Virtual Shared Memory: A Survey of Techniques and Systems

Sanjay Raina*
Dept. of Computer Science
University of Bristol
Bristol BS8 1TR, U.K.

Abstract

Shared memory abstraction on distributed memory hardware has become very popular recently. The abstraction can be provided at various levels in the architecture e.g. hardware, software, employing special mechanisms to maintain coherence of data. In this paper we present a survey of basic techniques and review a large number of architectures that provide such an abstraction. We also propose new terminology which is more consistent and orderly as compared with the existing use of terminology for such architectures.

1 Introduction

Virtual Shared Memory (VSM) in its most general sense refers to a provision of a shared address space on distributed memory hardware. Such architectures contain no physically shared memory. Instead the distributed local memories collectively provide a virtual address space shared by all the processors. VSM combines the benefits of the ease of programming found in shared-memory multiprocessors with the scalability of message-passing multiprocessors. The implementations of VSM vary from software-based schemes, integrated into the operating system, to hardware-based schemes which employ conventional caching algorithms. In this paper we will review the basic techniques used in the implementation of VSM architectures and present a survey of a large number of systems in existence or under development. A number of papers have recently presented surveys of VSM architectures including [89, 66, 87, 65]. The survey described herein is more comprehensive and elaborate and attempts to organise the design space of such architectures.

2 Shared-memory abstractions

Besides being referred to as VSM, such architectures have also been called by different names by different people such as Distributed Shared Memory (DSM), Shared Virtual Memory (SVM), Distributed Shared Virtual Memory (DVSM) and so on. Such arbitrary use of terms, in our opinion, can cause confusion amongst the readers. So, we shall classify such architectures into three different types of organisation and use distinct terminology for each as follows:

Virtual Shared Memory (VSM): such systems generally use hardware assistance (similar to hardware cache-coherence) to provide a coherent shared address space. The unit of sharing is smaller (of the order of a cache-line) and virtual memory (in the conventional sense) can be implemented on top of this single implicitly shared address space. Examples of such architectures include the scalable cache-coherent architectures like the DASH and DDM.

*Seconded from Meiko Limited, Aztec West, Bristol, UK
Shared Virtual Memory (SVM): in contrast to the above, such systems generally implement shared memory on top of per-node virtual memory. The unit of sharing is larger, generally a page, and there is no hardware assistance for coherency. Instead the shared address space is managed by specialised (kernel) page-fault handlers and servers. Examples include IVY, KOAN etc.

Distributed Shared Memory (DSM): such systems include NUMA or "dancehall" architectures that provide the ability to access remotely shared data but no replication (and thus no coherence). The data is always at a fixed locus from its computation. Examples include Butterfly, RP3 etc.

Figure 1 shows this classification. We shall only discuss the first two classes because of the lack of coherence in DSM systems.

A better solution to the problem of inconsistent terminology would be to develop a more objective taxonomy similar to the one invented by Flynn [35] to classify (multi) processor architectures.
We propose one such taxonomy as follows. Just as Flynn based his classification on the two fundamental activities in a computer system, we use the address space and the physical memory organisation as the basic entities that describe a memory system and its relation to the processor(s). We designate the address space to be either shared or disjoint. Likewise we designate the physical memory to be either shared or distributed. We can now classify memory systems into four different classes as: shared address-space, shared memory (SASM) which corresponds to the conventional shared memory systems; disjoint address-space, distributed memory (DADM) which corresponds to the conventional distributed memory machines; disjoint address-space, shared memory (DASM) which although possible (e.g. RPC on a shared memory machine) is not useful; and finally, shared address-space, distributed memory (SADM) systems which describes VSM, DSM, SVM etc. systems more succinctly. Part of the problem in existing terminology lies in the use (rather abuse) or the word "virtual" which has no clear interpretation. Hence we have avoided the use of virtual and used a terminology which captures the layout of memory and address spaces instead. Given such a terminology, we can subclassify the SADM category depending upon the way of realisation of a shared address space on distributed memory. The first category SADM-NUMA refers to the NUMA architectures in this class. The next category SADM-OS refers to the abstraction of shared data at the operating system level. The third category SADM-CC refers to the cache-coherent shared data architectures and the last category, SADM-COMA refers to the Cache Only Memory Architectures (COMA) like the Data Diffusion Machine or the Kendal Square Research KSR1. Table 1 summarises our classification by giving example architectures from each class.

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>SASM</td>
<td>Sequent Symmetry etc.</td>
</tr>
<tr>
<td>DADM</td>
<td>Meiko, Hypercube etc.</td>
</tr>
<tr>
<td>SADM-NUMA</td>
<td>IBM RP3, BBN Butterfly etc.</td>
</tr>
<tr>
<td>SADM-OS</td>
<td>IVY, KOAN etc.</td>
</tr>
<tr>
<td>SADM-CC</td>
<td>DASH, Alewife etc.</td>
</tr>
<tr>
<td>SADM-COMA</td>
<td>DDM, KSR1</td>
</tr>
</tbody>
</table>

Table 1: Memory system taxonomy and example architectures

For the lack of suitable generic term that covers all of the above classes we shall refer to them as VSM architectures in general from now on and qualify wherever necessary.

The roots of VSM lie in the areas of cache-coherency and virtual memory management and, a large body of literature exists in both the fields. The former provides a study of the techniques for maintaining a consistent view of the address space while allowing for multiple copies to exist. The latter is concerned with the study of algorithms and mechanisms that deal with hierarchical storage systems.

Although most systems consider the address space to be an unstructured linear array of words, some structure the address space as objects, language data structures or associative structures. the unit of sharing ranges from a byte to a page. the consistency model provided by a VSM is an important design consideration. Most systems so far provide the strict sequential consistency model but some recent systems provide weaker models. Weaker models reduce synchronisation and increase efficiency. Similar to multiprocessor caching a VSM system consists of a cache coherence protocol. Several cache-coherence protocols have been designed and implemented. In addition, efficient mechanisms for data replacement are also needed. Finally, efficient and scalable mechanisms for synchronisation also form an important part of the system as synchronisation by nature implies congestion and competition for limited network resources.
The idea of combining the benefits of shared memory with those of distributed memory multiprocessing is not entirely new. CM* was one of the first architectures which had memory physically distributed but provided hardware mechanisms for one processor to access a remote processors memory which formed part of the global virtual memory. Since then, although several attempts have been made in this direction, only recently has it attracted widespread interest.

The issues involved in the design of a virtual shared memory architectures include the structure of the address space (flat, segmented or physical and whether the memory is laid out as objects, language types etc.), the cache-coherence protocol (write-update/write-invalidate, snoopy/directory-based etc.) and synchronisation mechanisms (test-and-set, fetch-and-op, queue-based etc.). In the remainder of this paper we will look at each of these issues in detail and then review a variety of architectures that support virtual shared memory.

3 Caching mechanisms

The mechanism of caching is an embodiment of virtual memory where a subset of the data-space is stored closer to the processor than the main memory. The design decisions for caches include the fetch, place and replace algorithms together with the cache size and consistency model supported. The above factors are used to optimise key metrics of cache performance which include maximising the hit ratio and reducing mean cache access time.

3.1 Caches in unprocessors

Caches were first introduced in the IBM 360/85 in 1969 and since then the design and organisation of caches has become increasingly complex. In this section we outline the main characteristics and considerations of cache design. For more details the reader should refer to an in-depth survey by Smith [79] and other similar surveys.

