodes due to locking access to data.

### 3.1.3 Read Replication

When the reasons for the poor performance of migration-based DSM were examined it was found that much of the migration of data was due to reading the data that was processed, i.e. the problem data. These data have the access nature that they are never, or rarely, written to. Thus read-only replication of data-blocks was introduced. The philosophy is rather simple, and very similar to the copy-on-write policy used for process replication in many operating systems. Data-blocks are marked read-only, and because of this they can't change, and thus can be migrated arbitrarily without concern for memory coherence. The problems occur when a node wishes to write to a read-only data-block. The runtime environment must then trap the write and then invalidate all read replicas, before the write can be allowed to execute. The invalidation will inhibit the other nodes from reading the data-block before they have obtained a new read replica. Because of this it is of interest to change the nature of the data-block at the correct time, i.e., when the least CPU cycles will be spent waiting for access to the data-block. The nature of this problem is the same as with optimal scheduling and cannot be solved, only approximated.

### 3.1.4 Full Replication

Just as some data-blocks are read-only by nature, some data-blocks behave as write-only for most of their life-span, e.g. result matrixes that are written often and only read when propagating the results to output of some sort. The programmer can often identify these write-only data-blocks, and can then be optimized by dividing a data-block into smaller logical blocks and distribute these to minimize migration. Alternatively, full replication can be incorporated into the DSM system. Where read replicated systems have data-block in either read-only or read-write mode, full replication adds a third mode, write-only.

Write-only replication performs in two steps, when the data-block enters write-only state a copy of it is made, all nodes that receive a write only copy of the data-block also make a copy of the block so that they each have two copies, an original and a work-copy. Once the data-block is read again every node that holds a replica invalidates its replica, it then compares it's work copy with the original, and sends the difference, called a *diff*, to the node who is maintaining the data-block, i.e. the node that distributed the write-only replica's. This node in turn collects all the *diffs* and checks if two nodes have changes the same byte. If two nodes have written the same address, a writing error is raised and no coherency is guaranteed, otherwise the diffs are all applied to the original copy of the data-block, and the updated data-block is migrated to the node that requested read access. While this approach may seem very useful the performance gained from allowing concurrent writes has been shown to be rather small [Keleher 96].

### 3.1.5 Comparing Distribution Models

The different distribution models presented in this chapter all represent a tradeoff between complexity and potential size of the DSM system. It is obvious that fixed location or pure migration is not efficient for systems with more than a few nodes, on the other hand it is also evident that the added scalability that comes with full replication, comes at the price of a significant increase in complexity.

An approximation of the cost of an average access to the four models in given in [Stumm 90]:

Central Server: $C_c = (1 - \frac{1}{S}) \cdot 4p$

Migration: $C_m = f \cdot (2P + 4p)$

Read replication: $C_{rr} = f' \cdot [2P + 4p + \frac{Sp}{r+1}]$

Full replication: $C_{fr} = \frac{1}{r+1} \cdot (S+2)p$

Where:

| | |
|---|---|
| *p:* | The cost of a packet event, i.e. total cost of sending or receiving a package. |
| *P:* | The cost of sending or receiving a data block, i.e. same as *p* plus the added cost for the actual data size. |
| *S:* | The number of nodes (Sites) in the DSM system. |
| *r:* | Read/Write ratio, e.g. *r* reads for each write. |
| *f:* | Probability of an access fault on non-replicated data. |
| *f':* | Probability of an access fault on replicated data. |

*Figure 10 Cost of communication with various data distribution models*

Below the distribution models are placed relative to each other when comparing potential system size and added complexity.

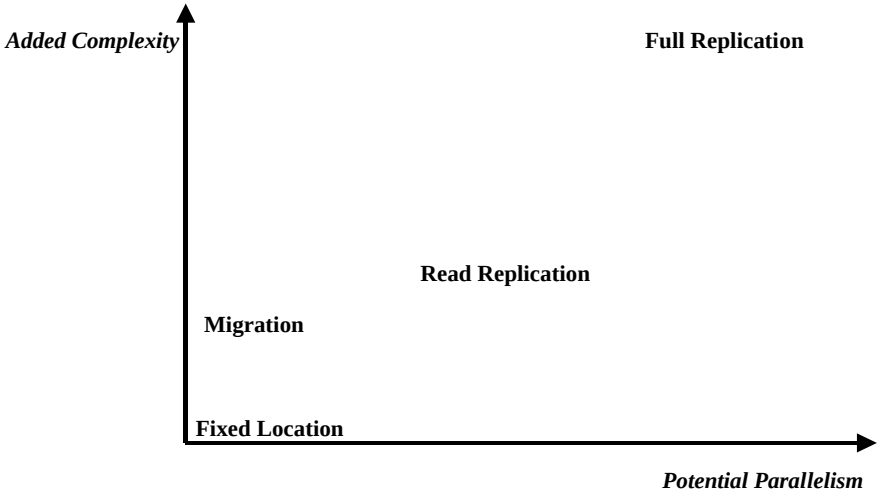


*Figure 11 Complexity of distribution method relative to potential size of the DSM system.*

## 3.2 Data location

Once distribution is added into an application the concept of data location becomes crucial to the functionality of this application. With distributed shared memory this problem is no smaller than with custom distributed applications. The efficiency of replication and migration of data-blocks in DSM systems is highly dependent on locating required data with little effort. The following is a brief description of the most common approaches for handling data location.

