J. Webster (ed.), *Wiley Encyclopedia of Electrical and Electronics Engineering*
Copyright © 1999 John Wiley & Sons, Inc.

SHARED-MEMORY MULTIPROCESSORS

Multiprocessor systems that logically integrate individual processor memory modules to implement a single global address space are termed shared-memory multiprocessors. The shared-address-space model for parallel programming is simple to use and frees the programmer from the responsibility of orchestrating all communication and synchronization through explicit message passing to access remote data. As a result, this class of multiprocessor systems has received much commercial as well as research interest. Most existing multiprocessor systems today support shared-memory abstraction in some form (either in hardware or in software or in both).

The effectiveness of shared-memory systems in providing parallel and distributed systems as a cost-effective option for high-performance computation is quantified by four key properties: *simplicity, portability, efficiency, and scalability*.

Simplicity Shared-memory systems provide relative ease of use and a uniform model for accessing all shared data, whether local or remote. In addition, shared-memory systems should provide simple programming interfaces that allow them to be platform- and language-independent.

Portability Portability of the shared-memory programming environment across a wide range of platforms such as clusters of PCs and programming environments is important, as it obviates the labor of having to rewrite large, complex application codes. In addition to being able to be portable across "space," however, good shared-memory systems should also be portable across "time" (able to run on future systems), as that enables stability of the system.

Efficiency For shared-memory systems to achieve widespread acceptance, they should be capable of providing high efficiency over a wide range of applications, especially challenging applications with irregular and/or unpredictable communication patterns, without requiring much programming effort.

Scalability To provide a preferable option for high-performance computing, good shared-memory systems should be able to run efficiently on systems with hundreds (or potentially thousands) of processors. Shared-memory systems that scale well to large systems offer end users yet another form of stability—knowing that applications running on small- to medium-scale platforms could run unchanged and still deliver good performance on large-scale platforms.

Most existing shared-memory multiprocessor systems provide an efficiently balanced support to all of these properties to the maximum possible extent. In addition to this, all shared memory systems have to address two critical issues of *cache coherence* and *shared-memory consistency*. Figure 1 illustrates the spectrum of shared-memory multiprocessor systems. Based on the architectural approaches to support the shared memory, the current mainstream multiprocessors can be broadly grouped into two categories:

(1) Physically shared memory (*PSM*) multiprocessors
(2) Distributed shared-memory (*DSM*) multiprocessors