A cache is generally defined as a storage medium located in between the main memory and the CPU. It is usually faster and smaller than the main memory. The main goal of the cache is to provide data to the processor at a much faster rate than the main memory. However, due to the smaller size of the cache, its usefulness is justified so long as the data that the processor requests for is ”almost always” here, i.e. the cache has a high hit-ratio. The effectiveness of a cache-design depends on a variety of (often conflicting) parameters that need to be skillfully balanced for a given requirement [86].

The first design consideration is the fetch policy. Most designs use fetch on demand because of its simplicity, but some use prefetching which can improve the miss-ratio. Three prefetching schemes have been used: always prefetch, fetch on miss and tagged prefetch. Przybylski [69] remarks that complicated fetch strategies perform only marginally better than simple fetch on demand due to limited memory resources and strong temporal clustering of cache misses. The next consideration is the organisation of data items into the cache, i.e. placement. The choices are direct mapping, fully associative or set associative. Fully associative caches are expensive to implement and set-associative caches generally outperform direct-mapped caches [79, 48, 70], although Hill [49] argues for direct-mapped caches, concluding that as the cache size increases their performance gets closer to set-associative caches. A set-associative cache organises itself as a number of sets with a number of elements per set. A hash function maps an address to a set and that set is then associatively searched for the item. Typical values of associativity, i.e. number of elements per set, range from 2 to 16. The next design parameter is the line size, where a line is the basic unit of cache storage. A smaller line size means shorter transfer time but higher miss ratio. Because of the smaller size of
the cache there is the inevitable need to provide an efficient replacement algorithm. Replacement policies include least recently used, working set, random, first-in first-out etc. Updating of memory can be performed in two ways i.e. write-through and copy-back. In the case of former, the main memory always contains an up-to-date copy and the consistency problem is simplified. In the copy-back case the memory traffic is lesser but the logic to support consistency is more complicated. The size of a cache is another criteria. A larger cache size generally implies higher hit ratio and higher cost. Most caches are real indexed, in that they operate on physical addresses. However a cache can be addressed directly with the virtual addresses. The address translation now takes place after the address emerges out from the cache. The Translation Lookaside Buffer (TLB) stores the mapping between the most recently used real and virtual addresses and is organized like a small set-associative memory. In multiprocessors the TLBs should also be maintained consistent in addition to the caches. TLB consistency is of particular concern to operating system designers and has been addressed by several people including Black et al [17] and Teller [91]. Another design choice is the use of a multi-level cache [78, 11]. These further exploit cache-based locality property, by reducing the (first-level) cache-miss times.

3.2 Caches in Multiprocessors - the problem of consistency

Multiprocessor cache design is much more complex than the uniprocessor one primarily because of the consistency problem. Multiple caches means the existence of multiple copies of the same data item and the requirement is to keep all the copies consistent. To maintain consistency both hardware and software mechanisms or protocol have been developed and implemented.

In its most general sense the problem of cache-consistency can be stated as the requirement that a read of a quantum of memory always returns the most recently modified copy of that quantum. Solutions to the problem of cache-coherence are numerous. Some early simple solutions rely on software enforced compiler assisted approaches which restrict the type of data that can be shared. Only read-only and private data is allowed to be cached and the rest is uncachable. The early hardware schemes like [21] were directory-based but with the easy-availability of bus-based tightly-coupled multiprocessors, snoopy schemes have become more popular. However, due to the limited scalability of bus-based systems, interest into directory-based protocols has been revived. The following two sections review the two coherence protocol classes. Either scheme is implemented by associating a state with each of the cached blocks so that when a request to the cache is issued, the coherence apparatus looks up the state of the block and then updates the state before satisfying the request. The main property that distinguishes one protocol from another is the mechanism to handle writes to a shared block, i.e. a replicated block. These broadly fall into write-invalidate or write-update protocols.

The area of cache-coherence has received a great deal of attention in recent years. A number of exhaustive papers have reviewed cache-coherence mechanisms including Smith [79], Hill [50], Stenstrom [84], Chaiken [22] and Archibald [10].

3.2.1 Consistency Models

In a multiprocessing environment the programmer assumes a model or a view of the memory and because of the history of writing sequential programs, such a model is closer to the model of the memory operations seen in a uniprocessing environment. The main characteristic of such a model is that memory accesses are seen in program order and processors execute instructions one by one in program order. This model is known as sequential consistency [32, 37] or the strong coherence model. The other class of weak coherence [1] models relax this restriction. This relaxation allows
instructions to be executed out of order and thereby increases performance. We now review the 
models of consistency that have been described in the literature.

**Sequential Consistency**
Sequential Consistency was first defined by Lamport [56] as the behaviour observed when the 
operations of a multiprocessor were executed in some sequential order and the operations of each 
individual processor appear in this sequence in the order specified by its program. The model seen 
by multiple processes on a time-shared uniprocessor is the best example of a sequentially consistent 
system. It is believed that such a model is too restrictive for multiprocessors in general.

**Processor consistency**
A more relaxed model than sequential consistency is the one proposed by Goodman [39] and is 
called processor consistency. It requires that writes from a processor may not appear in any order 
other than the one in which they were issued. However writes from two processors may appear in 
any order to one of the two or a third processor. He claims that most programs execute correctly 
under processor consistency and points out that many existing multiprocessors like the VAX 8800 is 
processor consistent but not sequentially consistent. Strict event ordering is guaranteed by explicit 
synchronisation annotations from the programmer rather than relying on strict sequential ordering.

**Release Consistency**
Release consistency was proposed by the DASH group [37]. In this model sequential consistency 
is guaranteed only at the time of a synchronisation release and the assumption is that at all other 
times the programmer has guaranteed sequential consistency.

### 3.2.2 Cache-coherence protocols
Several schemes have been proposed for maintaining cache-coherence in multicache systems. We 
review some of these schemes in this section. Cache-coherence schemes can be broadly divided 
into software-based or hardware-based. Software-based protocols are less expensive but require 
more support from the compiler or the programmer (or the run-time system). Hardware-based 
protocols are generally further classified into snoopy and directory-based schemes. Snoopy protocols 
require that each cache controller listens to (or snoops) all cache-consistency traffic on the network. 
Because of simplicity snoopy protocols have been designed and implemented for many bus-based 
multiprocessors (e.g. a write-invalidate protocol in the Sequent Symmetry [59] and a write-update 
protocol in the DEC Firefly [92]). Such multiprocessors however have limited scalability and many 
large scale multiprocessors employ general interconnection networks to overcome the bus-bandwidth 
bottleneck. Implementing snoopy protocols on general interconnect based systems is not a good idea 
since broadcast and snooping are costly operations on such networks. Such systems typically utilise 
the so called directory-based schemes where a directory maintains pointers to the exact location 
of caches that hold copies of a memory block. A survey and evaluation of various directory-based 
schemes appears in [4] and [22].

Directory-based protocols are further divided into full-map, limited-directory, and chained-
directory schemes. Full-map schemes, like those in [21] and [83], minimise network traffic but 
are not scalable in terms of memory overhead (e.g a 100 processor machine would need 100 bits 
for each entry in the directory). An alternative is to reduce the number of pointers to less than 
the actual number of caches and is known as the limited directory scheme (e.g. [19]). The number 
of pointers (and thus the bit-overhead) remains fixed even as the number of processors increases. 
Such schemes are evidently sensitive to the extent of sharing in programs and broadcast is the 
usual recourse when the number of cached-copies exceeds the number of pointers. Finally, the 
last category, i.e. the chained-directory schemes, provide a full map by distributing the directory
amongst caches and then linking together all the caches that have a copy of a memory block. The Scalable Coherence Interface [51], the Stanford Distributed Directory project [93] and the DASH multiprocessor [57] all use this scheme.