### 3.2.1 Central Server

Most often the trivial solution to a distributed problem is to centralize the job. This approach can be used for data location also. The system designates one node to keep track of all data-blocks in use, and when a node requires ownership of a block the node sends a request to the central server. The central server can then do one of two things; it either returns the location of the block to the requesting node, which can then obtain ownership directly from the current owner. Alternatively the server forwards the request to current owner of the block which then transfers ownership to the requesting node. Under ideal circumstances the latter solution saves one message over the first, which would indicate that one would prefer the latter to the former. However, when things go wrong and messages are lost, the central server needs to ensure that all requests are met, which again puts more strain on the server.

This, in essence, is the main problem of the central server approach. As the DSM system grows the workload of the central server grows linearly. If the system supports read replication of data-blocks, a request for exclusive access amplifies the previous problem. Either the server has to invalidate all the replicas and then pass the requesting node a copy for writing, or the server must send the requesting node a list of all nodes that have a copy of the data-block. Generally one can expect more replicas of each data-block when the system grows, so that for large systems the central server approach is not well suited and more complex algorithms must be derived, but for small systems the central server is both simple and fast.

### 3.2.2 Distributed servers

Once the central server becomes a bottleneck in the system the intuitive solution is to add more servers. This however only pushes the existing problem down one level, and becomes the problem of which server to contact in order to find the location of the data-block. The simplest solution to this problem is to have static assignment between blocks and servers. If a number identifies data-blocks, as most often is the case, the lower bits in that number can be used to assign a server, i.e. using the lower three bits one could name eight block-servers.

The problem of invalidating read replicas of a data-block remains the same as in the centralized server even though it also benefits from the added number of servers. The

distributed servers solution is simple and scales well if the distribution of blocks is uniform. If the block distribution is not uniform, or if the access to the blocks is not uniformly distributed, one server can, once again, end up as the bottleneck in the system.

### 3.2.3 Dynamic Distributed Servers

The extreme case for distributed servers is to have one server for each node in the system, this however returns the problem to its origin, as the problem of locating the correct server becomes as complex as locating the node where the data-block is placed [Li 89]. Broadcasting every block-request may solve the problem. If the interconnect does not support a broadcasting mechanism however, this becomes inherently expensive. Even if broadcasting is supported by the interconnect, frequent requests will interrupt all nodes in the system and this is also costly, especially with large systems.

Alternatively each node can try to keep a log of the last known location of a data-block. Any requests to that block will then be sent to the server at the node of the last known position of the block. If the block has moved, this request is forwarded to the new owner of the block, and both the previous and the current owner of the block will set the requesting node as the location of the block. Under the worst circumstances this scheme produces a high number of messages on the network, but under typical conditions the hit rate will be rather high.

The problem of invalidating read replicas might become much larger using distributed servers, depending on how read replicas are obtained. Read replicas may be handled by any node holding a replica of the data-block, or only by the owner of the data-block. If only the owner can distribute read replicas the problems from the distributed server method are the same. If any node holding a replica may distribute replicas, this improves the likelihood of a request for a read replica arriving at a node that can service the request, unfortunately this complicates the invalidation of read replicas. If there exists no central list that holds the position of all read replicas, the invalidation can be handled in two ways; either write requests may only go to the original owner of the data-block, and locating this in turn may be complex. Alternatively any node holding a read replica can service write requests by first invalidating all the replica's it has distributed, and then send the invalidation to the node that in turn gave it its replica. This is then repeated until the node that originally held the data-block has invalidated all the replicas it distributed, at this point the original owner can transfer ownership to the node that requested the write access.

### 3.2.4 Home Based distribution

Home based distribution [Zhou 96] is very similar to the distributed server method. In home based distribution, as in distributed servers, all data-blocks have a specific origin to which all requests for access is routed. However, whereas distributed servers only keep track of the location of the data-block, a home node maintains a coherent version of the data-blocks location, and all other copies of the data-block are replicas of that home version. Depending on which consistency model is used, the home node will collect

changes to the data-block and distribute coherent versions as needed. Invalidation of read replicas in home based distribution is as simple as with the distributed server method, as all read replicas per definition are distributed from the home node which in turn will know the location of all replicas.