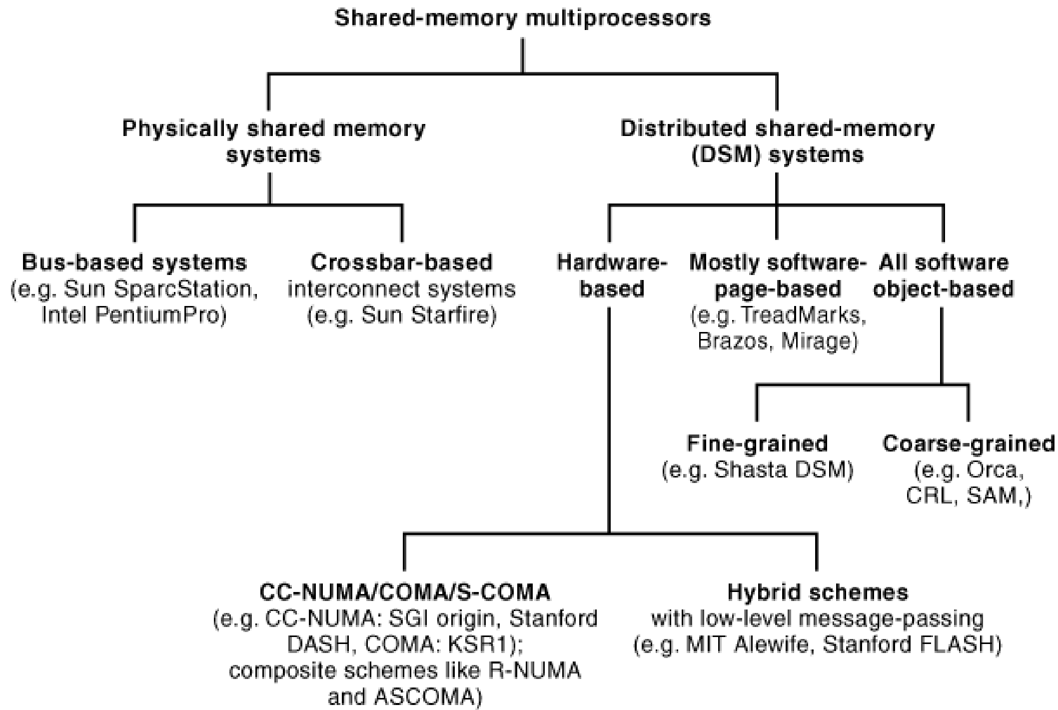

Fig. 1. Taxonomy of shared-memory multiprocessors.

Time	Processor P1	Processor P2
↓	$x = 0$	$y = 0$
	$x = a$	$y = b$
	⋮	⋮
	$y = c$	
		$x = d$

Fig. 2. Coherence problem when shared data are cached by multiple processors. Suppose initially $x = y = 0$ and both P1 and P2 have cached copies of x and y. If coherence is not maintained, P1 does not get the changed value of y and P2 does not get the changed value of x.

Cache Coherence

Although shared-memory systems facilitate global accesses to remote data in a straightforward manner, from the programmer's point of view the difference in access times (latencies) of local and remote memories in some of these architectures is significant (it can be a factor of 10 or higher). Uniprocessors hide these long main-memory access times by the use of local caches at each processor. Implementing (multiple) caches in a multiprocessor environment presents a challenging problem of maintaining cached data coherent with the main memory (possibly remote), that is, *cache coherence* (Fig. 2).

Snoopy Cache-Coherence Protocols. Small multiprocessor systems, using a shared bus, implement what are known as *snoopy protocols* for maintaining cache coherence. The basic idea is to make all caches snoop the shared snoopy bus. When a processor wants to write into a shared cache block, the write request

is transmitted on the bus. All caches read the address associated with every read/write request and check whether that address is currently cached. If so then the corresponding entry is either invalidated (in case of a write request), or used to satisfy the read request. For *write-through* caches, wherein data are written to main memory and the caches simultaneously, this is only an incremental addition to normal cache protocol, and the memory is always up to date. *Write-back* caches, which copy the modifications to data to the source (main memory) only when a cache block is replaced, require some extra work. In this type of caches, the most recently modified copy of the data may be cached. For read misses, the coherence mechanism has to snoop all the caches and retrieve data from the cache that has the most recent data.

Snoopy protocols are implemented by broadcasting both read and write requests on the bus, making all the caches read the address, and, if the address matches with one in the processor's cache, either invalidating the data or supplying data from there. Since the bus processes only one request at a time, concurrent writes to the same cache are automatically serialized. This serialization of requests by the bus imposes an ordering on all writes, which is critical to maintaining coherence.

For small multiprocessor systems (up to 64 processors), snoopy cache coherence protocols work well. The use of caches reduces the bandwidth requirements for bus and memory. Furthermore, as the caches are kept functionally transparent, the shared-memory programming model is preserved. For large and scalable systems, however, the bus becomes a communication bottleneck. Thus, the early shared-memory models did not allow caching of shared data, and the programmers had to deal with the long memory latencies. As a result, shared-memory systems were no easier to program than message-passing architectures. This has emphasized the need to develop a scalable cache coherence mechanism to extend to the scalable shared-memory architectures.

Directory-Based Cache Coherence. The directory-based cache coherence protocols use a directory to keep track of the caches that share the same cache line. The individual caches are inserted and deleted from the directory to reflect the use or rollout of shared cache lines. This directory is also used to purge (invalidate) a cache line when that is necessitated by a remote write to a shared cache line.

The directory can either be centralized, or distributed among the local nodes in a scalable shared-memory machine. Generally, a centralized directory is implemented as a bit map of the individual caches, where each bit set represents a shared copy of a particular cache line. The advantage of this type of implementation is that the entire sharing list can be found by simply examining the appropriate bit map. However, the centralization of the directory also forces each potential reader and writer to access the directory, which becomes an instant bottleneck. Additionally, the reliability of such a scheme is an issue, as a fault in the bit map will result in an incorrect sharing list. Fault tolerance is also a major concern in this centralized approach, as an incorrect sharing list becomes a single-point failure.