**Software schemes**

Because of the increased complexity of hardware solutions, software-based coherence schemes are sometimes preferred. This section reviews some of the software solutions to cache-coherence problem. Most schemes operate on either compile-time (static) or at run-time (dynamic) information. Static schemes are usually implemented as part of the compiler whereas dynamic schemes are usually part of the operating system kernel.

**Smith’s One Time Identifier scheme**

The One Time Identifier scheme presented by Smith [81] is a dynamic scheme and operates as follows. Each shared page has associated with it a unique identifier called the one-time identifier (OTI). An OTI field is added to each cache line and each TLB entry. Whenever the TLB is updated, a new OTI value is written into the OTI field. When a line is accessed for the first time, it is loaded from the main memory and the TLB OTI entry is copied into the cache OTI entry. All subsequent references compare the cache OTI with the TLB OTI. A match results in a hit and otherwise it is a miss. On exiting from a critical region the processor invalidates TLB entries corresponding to the data protected by the critical region. On the next processor reference to shared data, the corresponding TLB entry is loaded again from the OTI register. Thus each subsequent reference to a stale cache line is treated as a miss because the OTI entry pair does not match.

**Cheong’s Version Control scheme**

Viedenbaum and Cheong proposed a static scheme called the Versions Control Scheme [25]. Data dependency between two tasks is represented by a directed graph. Two tasks on the same graph level are independent and can be run on two processors without any coherence check. To move from one level to another a synchronisation action is needed. Each task that writes to a shared variable defines a new "version" of its contents. Each processor takes care of the version of each shared variable in its own memory (current version number CVN). When moving to the next task level, a processor increments the CVN for all variables being written during the execution of the previous level tasks. These actions are performed in accordance with static program analysis. Besides this each cache line also contains a Birth Version Number (BVN) field which is written on each cache allocation. The values of BVN and CVN are compared for every access to a shared variable in the cache. If BVN is smaller than CVN it is a miss, otherwise the access is a hit.

**Snoopy coherence protocols**

These schemes are sometimes called bus-based protocols because they rely on the assumption of the existence of a broadcast medium like a bus to implement snooping. Each cache is able to listen to the transactions put on the bus by any other cache and thus snooping on the requests of other caches, updating the states locally. The state information is thus distributed over the various caches. Snoopy cache protocols can be either write-invalidate or write-update type.

Write-invalidate protocols allow only one writer and multiple readers. A write to a shared copy results in all other copies being invalidated before the write can go ahead. Write-update, instead of
invalidating, updates all copies before the write can go ahead. Some of the snoopy protocols that have been designed in the past are as follows.

**Write-once** [40] has the following states: **INVALID, VALID** - not modified, possibly shared, **RESERVED** - not needing a write-back, guaranteed the only copy in any cache and **DIRTY** - written more than once and the only copy in any cache.

On a read miss the block comes from another cache if it is in state **DIRTY** or from the main memory. On a write-hit a **DIRTY** block returns immediately, a **RESERVED** block changes into **DIRTY**, a **VALID** block is changed into **RESERVED** while at the same time writing through the word to main memory and at the same time other caches with the block change the state to **INVALID**. A write-miss behaves like a read-miss.

**Synapse** is used in the Synapse N+1 fault-tolerant multiprocessor with a additional bus which results in added bandwidth. The states of the protocol are: **INVALID, VALID** and **DIRTY**. Any cache with a **DIRTY** copy is called the **owner** of the block. In case of no dirty copy, memory is the owner.

**Berkeley** is implemented in the SPUR RISC multiprocessor [53]. It is similar to the Synapse protocol except that it uses direct cache-to-cache transfers in case of shared blocks, and dirty blocks are not written back to memory when they become shared. The state used are: **INVALID, VALID, SHARED-DIRTY** and **DIRTY**.

**Illinois** [67] assumes that blocks always come from other caches if cached or from the main memory otherwise. It uses the following states: **INVALID, VALID-EXCLUSIVE, SHARED** and **DIRTY**.

The following schemes use write-update instead of write-invalidate allowing multiple writers and multiple readers.

**Firefly** is a multiprocessor workstation [92] developed by Digital Equipment Corporation. The states of its protocol are: **VALID-EXCLUSIVE, SHARED** and **DIRTY**. Only dirty blocks are written back to memory at replacement. Because of update it never needs an invalid state. Caches use a special bus-line called the SharedLine to indicate sharing.

**Dragon** is another multiprocessor workstation being developed by Xerox PARC. The protocol is similar to that of Firefly. Possible states are: **VALID-EXCLUSIVE, SHARED-DIRTY, SHARED-CLEAN** and **DIRTY**. Unlike Firefly, writes to shared blocks are not sent to the main memory immediately but only to the caches.

**Directory-based schemes**

The main criticism of snoopy schemes is that because of the broadcast nature (i.e. the bus) of the interconnect, they are not scalable - in the same sense that bus-based multiprocessors are not scalable beyond tens of processors. This has led to the revival of interest in directory-based schemes similar to the first scheme proposed by Censis and Feautrier [21] more than a decade ago. The main drawback of these schemes however is enormous space requirements which (in simple schemes) grows linearly with the number of processors. Several approaches have been proposed to reduce the memory requirements of such directories including limited directories and chained (or linked-list) directories. Several people have surveyed existing directory schemes, especially Aggarwal in [4], [22] and Stenstrom in [84]. We will now discuss some of the directory schemes.

**Cache Directory duplication** was one of the first ever coherence schemes [90] and uses a central memory controller which duplicates each cache’s directory. This method has very poor performance due the centralised service bottleneck.

**The Full-map scheme** was first proposed by Censis and Feautrier [21] and has been the basis of many subsequent designs. It works by appending to each memory block a vector of bits with one bit per cache. The vector is generally refered to a **presence flag vector**. Each bit indicates the
Figure 2: Different types of cache-coherence protocols
presence or absence of the block in the corresponding cache. This method, though not very space efficient is very time efficient.

The Two-bit scheme by Archibald and Baer [10] uses two-bits as the state of each cache line. A cache line can be in one of the four global states:

- not present in any cache.
- present in one cache in read only mode.
- present in 0 or more caches in read only mode.
- present in one cache and modified.

The scheme resorts to broadcast if a block is present in more than one cache and as a result is inefficient in general networks if the degree of sharing is high.

Stenstrom’s MIN protocol [83] is similar to the Censor and Feautrier scheme with the difference that instead of associating the state information and the presence flag vector with the memory copy, this information is associated with the cached copy.

The Chained-directory scheme links caches with copies together in a linked list. Although such schemes are space efficient, they are not time efficient. The SCI scheme is based on chained-directories. The Stanford Distributed Directory (SDD) protocol [93] is similar to the SCI protocol but uses a singly-linked list to chain caches together. The main memory contains a head pointer to the last cache to access a particular line. This cache contains a pointer to the previous cache to access this line and so on. New caches are inserted at the head of the list and deletion results in invalidating all caches from the current cache to the tail of the list.