The mapping between data-blocks and home nodes can be implemented in various ways, including static mapping as in a distributed server. More commonly the location of the data-block is kept in a data-block table that constitutes a shared virtual address space, i.e. a page table [Samanta 98].

### 3.2.5 Directory based distribution

Directory based distribution is especially suited for architectures where the nodes are segmented in some way so that the cost of sending messages to arbitrary nodes is not constant. Directory based distribution is very common in hardware based DSM systems, such as the Stanford Directory Architecture for Shared memory (DASH)[Leonski 89], and the MIT Alewife. Just as with the home based approach every data-block in a directory based system has a home node. Rather than every node knowing where a given data-block is located, the nodes are segmented into directories, and a node that requests a data-block passes that request to the local directory server. There is no limit to the number of levels of directories that may exist, and thus a set of segments of nodes may join together in a super-segment with a shared super-directory server, and super-segments again may be assembled in segments and so forth, an example of this approach is ASURA [Mori 93].

A directory server is responsible for two things. If a request arrives for a data-block whose location the directory server does not know, it must pass the request to its super directory server. When a request arrives that the directory server knows of it must do two things, mark a relation between the requester, which may be a node or a directory server, and pass the request to the corresponding directory server or node. Invalidation of read replica's are handled in the exact same way as any other request. The advantage of this approach appears in very large systems where the distance between perimeter nodes may be long. When requesting or changing a data-block, any directory server or node only has to send a few messages to the directories and nodes that is directly coupled to it and who has marked a relation with the data-block. In many hardware-based directory-orientated systems a shared bus that allows snooping connects the nodes and directory servers. This way the system can implement coherency at the byte level, by adding a status field to the relation status already described. This has proven to be a very efficient way to build hardware based CC-NUMA machines.

### 3.2.6 Comparing Location Models

The different data location models each represent a solution to data location for DSM systems of different size and nature. The most interesting part is probably the relationship between location model and the potential size of the system. There is no reason to use directory based distribution if the system consists of only a few nodes, similarly a central

server can only service a small number of nodes so that for large systems directory based data location is the optimal solution. The relationship between location model complexity and potential system size is sketched in Figure 12.

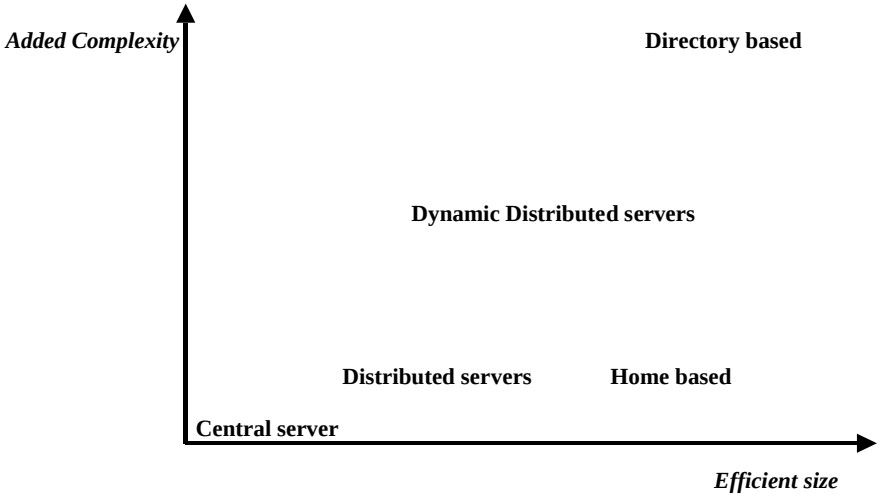


*Figure 12 Data location complexity vs. system size.*

## 3.3 Consistency Models

Distributed shared memory implementations often suffer from poor performance as a result of frequent synchronization problems [Carter 96], or from more hidden problems, such as the previously discussed thrashing and false sharing. These performance problems are most often addressed by introducing a memory consistency model that is more relaxed than the memory consistency model a programmer usually relates to [Petersen 93]. This section takes on some of the consistency models that are used in DSM systems. Some of the consistency models only differ in subtle ways, but the differences may prove essential when executing [Adve 96].

To ease the reading of this section the following notations will be used, W(x)*n* will mean write the value *n* into the variable x. R(x) will mean read the value of x and R(x)*n* will mean read the value of x as *n*. $P_n$ will refer to processor *n*. In all figures the variables are assumed to have the value zero at the beginning of the run.