The bottleneck presented by the centralized structure is avoided by distributing the directory. This approach increases the reliability of the scheme. The distributed-directory scheme (also called the distributed-pointer protocol) implements the sharing list as a distributed linked list. In this implementation, each directory entry (being that of a cache line) points to the next member of the sharing list. The caches are inserted and deleted from the linked list as necessary. This avoids having an entry for every node in the directory.

Shared-Memory Consistency Models

In addition to the use of caches, the scalable shared-memory systems migrate or replicate data to local processors. Most scalable systems choose to replicate (rather than migrate) data, as this gives the best performance for a wide range of application parameters of interest. With replicated data, the provision of *memory consistency* becomes an important issue. The shared-memory scheme (in hardware or software) must control replication in a manner that preserves the abstraction of a single-address-space shared memory.

The shared-memory consistency model refers to how local updates to shared memory are communicated to the processors in the system. The most intuitive model of shared memory is that a read should always return the last value written. However, the idea of "the last value written" is not well defined, and its different interpretations have given rise to a variety of memory consistency models, namely sequential consistency (1),

processor consistency, release consistency (2), entry consistency (3), scope consistency (4), and some variations of these.

Sequential consistency implies that the shared memory appears to all processes as if they were executing on a single multiprogrammed processor. In a sequentially consistent system, one processor's update to a shared datum is reflected in every other processor's memory before the updating processor is able to issue another memory access. The simplicity of this model, however, exacts a high price, since sequentially consistent memory systems preclude many optimizations such as reordering, batching, or coalescing. These optimizations reduce the performance disadvantage of having distributed memories and have led to a class of weakly consistent models.

A weaker memory consistency model offers fewer guarantees about memory consistency, but it ensures that a suitably well-behaved program executes as though it were running on a sequentially consistent memory system. Again the definition of "well behaved" varies according to the model. For example, in *processor-consistent* systems, a load or store is globally performed when it is performed with respect to all processors. A load is performed with respect to a processor when no write by that processor can change the value returned by the load. A store is performed with respect to a processor when a load by that processor will return the value of the store. Thus, the programmer may not assume that all memory operations are performed in the same order at all processors.

Memory consistency requirements can be relaxed by exploiting the fact that most parallel programs impose their own high-level consistency requirements. In many programs this is done by means of explicit synchronization operations on synchronization objects such as lock acquisition and barrier entry. These operations impose an ordering on the access to data within the program. In the absence of such operations, a program is in effect relinquishing all control over the order and atomicity of memory operations to the underlying memory system. In the *release consistency model*, a processor issuing a *releasing synchronization operation* guarantees that its previous updates will be performed at other processors. Similarly, a processor issuing an *acquiring synchronization operation*, guarantees that other processors' updates have been performed locally. A releasing synchronization operation signals other processes that shared data are available, while an acquiring operation signals that shared data are needed. In *entry consistency model*, data are guarded to be consistent only after an acquiring synchronization operation and only the data known to be guarded by the acquired object are guaranteed to be consistent. Thus, a processor must not access a shared item until it has performed a synchronization operation on the items associated with the synchronization object.

Programs with good behavior do not assume a stronger consistency guarantee from the memory system than is actually provided. For each model, the definition of good behavior places demands on the programmer to ensure that a program's access to the shared data conforms to that model's consistency rules. These rules add an additional dimension of complexity to the already difficult task of writing new parallel programs and porting old ones. But the additional programming complexity provides greater control over communication and may result in higher performance. For example, with entry consistency, communication between processors occurs only when a processor acquires a synchronization object. A large variety of DSM system models have been proposed over the years with one or multiple consistency models, different granularities of shared data (e.g. object, virtual-memory page), and a variety of underlying hardware.