The Scalable Tree Protocol (STP) has been proposed recently by Stenstrom [85] and is similar to the linked list protocol described above but overcomes its limitation of long write latencies of $O(n)$. Instead of linking the caches as a chained lists, the protocol attempts to link all the caches in the list as an optimal tree. The result is a write latency of $O(\log n)$. The price to be paid for this is a slight increase in memory overhead of $O(k\log n)$ which the authors do not consider to be significant.

In this protocol each memory block has a pointer to the root of the tree, a pointer to the cache that most recently fetched the block and a pointer to the cache with a pending write request. Also, each cache line has a pointer to the parent of the cache in the tree, pointers to the children in the tree, a pointer to the cache that fetched the block after this cache and a pointer to the cache that fetched the block before this cache.

The Fuzzy-directory scheme [19] is another limited directory approach where the caches in the system are divided into a number of subgroups with a state bit identifying the occupancy of each sub-group. The scheme can track a small number of outstanding readable copies exactly depending upon the number of directory state bits associated with a cache line. On an overflow the protocol switches to a fuzzy state which no longer guarantees exact existence of a copy. A bit set in the presence vector only is a guess to the existence of the line. In order to invalidate the unknown number of copies in a group a multicast is required.

Tree-directory and hierarchical full-map These two schemes appear in [60] and improve on the storage overhead problem of the full map schemes. The first (tree) scheme overcomes the overflow problem of the limited directories scheme without resorting to broadcast or sacrificing existing copies. Whenever an overflow occurs, an extra bit associated with each pointer is set 1 to indicate that the entry points to a sub-tree. In this mode the first entry holds the pointer to the
parent node and the remaining entries pointers to the children. The overhead of this scheme is 25 % that of the limited directory scheme and 0.2 % more than that of the chained scheme.

The other scheme called the hierarchical full-map scheme is quite similar to the DDM protocol and organises the intermediate directories as set-associative. This satisfies (a modified form of the multi-level) inclusion property i.e. that all valid entries in the directory refer to data blocks in its descendent caches, however blocks not allocated in its descendent caches cannot have a valid entry in the directory. The authors suggest three configurations in which the hierarchical scheme can be incorporated including a multistage (omega) network with the directories in the switches and a hierarchy of rings and the third topology is embedded into a mesh.

4 Synchronisation mechanisms

In order to run shared memory programs, it is imperative to provide some form of synchronisation primitives in addition to the basic read-write protocol. A large number of shared memory multiprocessors provide spin-locks as a basic synchronisation mechanism to guarantee mutually exclusive access to critical sections. Typically an atomic or indivisible instruction reads and updates a memory location (i.e. a lock), inhibiting simultaneous attempts by other processors to acquire the lock. While a lock is held by a processor, other processors simply busy-wait until the lock is relinquished.

The rationale behind spinning is that most current parallel machines are used to run applications in essentially rigid environments with reasonably small critical sections [101]. In such situations spinning is favoured because the waiting processors usually have no other work to do and the cost of switching tasks is far greater than the gain in utilising the small duration of the critical sections. However, in a general purpose computing environment, spinning is clearly not desirable, as many different tasks may be active on a single node at a given time. Moreover, increasingly architectures tend to support lightweight threads with rapid context switching [8, 61, 76], resulting in the need for non-spinning mechanisms. Blocking locks are therefore considered more general.

We will first describe primitive operations (like test-and-set) that are needed to implement higher level synchronisation structures. These primitives can be found in most modern processors.

4.1 Primitives

Synchronisation primitives are required to perform some form of un-interruptable read-modify-write. Such operations can be regarded as atomic and many modern processors provide a primitive atomic operation to implement synchronisation. The use of such an atomic operation is to provide mutually exclusive access to a data structure. In [46] Herlihy describes the usefulness of various primitives. Concurrent data structures are normally implemented via critical sections, however, Herlihy concludes that of all the primitives only compare-and-swap has both non-blocking and wait-free properties. We shall focus more on synchronisation via critical sections.

Test-and-set

A test-and-set operation is an example of an atomic instruction used to implement spin-locks [38] and is supported by many processors in hardware. The primitive can be used very easily to implement locks. Test-and-set operates as follows:

test-and-set(address)
temp := [address];
[address] := 1;
return(temp)
}

The operation is un-interruptable and always returns (the previous value). Spinning on pure test-and-set generates excessive bus traffic because the lock value is modified irrespective of the availability of the lock. An optimisation of spinning on a read before attempting to test-and-set (known as test-and-test-and-set) [75] has found its way into most commercial multiprocessors in existence today. It, however, cannot prevent the avalanche of transactions that results whenever a lock is released and several processors attempt to do a test-and-set simultaneously. The excess traffic generated is due to the processors trying to acquire the lock with all but one competing to update the lock unsuccessfully on each lock release. The traffic generated due to such write-competition can be \(O(n^2)\), for an \(n\) processor system.

Atomic-increment/decrement

These instructions atomically increment and decrement a memory location and are generally more powerful than test-and-set and can be used to implement operations with fewer instructions. This primitive can be used to implement synchronisation points where more than one (upto \(M\)) processes can pass a fence while others wait. When the synchronisation is released, another waiting process can be allowed to pass. Test-and-set in contrast allows only one process to pass at any time.

Compare-and-swap

This primitive is considered to be universal by Herlihy and reduces locked regions to single instruction. Rather than constructing a critical region to update a shared variable, a single instruction per shared variable can do the job. Compare-and-swap takes the following form:

```c
boolean compare-and-swap(address, old, new)
{
  if w == old then
    w := new;
    return TRUE;
  else
    return FALSE;
}
```

This primitive is useful typically in queueing and dequeueing of tasks without locking.

Fetch-and-add

This primitive provides for adding an increment to a shared sum in parallel. A number of processors can simultaneously succeed in this instruction and the shared variable eventually gets updated by combining all the requests without involving any locking or unlocking. This primitive is found on the Ultracomputer [43] and the IBM RP3 [68].
Load-linked/Store-conditional

The load-linked and store-conditional primitive pair [47] and can been efficiently implemented in cache-coherent architectures [7, 52]. The load-linked operation copies the contents of a shared variable to a local variable and the store-conditional operation to the shared variable changes its value only if no other process has updated it in the mean time. Store conditional always returns success or failure.

4.2 Higher-level operations

These are high level abstractions of synchronisation primitives discussed earlier that can be used as system calls by the programmer, and include: locks, barriers and self-scheduling loops.

Locks

Lock and unlock operations are typically used to guard critical sections. The simplest locking operation has been implemented with the simple test and set primitive. However this lock generates excessive traffic because of the contention for the shared flag. This contention can be reduced by the test-and-test-and-set lock [75] which ensures that competing processors spin locally on read rather than on a shared flag. The latter type still results in a traffic avalanche when a lock is released. Nevertheless commercial multiprocessors like the Sequent Symmetry and the Encore Multimax support this type of lock.

Many researchers have proposed alternatives to test-and set locks. Anderson [9] proposed a test-and-set lock with exponential backoff. In this case test-and-test-and-set is not necessary. A ticket lock [62] avoids the additional traffic of the test-and set lock and also avoids the possibility of starvation. It consists of a counter to count the number of requests to acquire the lock and another counter for the number of times the lock has been released. A processor performs a fetch and increment on the request counter and then waits until the result (ticket) is equal to the release counter. This lock still causes contention due to polling on the common location. The performance of this lock can be improved like the test and set lock by adding suitable backoff.