### 3.3.1 Strict Consistency

When programming a single CPU system the memory model defined by most CPUs is **strict consistency**. Strict consistency is defined as:

> *Any read to a memory location x returns the value stored by the most recent write to x.*

E.g. after a W(x)1 a following R(x) will return 1 no matter how close the read follows the write. Even though this consistency model is the strongest possible, it is of little interest when considering more than one CPU as strict consistency can't be implemented in systems with more than one CPU. This can be shown simply by considering the definition of strict consistency; if W(x)1 happens at time $n$ then at time $n + \varepsilon$ R(x) must return 1. It is evident that no matter how close we place the CPUs we can choose an $\varepsilon$ so that the write to x cannot be propagated from CPU one to CPU two when the signal speed is limited by the speed of light[1]. Another reason why strict consistency can't be used in distributed systems is the fact that $n$ refers to an absolute global clock which cannot be maintained in a distributed system. Surprisingly, strict consistency would be of little or no value in a system with more than one CPU, as any program that relies on strict consistency between two processes/threads would necessarily contain a race condition and would be unreliable by nature [Tanenbaum 95]. We include strict consistency here because it is the consistency model from which all others will be relaxed.

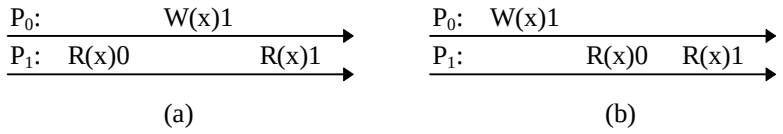


*Figure 13 Strict consistency with the line being a global time line (a) complies with strict consistency whereas (b) does not.*

## 3.3.2 Sequential Consistency

As it is agreed that programs that require strict consistency must contain some level of race-condition prone code, we can assume that most programs will adopt some sort of synchronization mechanism to avoid race conditions. The nature of synchronizations gives us an extra semantic knowledge of the programs that comply with the rules of race condition free code. This semantic information can be used to relax the strict consistency model into the model called **sequential consistency**. Lamport [Lamport 79] defined sequential consistency in 1979 as :

> *[A multiprocessor system is sequentially consistent if] the result of any execution is the same as if the operations of all the processors were executed in some sequential order, and the operations of each individual processor appears in this sequence in the order specified by its program.*

The definition has two main components, first of all any two processes must see any two results in the same order, and any process must perform its operations in the order stated in the original program. An investigation of the practical aspects of this definition can be

[1] In fact in a system based on electric signals the propagation speed would be 2/3 of the speed of light.

found in [Adve 91]. The latter seems trivial but is an actual concern today, where all modern CPUs utilize some degree of 'out of order execution'. Some modern CPUs, such as the Intel Pentium Pro, preserve the sequential order by delaying reading and writing. Others choose to relax the memory consistency even further, and thus do not comply to sequential consistency, e.g. the Compaq Alpha and the Sun Sparc V.8 and V.9.

The first part of the definition states that all operations were executed in some sequential order. This means that the result of any set of instructions will be available to all processors in the same order.

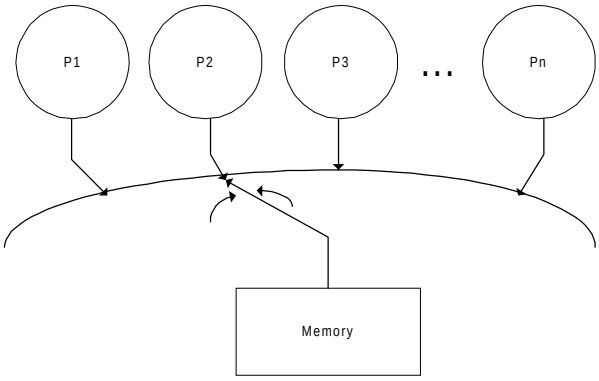


*Figure 14 Sequential consistency may be viewed as the processors gaining access to memory one at a time.*

It is important to notice that the sequential consistency model does not define any time boundary within which an operation must be visible, only that once it becomes visible it is visible to all processors. On the figure this means that access to memory is not guaranteed at any point. If we look at the figure from the section on strict consistency where the latter execution was invalid, under sequential consistency both executions now become valid.

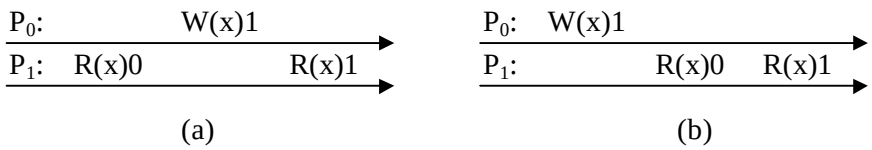


*Figure 15 Under sequential consistency both (a) and (b) are now valid executions.*

To violate sequential consistency a system must break either the clause of same order visibility or the clause on maintaining program order. An example of breaking the visibility clause could look as this:

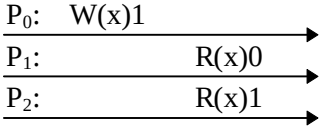


*Figure 16 An invalid run under sequential consistency, violating the visibility clause.*

In Figure 16 $P_1$ reads x as zero while at the same time $P_2$ reads x as one. The importance here is that even though no measure of global real time is available to the processes it is still invalid to read different values of the same variable at the same real time. Violating program order could look something like this:

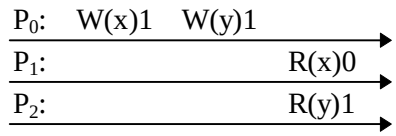


*Figure 17 An invalid run under sequential consistency, violating the program order clause.*

The run in Figure 17 violates the program order because at the same time as $P_1$ reads x as zero $P_2$ reads y as one. This is a violation because the write to x precedes the write to y. This breaks the program order as W(x)1 must be visible before W(y).