Physically Shared Memory Multiprocessors

The structure of a PSM multiprocessor system is illustrated in Fig. 3. A small number of microprocessors (typically less than 64) are integrated with a common memory using a shared bus or a crossbar interconnect. This allows all the processors to have roughly equal access time to the centralized main memory, that is, *uniform memory access* (*UMA*). PSM multiprocessors are also termed *symmetric multiprocessors* (*SMP*s) or

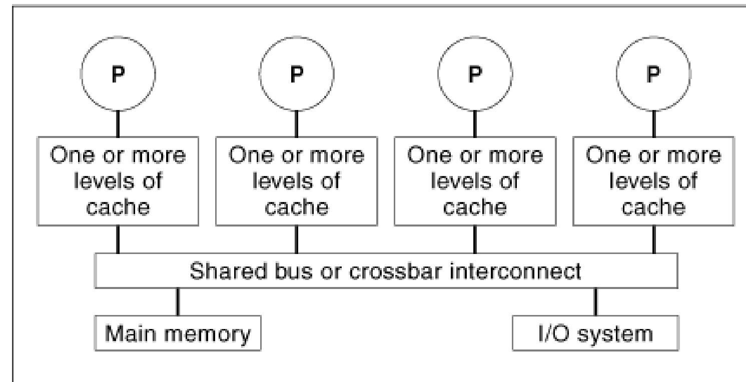

Fig. 3. Physically-shared-memory multiprocessors (P: processor; I/O: input and output).

centralized shared-memory processors. PSM multiprocessors using a shared bus interconnect are called *bus-based symmetric multiprocessors*.

The primary strengths of these systems include uniform memory access and a single address space, offering the ease of programmability. They do not need explicit data placement, as memory is equally accessible to all processors. The PSM design approach is used in commercially successful machines like the Compaq PentiumProm and the Sun UltraEnterprise.

Bus-Based Systems. The early PSM machines, which include many of the desktop PCs and workstations, used a shared serial bus with an address cycle for every transaction. This tied down the bus with each access for the needed data to arrive. A typical example is the Sun Microsystems Mbus (5), used in SparcStations. This shared bus, in addition to allowing access to the common memory, is used as a broadcast medium to implement the snoopy cache coherence protocol.

Subsequent PSM designs used the *split-transaction bus* (6), wherein separate buses exist for address and data. This allows an address cycle to overlap a data transfer cycle. Split-transaction buses were used in Sun Microsystems' original Gigaplane (6), in the Ultra-Enterprise 3000–6000. The split-transaction bus also allowed overlapping of snooping and data transfer activities, increasing the bandwidth. However, this needs handshaking during data transfer. The *pipelined bus*, used as the PentiumPro system bus, is a special case of the split-transaction bus wherein the address and data cycles are pipelined and devices can respond only in specific cycles, obviating the need for the data handshake. This scheme, however, requires the data cycle to correspond to the slowest device.

Crossbar-Based PSM Systems. In the crossbar-based systems, the data bus is replaced with a crossbar to provide a high-performance UMA. The address bus is also replicated by a factor of four. Point-to-point routers, and an active centerplane with four address routers, are the key components of today's largest UMA symmetric multiprocessors, such as the Sun UltraEnterprise series (5).

Although PSM multiprocessor architectures are used in most of the commercially successful machines, these systems have a high minimum memory access latency compared to high-performance uniprocessors. Furthermore, the inherent memory contention in these systems prevents scalability.

Sun Ultra-Port Architecture. Figure 4 illustrates the family of Sun's Ultra-Port architectures (5) used in its workstations. These systems use a combination of bus and crossbars to implement shared memory. In the smaller Ultra 450 system (one to four processors), illustrated in Fig. 4(a), a centralized coherency controller and a crossbar are used to connect the processors directly to the shared memory. As this is a low-cost, single-board configuration, this simple architecture meets its requirements. The intermediate-sized Ultra 6000 system