Array based locks have been proposed by both Anderson [9] and Graunke and Thakkar [44]. Each processor uses the atomic operation to obtain the address of the location on which to spin. Each processor spins on a different location in a different cache line. Other list based locks have also been proposed in literature. Goodman has proposed the Queue on Lock Bit (QOLB) scheme [41] for the Wisconsin Multicube which requires additional hardware support (bit). Mellor-Crummey and Scott have proposed another scheme called the MCS lock [63] and is rather similar to the QOLB but implemented entirely in software. All processors holding or waiting for the lock are linked together in a list and each processor spins locally.

Barriers

Barriers are used to co-ordinate different phases of processors during execution. The simplest of all are the centralised barrier implementations where each processor updates a common counter to indicate its arrival and then polls the state until the counter reaches a certain value. These have rather similar drawbacks as the simple test-and-set lock in that they generate excessive traffic due to contention. Agarwal and Cherian have proposed backoff schemes [2] for the simple barrier implementations. Yew, Tzeng and Lawrie [100] have proposed a software combining tree barrier. This has the same effect as hardware combining. A shared variable is represented as a tree of variables with each node assigned to a different memory module. Processors are divided into groups
with each group assigned to a separate leaf of the tree. Processors update the state, climbing up the tree and eventually propagate the update to the root of the tree.

In the butterfly barrier [18] each processor participates in a sequence of \( \approx \log_2 P \) pairwise synchronisation steps. Mellor-Crummey and Scott present a tree-based barrier in [62] which spins locally, requires only \( O(P) \) space for \( P \) processors and performs \( O(\log P) \) transactions on its critical path.

**Self-scheduling loops**

Self scheduling loops are particularly useful in numerical programs in the parallel execution of independent loop iterations. The contentious shared-data item in this case is the next available subscript of the loop. Each of the processes executes the body of an iteration of the loop for a unique subscript. at the end of the loop all processes wait until all have reached the end just like in the barrier.

5 Other design issues

There are a number of other issues that need to be addressed when designing a VSM system. A brief outline of some of these follows.

- Most schemes assume the address space to be a homogeneous linear array of data words. Some however, structure the shared data into objects or language types. The potential advantage of structuring is that by introducing some semantic content in the layout of data in memory, false sharing will be reduced.

- The other issue is that of granularity of data. depending on the implementation, the grain size of coherence or sharing is either a page or a cache line. As mentioned before, larger grain size results in increased locality but greater risk of false sharing and contention.

- Related to the above is the mechanism of integrating virtual memory (in the conventional sense) with the virtual address space. OS based VSM systems provide VSM as an extension of the basic kernel activities, whereas hardware implementations of VSM can have a kernel implement VSM on top of the flat-address space.

- Finally, heterogeneity is another issue. Most VSM systems use homogeneous architectures but some have been built on heterogeneous architectures. Heterogenous architectures introduce another dimension of complexity due to the disparity of the node architectures i.e. different page sizes, byte-ordering and so on.

6 A survey of VSM architectures

A large number of virtual shared memory systems have been built to date. In general, VSM is either offered at the OS level or via specialised hardware to maintain a coherent shared address space. Certain systems also use a hybrid of software and hardware to support VSM. We will now discuss a number of VSM systems by categorising them into software and hardware-based approaches. We will highlight the main features of each system like the coherence protocol, synchronisation mechanisms etc.
6.1 Software-based VSM

There are two sub-categories in this, those that provide shared-memory via compiler/program-analysis or via structuring of the address space and those that provide VSM at the OS kernel level via page-sharing. This approach is less implicit than true shared memory since it requires (typically) some annotations from the programmer.

Munin

Munin [20] is a distributed shared memory system whose main feature is that unlike other related systems it provides a suite of different coherence mechanisms. It uses program annotations to match a particular consistency protocol to a particular data-object. Munin classifies data objects according to their access pattern and uses a type-specific coherence scheme [14] depending on the type of the shared data object. The objects supported are: read-only, migratory, producer-consumer, reduction, concurrent-write-shared and result objects. There are provisions to incorporate additional objects. The authors observe that there are very few general read-write objects and parallel programs behave differently during different phases of execution. Munin uses the release consistency model of shared memory.

A sixteen-processor prototype has been built on a Sun network with each node running a Munin server. The server executes a distributed directory-based coherence protocol in software, where each directory entry corresponds to a single object. The server also implements distributed locks and barriers via a distributed queue-based synchronisation protocol. The state of the global shared memory is contained in the data object directory which has an entry for every shared data object. A separate directory called the synchronisation object directory is used for locks and barriers. Consistency protocols can be switched with the help of protocol control bits: invalidate_or_update, delay_updates, flush_all, flush_home, locked_down and stable, specifying how to provide consistency for a particular object.

Release consistency is implemented with the help of the delayed-update-queue by buffering pending outgoing write operations. This queue has to be flushed whenever the local thread releases a synchronisation point.

Kali

Kali [54] is a software layer supporting a global name space on distributed memory architectures. The computation paradigm consists of a set of parallel loops. Access to local/non-local data is detected via the compiler analysis of subscripts of array references. The program requirements are (similar to the SPMD paradigm) that the processor topology, the distribution of data structures and that of parallel loops be specified and these correspond to three distinct primitives in the Kali language. Hence, both data-distribution and load-balancing are under user control. Computation in this paradigm proceeds as follows. An inspector loop is performed before the main computation to record any non-local array references. This is then followed by the executor loop which uses this information to exchange data efficiently.

IVY

Li [58] extends the concepts of traditional paging to distributed systems where pages of data can migrate from the local memory of one processor to that of another. The author also discusses several algorithms to solve the memory coherence problem, raging from centralised to distributed
algorithms. Each processor maintains a "copy set" for each page which contains a list of all processors that contain the page. Each page also has an owner to aid memory coherence.

Each process' address space is partitioned into private and shared portions. The shared portions collectively form a flat paged shared virtual address space. The granularity of synchronisation is a page. A memory manager on each node implements the cache-coherence protocol satisfying local and remote requests. It implements page faults in the conventional virtual memory sense. Each node maintains a page-table which records access rights to the page, the physical location and a copy-set which contains information on the nodes that have a copy of the page. The copy-set avoids the need for broadcast.

Li has implemented a prototype system based on the above ideas called IVY on a network of Apollo workstations. Three different schemes for directory implementation have been tried in IVY. The first, centralised scheme, consists of a central manager keeping track of all the pages. A request is always sent to the central manager who forwards it to the owner of the page and at the same time adds the faulting host to the copy-set of the faulting node. As is obvious the central manager can be a potential bottleneck in the system and hence this scheme is not scalable. In the second, distributed manager scheme, the bottleneck is avoided by distributing the manager across different nodes with each node managing a predefined subset of pages. The third scheme called the dynamic distributed scheme eliminates the managers and extends the page-table entry of each node with another attribute called the probable owner. This helps the host in locating a page. IVY also implements replacement of pages to the secondary storage and to the memories of other nodes. The algorithm for replacement is of the LRU type which also prioritises the pages according to their state.

Synchronisation in IVY is based on the event-count primitive. The event-count is implemented independent of the shared memory using RPC.

Li has also implemented his system on a hypercube called Shiva [58].