Sequential consistency introduces very little extra complexity for the programmer, relative to strict consistency, yet whereas strict consistency is impossible to implement,, sequential consistency is trivial to implement[2]. Sequential consistency has its problems too. Lipton and Sandberg [Lipton 88] have shown that if *r* denotes the read latency, *w* the write latency and *t* the minimum inter-processor communication time, then $r + w \geq t$. That means that any optimization in read latency will worsen the write latency and vice-versa, once the critical time is achieved. With state-of-the-art interconnects yielding latencies of 1-5 µs, a DSM system based on sequential consistency may suffer a severe performance problem, thus an even more relaxed consistency model must be derived.

### 3.3.3 Causal Consistency

As the sum of all vices is constant under sequential consistency, this model must be improved. The immediate improvement is called **Causal Consistency** [Mohindra 94], and as may be derived from the name, works with the causality between operations. The nature of the model is that no operation may be seen before any operation that causally precedes it, if there are no causal dependency between two operations then no restrictions are made to their visibility. Tanenbaum defines causal consistency as:

> *Writes that are potentially causally related must be seen by all processes in the same order. Concurrent writes may be seen in a different order on different machines.*

An execution that complies with causal consistency could look as:

[2] This is to say that to implement the functionality of sequential consistency is easy, efficiency in sequential consistent systems is another matter altogether.

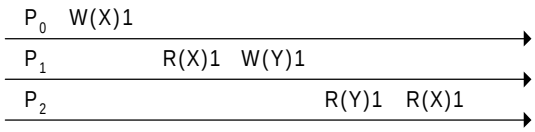


*Figure 18 A valid run under causal consistency.*

Where W(Y) may depend on the preceding R(X) and thus on the W(X), therefore when $P_2$ reads Y as the updated value it must read the updated value of X also, thus an execution that compromises causal consistency could look as:

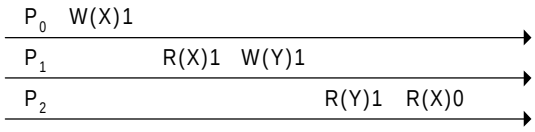


*Figure 19 An invalid run under causal consistency, the R(X)0 cannot come after R(Y)1.*

The causal consistency seems to solve all problems for DSM systems as updates are only made when necessary. Unfortunately, keeping track of causal dependencies requires the system to keep track of which nodes has seen which writes. This expands typically to a full dependency graph, which is costly to implement. The conclusion on causal consistency is that this it not efficient enough to support an efficient implementation of DSM.

### 3.3.4 PRAM Consistency

As causal consistency fails to meet our requirements due to the maintenance of the global dependency graph the intuitive approach it to weaken causal consistency in a way that eliminates the need for the global dependency graph. This consistency model is called Pipelined RAM, commonly noted **PRAM [**Lipton 88]. PRAM is defined as:

> *Writes done by a single process are received by all other processes in the order in which they were issued, but writes from different processes may be seen in a different order by different processes.*

As the name indicates, the PRAM model may be viewed as if all writes to memory from a processor are pipelined.

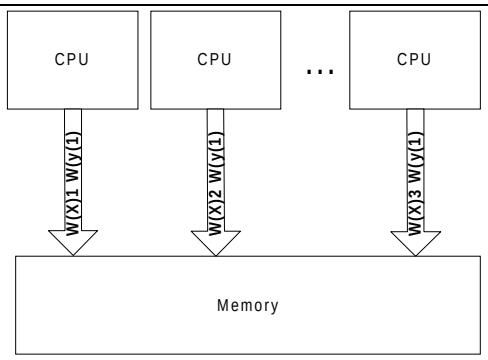


*Figure 20 Memory as seen from a PRAM consistent system.*

The resulting behavior differs from causal consistency in that whereas any two writes from a processor must be visible in order, writes from different processors may be mixed without constraints. This relaxes the consistency even more and the trace that conflicted with causal consistency is now a valid run.