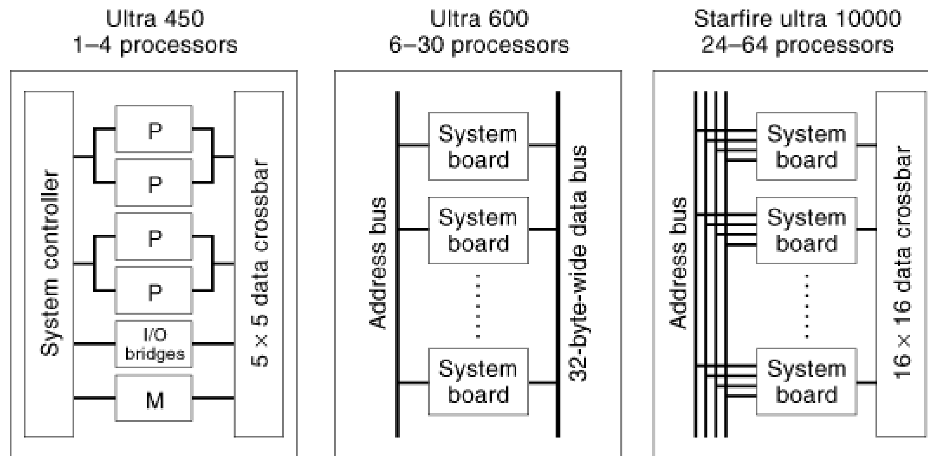

Fig. 4. Three Ultra-Port architecture implementations: (a) a small system consisting of a single board with four processors, I/O interfaces, and memory; (b) a medium-sized system with one address bus and a wide data bus between boards; (c) a large system with four address buses and a data crossbar between boards. [Source: A. Charlesworth (5).]

has a Gigaplane bus that interconnects multiple system boards and is designed to provide a broad range of expandability with the lowest possible memory latency, typically 216 ns for a load miss.

This scheme supports systems with 6 to 30 processors as shown in Fig. 4(b). For large systems, with 24 to 64 processors, the address bus is replicated by a factor of four. The scheme is illustrated in Fig. 4(c). These four address buses are interleaved so that the memory addresses are statically divided among the four buses, that is, each address bus covers one-quarter of the physical address space. A 16 × 16 crossbar is chosen to match the quadrupled snoop rate. To avoid the failures on one system board from affecting other boards, and to electrically isolate the boards, point-to-point router application-specific integrated circuits (*ASICs*) are used for the entire interconnect, that is, for the data crossbar, the arbitration interconnect, and the four address buses. The ASICs are mounted on a centerplane, which is physically and electrically in the middle of the system.

Physically Distributed Memory Architectures

The structure of a typical distributed-memory multiprocessor system is shown in Fig. 5. This architecture enables scalability by distributing the memory throughout the machine, and using a scalable interconnect to enable processors to communicate with the memory modules. Based on the communication mechanism provided, these architectures are classified as:

- Multicomputer message-passing architectures
- DSM architectures

The computers in multicomputer architectures use a software (message-passing) layer to communicate among themselves, and hence they are named message-passing architectures. In these systems, programmers are required to explicitly send messages to request or send remote data. As these systems connect multiple computing nodes, sharing only the scalable interconnect, they are also referred to as multicomputer systems.

DSM machines logically implement a single global address space, although the memory is physically distributed. The memory access times in these systems depended on the physical location of the processors

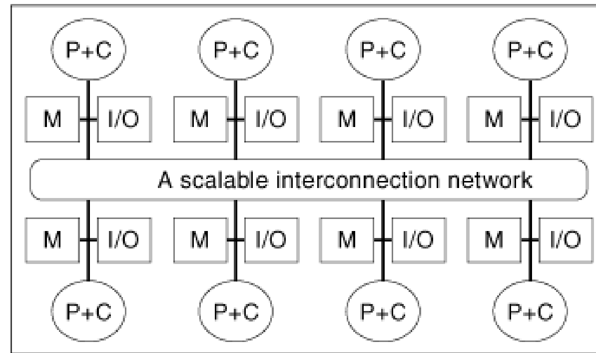

Fig. 5. Distributed-memory multiprocessors (P+C: processor + Cache; M: memory). Both message-passing systems and distributed shared memory (DSM) systems have the same basic organization. The key distinction is that the DSMs implement a single shared address space, whereas the message-passing architectures have distributed address space.

and are no longer uniform. As a result, these systems are also termed nonuniform memory access (*NUMA*) systems.

Classification of Distributed Shared-Memory Systems. Providing DSM functionality on physically distributed memory requires the implementation of three basic mechanisms:

(1) *Processor-Side Hit–Miss Check* This operation, on the processor side, is used to determine whether a particular data request is satisfied or not in the processor's local cache. A *hit* is a data request satisfied in the local cache, while a *miss* requires the data to be fetched from main memory or the cache of another processor.
(2) *Processor-Side Request Send* This operation is used on the processor side, in response to a miss, to send a request to another processor or main memory for the latest copy of the relevant datum and wait for a response.
(3) *Memory-Side Operations* These operations enable the memory to receive a request from a processor, perform any necessary coherence actions, and send its response (typically in the form of the requested data).