**DVSM on TOPSY**

TOPSY [98] is a distributed memory multiprocessor developed at the City University which can support up to 256 processors. Each processing node in TOPSY is based on an Motorola 68030 processor and the network is a toroidal circuit switched network with a bandwidth of 12 Mbyte/sec/channel. A Unix like operating system called Meshix also runs on TOPSY. Recently VSM has been implemented on TOPSY [86] using an external-pager approach similar to Mach. This scheme allows distributed Unix processes to share paged-virtual memory regions. Coherence is enforced via user-level server processes and the mechanism is similar to Li's dynamic distributed coherence algorithm. The coherence model is strict coherence and features page-invalidation. The main synchronisation mechanism provided is the spin-lock.

**KOAN**

KOAN [55] is similar to IVY and Shiva and has been implemented on the IPSC/2. The VSM is implemented with the help of a set of handlers and servers. The handlers include a read-fault handler and a write-fault handler. A library of system calls provides an interface to manage the shared virtual memory.
PLUS

PLUS [16] is a low-hardware-cost virtual shared memory system which provides processor consistency with the help of a write-update protocol. The unit of replication is a page whereas the unit of coherence and memory access is a word. A PLUS node consists of a processor, cache, local memory and a memory-coherence manager. Nodes are interconnected by a fast interconnection network. A page in the virtual address space corresponds to a number of replicated pages on several nodes. The first of these pages is called the master copy. Each virtual page is mapped by a node to its closest copy. The copy-list is ordered by the kernel to minimise the network path length in the list. A write does not block a processor until it receives a subsequent read to the same address. Thus multiple writes can be in progress at the same time. This ensures strong ordering within a processors but does not guarantee the inter-processor ordering - i.e. processor consistency. To force strong ordering amongst processors, the programmer can use explicit fence operations. PLUS provides a set of delayed read-modify-write operations for synchronisation. The delayed synchronisation allows software pipelining and fast context switching.

VMP and Paradigm

The VMP [26] is the precursor to the Paradigm multiprocessor project [28]. The main feature is the software-based cache management. Each VMP processor board contains a virtually addressed cache with a 128-byte block size and a local memory that stores cache-management code and data structures. VMP provides several refinements to the original design including locking support in hardware, memory-based consistency support and hardware handling of simple cache misses.

The VMP coherence scheme [27] is a dynamic scheme implemented as part of the kernel of the VMP multiprocessor. Coherence is maintained on a page basis. The main memory is organised as a sequence of cache page frames and a cache page can be in one of the two states: shared or private. The protocol is ownership based and the private copy of a cache page is said to be owned by the cache that holds it. The processors use read and write bus transactions to request or release ownership. A bus monitor on each processor performs snoop-like function by servicing transactions. Each processor also has an action table to record consistency information.

Paradigm is a subsequent development of the VMP architecture. The hardware architecture consists of processing configurations connected via a switching network. The configurations consist of clusters of processors linked hierarchically to memory modules. Each cluster, called a multiprocessor module (MPM), contains an interface that links it to a high-speed network. A number of MPMS are connected by a higher-level bus to an inter-bus cache module called an MPM group. A number of MPM groups linked by a memory bus form a node. Each MPM is connected to all other MPMS via the switching network.

The operating system provides a model with multiple shared virtual address spaces for separate applications and multiple processes per address space for parallel applications. Based on the V distributed operating system, Paradigm also supports multicast communication and process groups. Unlike other virtual shared memory systems, Paradigm implements virtual memory on top of files which are built on top of a communication facility (RPC). The authors believe that this avoids building a separate file caching and consistency mechanism and avoids building files on top of the virtual memory system, arguing that files are a more general facility.
Spectre

Spectre [72] is a VSM system proposed for a transputer-based general purpose distributed memory multiprocessor. It takes advantage of the fast context switch on the transputer to support multiple application contexts in order to hide long latencies. The local memory on each node is partitioned into private memory and non-local cache. It supports a snoopy coherence protocol on a ring interconnect and a limited pointer directory-based (NEWS) protocol for a mesh interconnect. Rather than storing exact pointer the copy-set only tells the probable existence of the copies of a data-page in one of the four directions at a mesh point (similar to Li’s probable-owner concept).

Clouds

Clouds [29, 30] is an object-based distributed operating system. In [73], Ramachandran et. al. have proposed implementation of distributed shared memory in the Clouds kernel. This work is based on Li’s work but provides additional efficient support for synchronisation by integrating synchronisation with cache-coherence. In Clouds, virtual shared memory is used for object invocation rather than parallel processing.

Mermaid

Mermaid [102] is a heterogeneous distributed shared memory system which has been implemented on networks of SunOS workstations and DEC Firefly multiprocessors. The VSM is page-based similar to Li’s IVY. It consists of three modules: a thread-management module responsible for thread creation, termination, termination and scheduling. A shared-memory module manages memory and handles page-faults. The third module called the remote-operations module implements the inter-host communication protocol. It uses Li’s fixed-distributed algorithm. The VSM is implemented as a library package at the user level and linked into the Mermaid applications program. To support heterogeneity, Mermaid supports automatic data conversion routines.

In addition to above, most modern (distributed) operating systems have begun to support virtual shared memory in some form. Examples include Mach (the external pager mechanism) [36], Amoeba [13], Agora [15], Amber [24] CapNet, Choices[77], Emerald, Mether [64] and Mirage [34].

6.2 Hardware-based VSM

The other class of VSM systems involve some kind of hardware support to provide a coherent shared virtual memory. The hardware support mostly consists of efficient cache-coherence mechanisms. Again two sub-categories are common. Those that are based on snoopy coherence schemes and those that support directory-based coherence. Snoopy cache-coherent architectures are less common due to their poor scalability. Most schemes therefore use either directory-based protocols or some-kind of multi-level snooping.

MIT Alewife

Alewife [3] is a distributed shared memory machine with coherent caches. A directory-based protocol ensures cache-consistency. An Alewife node consists of a processing element, a floating point unit, a cache, main memory, a cache/directory controller and a router. The protocol enforces strong coherence and multithreading is used to overcome the long communication latencies. The processing element is a modified version of SPARC. Alewife provides the full/empty bit approach for fine-grain synchronisation as in the HEP multiprocessor [82]. A bit is associated with each memory
word to indicate its status. Loading an empty word or storing a full word cause the processor to
trap and switch context, thereby hiding the synchronisation latency. Coarse-grain multithreading
is achieved by the SPARC register windows as task frames for multiple threads.

The MIT Alewife machine incorporates a protocol called the LimitLESS scheme [23]. The
LimitLESS (Limited directory Locally Extended through Software Support) scheme implements a
limited number of pointers in hardware. If an overflow occurs, the memory module interrupts the
local processor. The processor then emulates a full map directory for the memory block that caused
the interrupt. The Alewife processor has a fast trap mechanism which guarantees the efficiency of
this operation.

**Wisconsin Multicube**

The Multicube [42] consists of a symmetric grid of buses having a processing element at each bus
intersection. Also, each column bus is connected to a memory module. Each processing node
consists of a processor and a snoopy cache which is connected to both the row and the column bus.
The snoopy protocol requires that each memory block is associated with a home column or a home
bus. The architecture is based on a family of more general interconnection topology called the
multicube. It consists of \(n^k\) processors with each bus connected to \(n\) processors and each processor
connected to \(k\) buses, where \(k\) is the dimension of the multicube. The Wisconsin multicube is a 2
dimensional network. A conventional bus-based multiprocessor can be considered as a multicube of
1 dimension. The origins of this network lie in the hypercube topology. Multicube does not inherit
the long message latencies of the hypercube, instead a cache miss requires only twice the number
of bus transactions as a single bus multiprocessor.