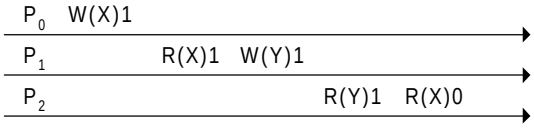


*Figure 21 An invalid run under causal consistency, however this run is valid under PRAM consistency.*

To violate PRAM consistency two writes from the same processor must be viewed out of order, this could look as:

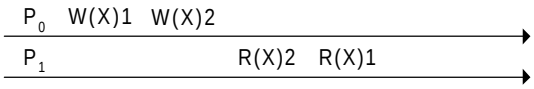


*Figure 22 An invalid run under PRAM consistency, the R(X)1 cannot come after R(X)2.*

The problem with PRAM consistency is that we have now weakened the consistency so much that writing programs under PRAM consistency is much harder than under, i.e. sequential consistency. Adding a single restriction to PRAM may make it more usable.

### 3.3.5 Processor Consistency

Goodman [Goodman 89] challenged the weakness of PRAM consistency. By adding the extra clause to PRAM consistency that all writes to a single location should be agreed upon by all processors, whereas writes to different locations need not be agreed upon. The slightly stronger model is called **Processor consistency** [Ahamad 93]. Processor consistency differs only slightly from PRAM consistency, actually the models are so similar that they are often mistaken for the same. Processor consistency may be defined as [Tanenbaum 95]:

1. *Before a read is allowed to perform with respect to any other processor, all previous reads must be performed.*

2. *Before a write is allowed to perform with respect to any other processor, all other accesses (reads and writes) must be performed.*

### 3.3.6 Weak Consistency

The improvements that were made from sequential consistency to PRAM consistency all derived from the fact that no programmer should program code that contains race conditions, and thus should use synchronizations in the code. These synchronizations were used implicitly to form the behavior we expect from a correctly execution. A natural extension to this concept is to use explicit synchronization variables. The idea is that when a synchronization is performed all writes since the last synchronization are propagated. This model is called **Weak-consistency** [Adve 90]. Weak consistency is defined as:

1. *Accesses to synchronization variables are sequentially consistent.*

2. *No access to a synchronization variable is allowed to be performed until all previous writes have completed everywhere.*

3. *No data access ( read or write ) is allowed to be performed until all previous accesses to synchronization variables have been performed.*

This means that synchronization variables are treated differently from ordinary variables, specifically synchronization variables are treated with sequential consistency. The second requirement says that the memory is locked, e.g. Figure 14, to the processor doing the synchronization until this is finished. This means that when a synchronization is finished all outstanding writes from the synchronizing node are completed, and no other nodes are allowed to perform writes. The finale clause says that no data access can be performed from a node as long as it has an outstanding synchronization, this means that a read following a synchronization can be sure to return the latest written value.

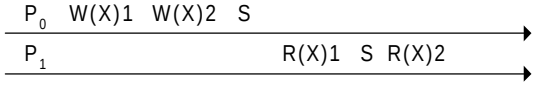


*Figure 23 A valid run under weak consistency, P1's first R(X) may return 1 or 2, whereas the second R(X) must return 2.*

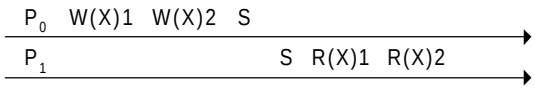


*Figure 24 An invalid run under weak consistency, P1's first R(X) must return 2 as it follows a synchronization.*

The obvious advantage of weak consistency over, i.e., PRAM consistency is the differentiating between synchronization variables and data variables. Whereas this may seem an odd extension to most programmers, it is quite natural to any programmer that writes parallel/concurrent code. When writing operating system code, a tight polling loop might often look alike:

```
while(S==0);
```

*Example 6 Polling loop.*

Anybody that has tried this knows that any decent C-compiler will place the variable S in a CPU register because it recognizes a tight loop and running the condition variable in a register is a major speed advantage. Thus the above code would not work if we expect `S` to be changed by another process/processor, unless `S` has been defined as `volatile`. So even if a separation of synchronization variables and data variables seems awkward it is something that parallel application programmers are familiar with already.

Whereas weak consistency can make a significant improvement to the speed of DSM systems the consistency model can be weakened even more for even more speed.

### 3.3.7 Release Consistency

Weak consistency gets its performance from identifying synchronization variables, and thus isolating critical sections. What weak consistency does not do is to determine when a critical section is entered and when it is left. Because of this weak consistency must update both local writes and remote writes at each synchronization. If the type of synchronization could be determined, the system could reduce the update to the necessary type. This approach is called **Release consistency [**Dwarkadas 93**]** [Neves 94]. Release consistency relies on the programmer to use explicit *acquire* and *release* operations. Release consistency can be identified as:

1. *Before an ordinary access to a shared variable is performed, all previous acquires done by the process must have completed successfully.*

2. *Before a release is allowed to be performed, all previous reads and writes done by the process must have completed.*

3. *The acquire and release accesses must be processor consistent.*

The fact that a release consistent system can differ between entering a critical region and leaving it makes it possible for the system to only update shared data in one direction at a time. When an 'acquire' is performed the system will update all local copies of shared data to be consistent with remote versions. When a release is performed all local copies will be propagated to remote nodes.