Depending on how these mechanisms are implemented in hardware or software helps classify the various DSM systems as follows:

- *Hardware-Based DSM Systems.* In these systems, all processor-side mechanisms are implemented in hardware, while some part of memory-side support may be handled in software. Hardware-based DSM systems include SGI Origin (7), HP/Convex Exemplar (8), MIT Alewife (9), and Stanford FLASH (10).
- *Mostly Software Page-Based DSM Systems.* These systems implement hit–miss check in hardware by making use of virtual-memory protection mechanisms to provide access control. All other support is implemented in software. Coherence units in such systems are the size of virtual-memory pages. Mostly software page-based DSM systems include TreadMarks (2), Brazos (4), and Mirage+ (11).
- *Software Object-Based DSM Systems.* In this class, all three mechanisms mentioned above are implemented entirely in software. Software-object-based DSM systems include Orca (1), SAM (12), CRL (13), Midway (3), and Shasta (14).

Almost all DSM models employ a directory-based cache coherence mechanism, implemented in either hardware or software. DSM systems have demonstrated the potential to meet the objectives of scalability,

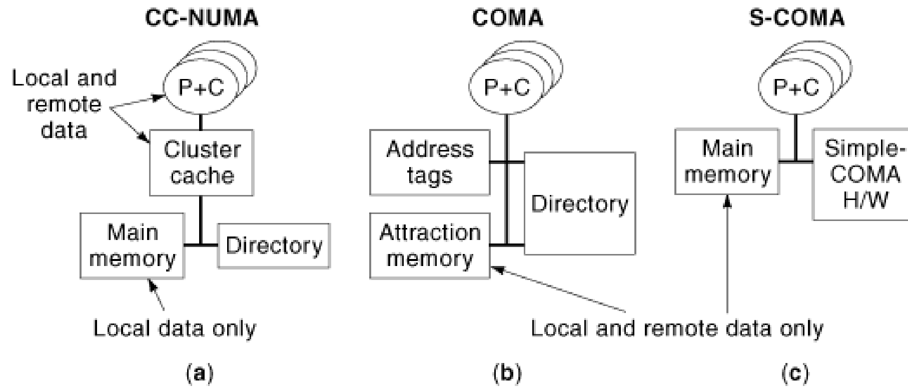

Fig. 6. Processor memory hierarchies in CC-NUMA, COMA, and S-COMA (P+C: processor + cache; H/W: hardware).

ease of programming, and the cost-effectiveness (7, 8). The directory-based coherence makes these systems highly scalable. The globally-addressable-memory model is retained in these systems, although the memory access times depend on the location of the processor and are no longer uniform. In general, hardware DSM systems allow programmers to realize excellent performance without sacrificing programmability. Software DSM systems typically provide a similar level of programmability. These systems, however, trade off somewhat lower performance for reduced hardware complexity and cost.

Hardware-Based Distributed Shared-Memory Systems. Hardware-based DSM systems implement the coherence and consistency mechanisms in hardware, making them faster but more complex. Clusters of symmetric multiprocessors (*SMP*s), with hardware support for shared memory, have emerged as a promising approach to building large-scale DSM parallel machines. Each node in these systems is an SMP with multiple processors. The large volumes of these small-scale parallel servers make them extremely cost-effective as building blocks. Software compatibility is preserved through a directory-based cache coherence protocol. This also helps support a shared-memory abstraction despite having memory physically distributed across the nodes. A number of different cache coherence protocols have been proposed for these systems. These include (1) cache-coherent nonuniform memory access (*CC-NUMA*), (2). cache-only memory access (*COMA*), (3), simple cache-only memory access (*S-COMA*), (4). reactive NUMA, and (5). adaptive S-COMA. Figure 6 illustrates the processor memory hierarchies in CC-NUMA, COMA, and S-COMA architectures.