The multicube consists of two-level caching - the processor cache and the snooping cache.
Consistency between the two levels is maintained with a write-through strategy. A line is always in
one of the two global states - unmodified or modified. Besides this there are also local modes: local,
modified and invalid. The protocol consists of four types of transactions: read, read-mod, allocate
and write-back.

**Stanford DASH**

DASH (Directory Architecture for SHared memory) [57] is a scalable shared memory architecture
being built at Stanford University. The architecture consists of a number of processing clusters
connected via a two-dimensional grid network. Each cluster has a directory memory which corre-
ponds to a portion of the shared physical memory. The network is actually a "pair" of meshes
- one mesh for requests and the other for replies. Wormhole routing is used to minimise latency
which is one of the main goals of the project.

Currently, a prototype is being built with 16 clusters, where each cluster is a Silicon Graphics
power station 40/240. A cluster consists of a number of processing elements and caches connected
via a bus. Consistency within a cluster is maintained by a snoopy Illinois like protocol. Each
cluster is connected to the network via a directory board. The directory board consists of a
directory controller (DC) a pseudo-cpu (PCPU), a reply controller (RC), the network interface and
hardware monitoring logic. The DC contains the directory memory and issues network requests
and replies. The PCPU is responsible for buffering/issuing incoming requests and the RC is used
to track outstanding requests made by local processors and to buffer replies from remote clusters.
The RC has access to a remote access cache (RAC) which maintains the state of the currently
outstanding requests and buffers reply from the network. The RAC is organised like a snoopy
cache.
The directory controller maintains consistency amongst the directory memories and DASH supports the release consistency model. Release consistency allows write operations to be pipelined thereby hiding the write latency.

DASH implements the release consistency model in its invalidation based ownership protocol. Release consistency as explained in section order with respect to other processors and order is preserved only before releasing synchronisation. This is implemented with the full-fence and write-fence operations.

The protocol consists of the following states: uncached-remote, shared-remote and dirty. For each block of memory there is always the notion of a local-cluster, home-cluster, and the owning cluster. Normally the home-cluster is the owning cluster but a remote-cluster containing the block as dirty may instead be the owner. Only the owning-cluster can update the directory state. Each entry in the directory of the prototype is 17 bits wide where one bit is reserved for the state (i.e. read(shared) or read/write(dirty)). The other 16 bits form a bit-vector of pointers to the clusters.

The Data Diffusion Machine

The Data Diffusion Machine (DDM) [95] is a scalable multiprocessor architecture that offers a shared virtual address space. In the DDM, a datum has no fixed home location and the virtual address is completely decoupled from its physical location. Where data resides at any given time is entirely dependent upon where it is most in demand. Copies of data with the same virtual address can co-exist in different parts of the machine and a request to a data item will return its nearest copy. Communication due to data access is thus localised only in the parts of the machine that are actively using the data.

The hardware organisation of the DDM is hierarchical. At the tips are the worker processors, each with a large, local set-associative memory, the sole form of main memory. The memory is connected to the local bus via a memory controller and the local bus is connected to a higher bus via a directory controller. The directory controller also has access to a set-associative directory. The root-directory has a disk attached to it which holds overflow data (not shown in the figure). The machine can be arbitrarily scaled by increasing the number of levels in the hierarchy. The unit of data storage is an item, the size of which is fairly small (of the order of a few words) to reduce false sharing. A directory stores state information for the items in the subsystems below it. The existence of multiple copies of data requires the consistency of data to be maintained.

The cache-coherence protocol of the DDM is a hierarchical snooping protocol based on write invalidation [45]. A directory-based version of the protocol has also been implemented to support an implementation of the DDM using point-to-point interconnects [71]. One of the features of the protocol is the combining of multiple reads to the same item. Read requests are combined on their way up, and read responses on their way down. The protocol also handles replacement in memory and directory. If a set that is to accommodate the new data item is full, one of the items is chosen to be replaced with the new item and a replacement transaction is sent above which carries on moving upwards until it either finds another copy, in which case the item is simply thrown away, or until a new site for the replaced item can be found. If the item cannot be accommodated in any subsystem (i.e. all sets are full), the item is moved out of the machine onto a secondary storage.

Kendal Square Research KSR1

The KSR1 from Kendal Square Research is a commercially available multiprocessor which has been installed at a number of sites recently. The architecture of the KSR1 [74, 7] is very similar to the DDM. In particular, like DDM KSR1 does not have a notion of home location and there

20
is no common memory. Also like the DDM, the organisation of the KSR1 is hierarchical, albeit a
hierarchy of rings rather than links or buses. A processor in the KSR1 is a custom superscalar 64-
bit device rated at a peak 40 MFLOPS. Besides the processor, each cell consists of a floating point
co-processor and an I/O co-processor. the cell board also contains a pair of data and instruction
 caches, 256 KByte each. The local memory on each node is 32 MBytes. A system can contain
upto 1088 such processors, resulting in a peak performance of 43 GFLOPS. The processors are
interconnected via rings with 32 processors per ring. Two levels of rings are used, the lowest
level rings (ring:0) are connected via a second level ring (ring:1). The bandwidth of ring:0 is 1
GByte/sec. The second level ring can be configured to 1, 2 or 4 GBytes/sec.

The memory model provided by the KSR1 is sequentially consistent. Consistency information
is stored in directory (one per ring:0) units called ALLCACHE Routing Directory cell of ARD
forming the Search Engine:0 (SE:0). The ARDs are connected together in a second level ring
called the SE:1. For synchronisation, the hardware assists in the get and release operation.
These primitives are used to implement the common synchronisation mechanisms. As an optimisation the
architecture provides the pre-fetch and the post-store asynchronous operations that help is reducing
the effective latency of data access.

The KSR1 runs an extension of the OSF/1 operating system and a number of programming
languages and parallel programming tools are also provided. Initial experiences with the KSR1 at
Oak Ridge National Laboratory [33] and Manchester University [6] have been promising.

NYU Ultracomputer

The Ultracomputer [43] is a shared memory multiprocessor whose distinctive feature is the fetch-
and-add operation used for synchronisation. The processing elements are connected to the memory
modules via an Omega network. The fetch-and-add operation is implemented in the network.
Another interesting feature of the network is that it provides combining in hardware.

The J-machine

The J-machine is basically a distributed memory message passing multiprocessor which can support
several programming paradigms, including shared-memory, efficiently. The J-machine uses unique
components and regards message passing and message management as critical and fundamental
operations. Instead of processors the machine consists of message driven processors (MDPs) which
execute code in response to a message arrival. Each MDP consists of hardware mechanisms to
dispatch (and receive) messages with minimum delay. The network is a three-dimensional mesh.
Optimised node-to-node communication takes 2 \( \mu \)sec. Synchronisation support is provided in
the form of futures [12]. To facilitate the support of shared-memory efficiently, optimal address
translation mechanisms exist which are visible to the programmer (in the form if the XLATE
instruction).

Because of its low overhead the J-machine is able to exploit much finer grain parallelism. Deliv-
ering a message and dispatching a task takes \( \approx 3 \mu \)sec as compared to 3 ms on an iPSC1 hypercube.
Among the models of parallel computation that are to be supported on the J-machine are actors
[5], shared-memory, dataflow and communicating sequential processes.