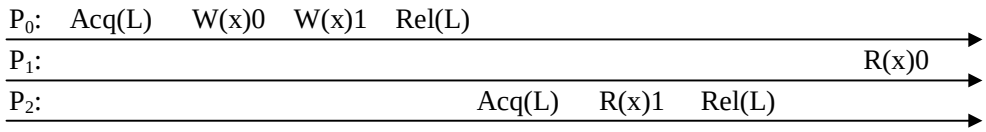


*Figure 25 A valid run under release consistency: As $P_1$ does not use lock variables no result is guaranteed for x.*

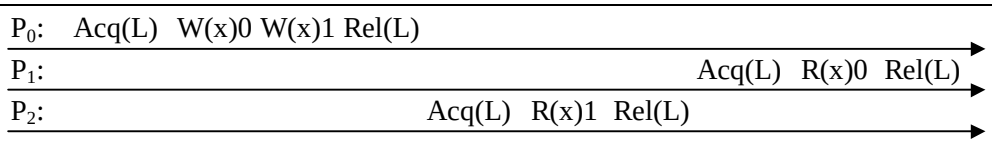


*Figure 26 An invalid run under release consistency, as $P_1$ uses lock variables the value of x should be guarantied.*

### 3.3.8 Lazy Release Consistency

Release consistency has one intuitive shortcoming with respect to reducing the necessary number of messages. When a release is performed all shared data are sent to all nodes that hold copies of those data. However these data will often not be read before the node in question produces another value for the protected variables. The intuitive solution is to avoid sending data as soon as they are ready, but rather wait until an acquire is performed by another node, and then send the updated data. This policy is called **Lazy Release Consistency** [Keleher 92]. Semantically lazy release consistency does not differ from ordinary release consistency, the difference is restricted to the point in the execution when the communication penalty is taken. In many programs lazy release consistency will perform better than standard release consistency, also often called eager release consistency. However, in certain applications eager release consistency will outperform lazy consistency, especially when the shared data are synchronization variables. Lazy release consistency favors shared variables that are used in tight loops.

### 3.3.9 Entry Consistency

The previous consistency models gained some of their efficiency from differentiating between data variables and synchronization variables. It was argued that this is a natural way for programmers to write code anyway, and thus introduces no real additional complexity. This concept can be taken one step further as programmers usually have a synchronization variable for each logical block of data variables. When one of the release consistency models updates data variables, it has to update all modified shared-variables. If

a consistency model requires data variables to be directly linked to synchronization variables, the model can use the explicit relationship between the two and by this reduce the communication. This scheme is called **Entry Consistency** [Bershad 93]. The setup is very similar to lazy release consistency. Once an 'acquire' is performed on a synchronization variable, the acquiring node is required to receive the correct version of the variable(s) associated with the synchronization variable.

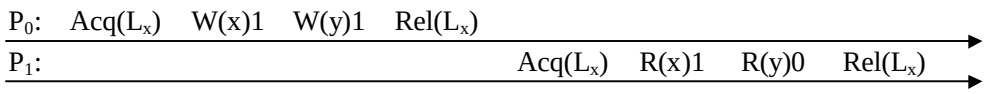


*Figure 27 A valid run under release consistency, $L_x$ is a synchronization variable associated only with x, therefore the value of y is not updated upon acquiring $L_x$.*

Entry consistency also adds a shared state to shared variables. The nature of this is that when a node performs an acquire, the lock can be set as shared or exclusive. In shared state all nodes with a shared lock on the variable may read the data variable. Exclusive state may only be held by one node at a time. Once a node has obtained exclusive access to a variable this variable may be read or written as required.

Entry consistency is defined as.

1. *An acquire access of a synchronization variable is not allowed to perform with respect to a process until all updates to the guarded shared data have been performed with respect to that process.*

2. *Before an exclusive mode access to a synchronization variable by a process is allowed to perform with respect to that process, no other process may hold the synchronization variable, not even in non-exclusive mode.*

3. *After an exclusive mode access to a synchronization variable has been performed, any other process' next non-exclusive mode access to that synchronization variable may not be performed until it has been performed with respect to that variable's owner.*

### 3.3.10 Scope Consistency

Entry consistency is a very efficient approach to relaxed memory consistency, and is able to achieve near optimal solutions to memory updates. Unfortunately the explicit relationship between synchronization variables and shared data makes Entry Consistency much harder to program than any of the other consistency models that has been covered. An attempt to address this problem is Scope Consistency, ScC [Iftode 96]. The idea behind Scope Consistency is that correctly written parallel program expose a scope within which

data is supposed to be coherent, i.e. shared data is protected by a critical section which is expressed directly by the access to synchronization variables. If for instance a variable *x* is updated within a critical section then the new value of *x* is not expected to be available by the other processes in the system, before the process inside the critical section has left the critical section. A system can therefore define a consistency scope from the enter and leave critical sections in the source code, or scopes can be explicitly defined inside the code.