Cache-Coherent Nonuniform Memory Access. Figure 6(a) shows the processor memory hierarchy in a CC-NUMA system. In this system, a per-node cluster cache lies next to the processor cache in the hierarchy. Remote data may be cached in a processor's cache or the per-node cluster cache. Memory references not satisfied by these hardware caches must be sent to the referenced page's home node to obtain the requested data and perform necessary coherence actions. The first processor to access a remote page within each node results in a software page fault. The operating system's page-fault handler maps the page to a CC-NUMA global physical address and updates the node's page table. The Stanford DASH and SGI Origin systems implement the CC-NUMA protocol.

Cache-Only Memory Access. The key idea in COMA architecture is to use the memory within each node of the multiprocessor as a giant cache (also termed as attraction memory) as shown in Fig. 6(b). Data migration and replication are done just as in caches. The advantage of this scheme is the ability to capture the remote-capacity misses as hits in the local memory; that is, if a datum is initially allocated in a remote memory and is frequently used by a processor, it can be replicated in the local memory of the node where it is being frequently referenced. The attraction memory maintains both the address tags as well as the state of the data.

The COMA implementation requires customized hardware and hence has not become a popular design choice. The Kendall Square Research KSR1 (15) machine implemented COMA architecture.

Simple Cache-Only Memory Access. An S-COMA system, shown in Fig. 6(c), uses the same coherence protocol as CC-NUMA, but allocates part of the local node's main memory to act as a large cache for remote pages. S-COMA is much cheaper and simpler to implement than COMA, as it can be built with off-the-shelf hardware building blocks. It also uses standard address translation hardware. On a first reference to a remote page from any node, a software page fault occurs, which is handled by the operating system. It initializes the page table and maps the page in the part of main memory being used as cache. The essential extra hardware required in S-COMA is a set of fine-grained access control bits (one or two per block) and an auxiliary translation table. The S-COMA page cache, being part of main memory, is much larger than the CC-NUMA cluster cache. As a result, S-COMA can outperform CC-NUMA for many applications. However, S-COMA incurs substantial page overhead as it invokes the operating system for local address translation. Additionally, programs with large sparse data sets suffer from severe internal fragmentation, resulting in frequent mapping and replacement (or swapping) of the S-COMA page caches, a phenomenon called *thrashing*. In such applications CC-NUMA may perform better. Since S-COMA requires only incrementally more hardware than CC-NUMA, some systems have proposed providing support for both protocols. For example the S3.mp (19) project at Sun Microsystems supports both S-COMA and CC-NUMA protocols.

Hybrid Schemes. Given these diverse application requirements, hybrid schemes such as R-NUMA (16) and adaptive S-COMA (*ASCOMA*) (17) have been proposed. These techniques combine CC-NUMA and S-COMA to get the best of both with incrementally more hardware. These schemes have not yet been implemented in commercial systems.

Reactive Nonuniform Memory Access. R-NUMA dynamically reacts to program and system behavior to switch between CC-NUMA and S-COMA. The algorithm initially allocates all remote pages as CC-NUMA, but maintains a per-node, per-page count of the number of times a block is refetched as a result of conflict or capacity miss. When the refetch count exceeds a threshold, the operating system intervenes and reallocates the page in the S-COMA page cache. Thus, based on the number of refetches, R-NUMA classifies the remote pages as *reuse* pages and *communication* pages and maps them as CC-NUMA and S-COMA respectively. A CC-NUMA page is upgraded to be an S-COMA page if the refetch count exceeds a threshold.

Adaptive Simple Cache-Only Memory Access. The ASCOMA scheme proposes a page allocation algorithm that prefers S-COMA pages at low memory pressures, and a page replacement algorithm that dynamically backs off the rate of page remappings between the CC-NUMA and S-COMA modes at high memory pressures.

ASCOMA initially maps pages in S-COMA mode. Thus, when memory pressure is low, S-COMA suffers neither remote conflict nor capacity misses; nor does it pay the high cost of remapping. ASCOMA reacts to increase in memory pressure by evicting cold pages (i.e., pages not accessed for a long) from and remapping hot pages (i.e., pages frequently accessed) to the local page cache. It adapts to differing memory pressures to fully utilize the large page cache at low memory pressures and avoids thrashing at high memory pressures. The adaptivity is implemented by dynamically adjusting the refetch threshold that triggers remapping, increasing it when memory pressure is high.