A directory-based hierarchical protocol [9,4] has been designed to implement global shared mem-
ory on the J-machine. The protocol is an ownership based write invalidate protocol. The directory
is organised hierarchically which is similar to a butterfly network. A child node can have more
than one parent in the butterfly network. A complete tree is formed with the collection of nodes
on which a particular address can be stored. The result is that it avoids the bottleneck of one top node because there are different trees for different addresses.

A mapping function is used to find the parent/child node of a node, given an address. The address consists of two parts - the map part and the key part. The function tries to keep messages confined to physically small areas wherever possible.

Directory entries are of two types - parent or leaf entry. The leaf entry consists of the actual data whereas the parent entry contains state and route-information for the memory block. A read miss is split into two phases. The first to locate the block and the second to send the block to the requester. Locating a block is done by sending a findLcowestCommonForRead message to the parent node, which is propagated upwards until the block is located. The second phases starts by the remote node sending data to the requester directly.

A write is performed by locating all copies, locking and then deleting them followed by transferring the ownership and the last data copy to the new owner. A synchronisation primitive is built on top of the read and write mechanisms.

MemNet

MemNet [31] is being developed at University of Delaware. The prototype consists of nodes connected through a 200Mbps insertion modification token ring. It uses the same coherence semantics as IVY i.e. a write-invalidate protocol. the ring interconnect serves as a logical broadcast medium. The coherence grain size is 32 bytes rather than a page. Each node has a Memnet device (hardware) responsible for servicing remote accesses and maintaining coherence. The address space is divided into a private part and a shared part, while the physical memory on each node is divided into a reserved part and a cache part. The reserved portion serves as a permanent home for a subset of the address space whereas the cache part temporarily stores blocks from other nodes. There is no hardware support for synchronisation.

Aquarius

The Aquarius multiprocessor project is similar to the Wisconsin multicube and consists of processor-memory nodes interconnected via multi-dimensional busses. Unlike the multicube the shared memory is distributed amongst the nodes. The amount of bandwidth increases with the dimensionality of the network. Each node consists of a processor, memory, cache and a directory and controller for coherence. Sharing is most efficient amongst processors on a single bus but access to processors/memories on other buses is possible. Aquarius provides a coherence protocol that maintains consistency on a single bus with a snoopy protocol and amongst buses with a directory protocol.

Memory is shared using the high bits of the memory address. This serves as a natural division of directory information and is efficient for private data.

The Aquarius protocol is a hybrid directory and snoopy protocol. A snoopy protocol maintains coherence on a single bus and a directory protocol takes care of multiple buses. The protocol utilises split transactions so that a cache is able to relay a transaction to another cache on behalf of the requester. The unit of coherence is a cache block. Each block is logically associated with the root node of that block (i.e. the home node). The root node contains directory information for the block. Each cache block has an associated state and a directory state.

The protocol allows the notion of local sharing by keeping communication local to a bus. The protocol also maintains the address of the bus on which a block resides. This aids in directing coherence transactions.
Gigamax and Lynx/Galactica Net

Gigamax [97] is an extension of Encore’s Multimax system with a hierarchy of busses and caches
and is similar in some ways to Paradigm. It consists of several processing clusters connected by
fiber-optic links and a high speed bus which serves as a global switch. A cluster consists of a
Multimax and a second-level cache which are connected together by the Nanobus (so called because
of its 80 nanoseconds cycle time). The cluster cache stores the locations of most recently accessed
memory blocks, the interconnection interface which couple with the rest of the system maintains
coherence at the cluster level\(^1\).

The Lynx/Galactica Net [96] project is based on the earlier work by Wilson on Encore Gigamax
and consists of a network of Lynx multiprocessors. The Lynx is a shared-bus multiprocessor
consisting of four Motorola 88110 processors, 256K of secondary caches, 256M DRAM and a VME
connection operating at 40Mbytes/sec. The Galactica Net employs a combined hardware/software
strategy for coherence. Each node in the net consists of a Lynx multiprocessor, a Galactica Net
Interface Module and a Mesh Routing Chip. The Net maintains local memory on each node on a
page basis as a globally shared address space. The shared memory pages are kept coherent with
an update based protocol.

SCI

The Scalable Coherent Interface (SCI) [51] is an IEEE standards project (# P1596) which is
based on the earlier work of Futurebus standardisation [88]. SCI assumes a large point-to-point
interconnection network for the sole purpose of scalability. However it does not describe a particular
topology as it is only an interface. At the heart of the SCI project is the distributed directory
protocol described earlier.

The standard defines a logical and physical layer for upto 64K nodes. A node is notionally a
combination of one or more of the following: a processor (multiprocessor), memory module or an
I/O adapter. Physical addresses are 64 bits long with 48 bits of address and 16 bits for the node-id.
Request and response messages are called subactions. A subaction contains a command, data and
status fields. The coherence unit is 64 bytes wide.

The interface consists of an input and an output link operating at 1 Gbyte/sec in initial imple-
mantations.

The protocol proposed for the SCI in [51] and reviewed by Stenstrom [84, 85] and Chaiken
[22], maintains the sharing set in a linear-list associating two pointers with each line. One of these
pointers points to the successor and the other to the predecessor in the list.

The mechanics of the protocol is as follows. On a read the memory returns the requested block
if the list is empty otherwise it returns a pointer to the head of the list; the head-pointer in the
memory is now modified to point to the reading cache, i.e. the new head in the list. The reading
cache sets its pointer to point to the head cache in the list and send a request to the (old) head.
The old head returns the block and updates its predecessor to point to the new head.

In case of replacement, the cache that replaces a block removes itself from the list by sending
the predecessor id to the successor and the successor id to the predecessor. This is the advantage
of having a doubly linked list of caches.

When a write is attempted the writing cache is put at the head of the list and invalidations are
sent to all the caches in the list one at a time. The invalidated caches remove themselves from the
list as above and when the last cache removes itself the write is said to have completed.

\(^1\)Due to increased power of current processors, Encore does not consider GigaMax to be commercially viable
anymore.
The protocol allows certain optimisations like the optimised DMA without going through the sharing list. It also allows for combining of multiple requests and responses.

Merlin

Merlin [99] is a multiprocessor with a two-level network. A small network is used for control whereas a larger one is used for application traffic. The application computer hosts are linked by sets of fast optical fibre networks with low latency. There is no global virtual memory, instead, each global page is a spanning tree of buffers and routers. The main principle is that of mapped reflective memory or mirror memory. Each node makes a temporary copy of the address and value of each word written to the local memory. If it is a shared page a copy is passed onto all other memories that share it. Each node consists of three sets of 64K RAMS that act as address maps. The output map translates local physical addresses to global addresses. A steering map guides a transaction at an intermediate node, and an input map translates a global address to local physical address.

7 Conclusion

We have presented a comprehensive survey of Virtual Shared Memory architectures and the basic principles involved. The systems are classified according to the level at which they provide the abstraction of a shared address space. Software schemes can be implemented at a relatively low cost either by the assistance of the compiler/programmer or by the extension of the OS kernel to provide systemwide page coherence. The main problem with latter is that page sizes are usually large and therefore are potentially prone to thrashing. Hardware-based VSM architectures are more complex and costly to built but can provide an efficient implementation of VSM.

8 Acknowledgements

The work described in this report is supported by the ESPRIT project 2471 PEPMA and has benefited from collaboration with our partners in the project, especially Meiko Limited, Swedish Institute of Computer Science and University of Madrid.

References