To achieve a comprehensible consistency model, ScC must define a way to discriminate between a reference being performed with respect to a consistency scope and reference being performed with respect to a process [Iftode 96]:

> *A write that occurs in a consistency scope is **performed with respect to that scope** when the current session of that scope closes.*

In order to conform to Scope Consistency a system must therefore comply with two rules:

> *1. Before a new session of a consistency scope is allowed to open at process P, any write previously performed with respect to that consistency scope must be performed with respect to P.*
>
> *2. A memory access is allowed to perform with respect to a process P only after all consistency scope sessions previously entered by P (in program order) have been successfully opened.*

### 3.3.11 Automatic Update

Release consistency and lazy release consistency are both good at reducing the number of messages that have to be sent to maintain memory consistency. But as lazy release consistency targets the situations where the same variable is accessed often by the same node, the reverse situation can occur, i.e., a variable that is often used by two or more nodes, effectively resulting in trashing the network with messages. The Automatic Update Release Consistency model [Iftode 96] has targeted this problem. The goal is that by adding a hardware snooping device on the memory bus, writes can be detected as they are performed and propagated to other nodes that are using the data. The idea is that data propagation can be cost-free with the use of special hardware that distributes the data without requiring the CPU to do any work. Automatic update consistency has the same semantics as lazy release consistency, and adding:

> *Before performing a release, all automatic updates must be performed.*

### 3.3.12 Consistency model summary

The previous described consistency models may be seen as a contract between the programmer and the system, a contract that guarantees a given behavior in return for a

certain set of rules that are followed by the programmer. It is obvious that the more rules the programmer complies to, the more efficient the system can run. On the other hand, for every extra rule the programmer must follow, DSM systems move further away from the desired illusion that the programmer is programming an ordinary shared memory system. Because of this one must either find a comfortable balance between restrictions and speed, or provide tools, i.e., programming languages that takes over some of the burden of understanding the consistency model. Research in the field of consistency models is still active, and the ultimate consistency model for DSM systems has yet not been established. It is very likely that we will see several new consistency models appear in the next few years. Figure 28 attempts to show the relationship between added semantics in consistency models and potential size of the DSM system.

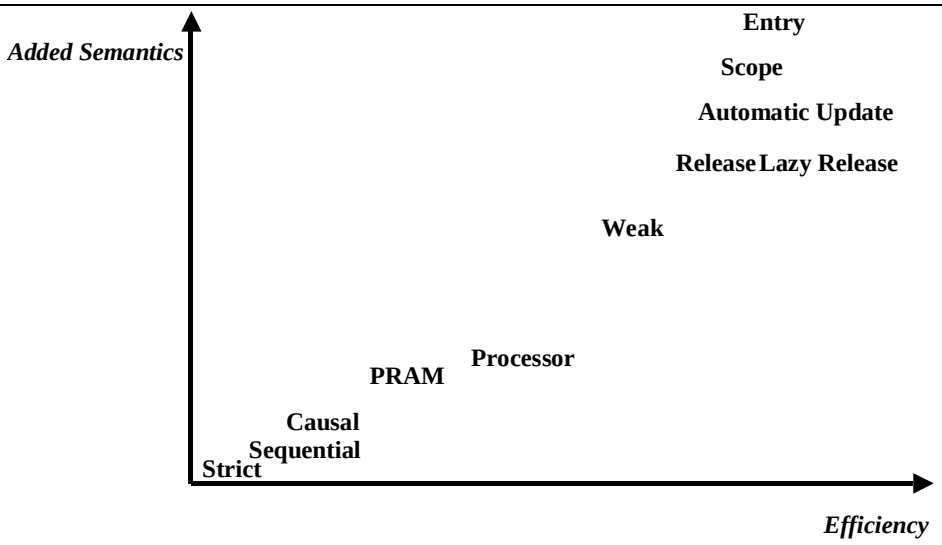


*Figure 28 The consistency model relationship between the added semantics the programmer must consider and the potential size of the system.*

This chapter has focused on identifying the rules for the various consistency models, an interesting study into the behavior of the different models, and how they may be implemented can be found in [Weiwu 98]. A DSM system may support more than one consistency model, such as CLOUDS [Cornilleau 95]. Memory models that include several consistency models have also been proposed, e.g. data-race-free-0 and data-race-free-1 [Adve 93].