SGI Origin. The SGI Origin implements a scalable shared-memory multiprocessing (*SSMP*) architecture, with DSM and a directory-based cache-coherence protocol. Origin is designed to minimize the latency difference between remote and local memory and to include hardware and software support to ensure that most memory references are local. Figure 7 shows a block diagram of the Origin architecture. The basic building block of the Origin system is a dual-processor node. In addition to the processors, a node contains up to 4 Gbyte of main memory and its corresponding directory memory, and has a connection to a portion of the I/O subsystem. The architecture supports up to 512 nodes, for a maximum configuration of 1024 processors and 1 Tbyte of main memory. The nodes can be connected together via any scalable interconnection network. The cache-coherence protocol employed by the Origin system does not require delivery of point-to-point messages and allows maximum flexibility in implementing the interconnect network.

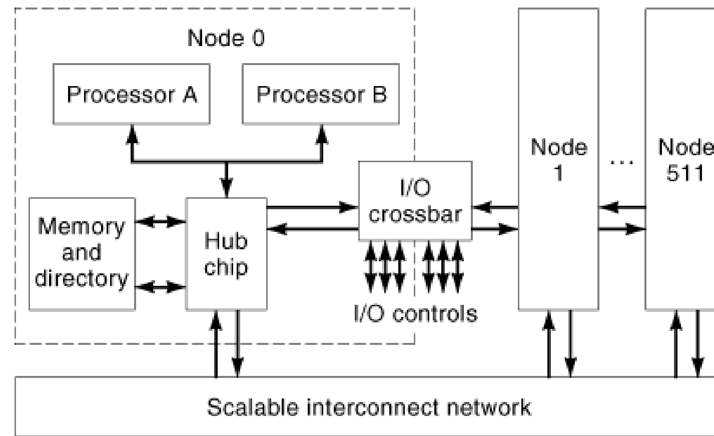

Fig. 7. Block diagram of Origin. [Source: Laudon and Lenoski (7).]

The DSM architecture provides global addressability to all memory in the system. While the two processors on a node share the same bus, they do not function as a snoopy cluster. Instead, they operate as two separate processors multiplexed over the single physical bus. This is unlike in many other CC-NUMA systems, where the node is a SMP cluster. Such architecture helps reduce both local and remote latencies and increase memory bandwidth. Thus both the absolute memory latency and the ratio of remote to local memory latencies is kept at a minimum.

Other CC-NUMA features provided in the Origin system include combinations of hardware and software support for page migration and replication. These include per-page hardware memory reference counters, a block-copy engine that copies data at near-peak memory speeds, mechanisms for reducing the cost of translation look-aside buffer (*TLB*) updates, and a high-performance local and global interconnect design. Furthermore, the cache coherence protocol minimizes latency and bandwidth per access with a rich set of synchronization primitives.

The intranode interconnect consists of a single hub chip that implements a full four-way crossbar between processors, local memory, the I/O, and network interfaces. The global interconnect is based on six-port router chip configured in a multilevel fat-hypercube topology. The coherence protocol supports a clean exclusive state (i.e., data are consistent with memory and are not shared) for cached data to minimize latency on read–modify–write operations. It allows cache dropping of clean exclusive or shared data without notifying the directory in order to minimize the effect of directory coherence on memory/directory bandwidth.

HP/Convex Exemplar. The HP/Convex Exemplar (8) combines the parallel scalability of message-passing architecture with hardware support for DSM, global synchronization, and cache-based latency management. Figure 8 shows the Exemplar architecture, which supports a global CC-NUMA structure. The global system organization comprises up to 16 SMPs interconnected by four scalable coherent (SCI) interface ring networks. The ring network is called the *convex toroidal interconnect*.

The symmetric multiprocessor, or hypernode, uses a 5×5 - crossbar switch to interconnect four functional blocks. Each block has two 200 MHz HP PA-RISC 7100 microprocessors, 1 Mbyte of data and instruction caches, 64 Mbyte of memory, cache management, and memory controllers, an interface to the internal switch, and one of the four external coherent toroidal interconnect (*CTI*) ring networks. The Exemplar supports both message-passing and shared-memory programming models. The parallel virtual machine (*PVM*) (18) and a message-passing interface (*MPI*) (19) are both supplied as system software. Physical memory is organized in 64-byte blocks interleaved across memory banks in each hypernode, with two such banks of memory per functional

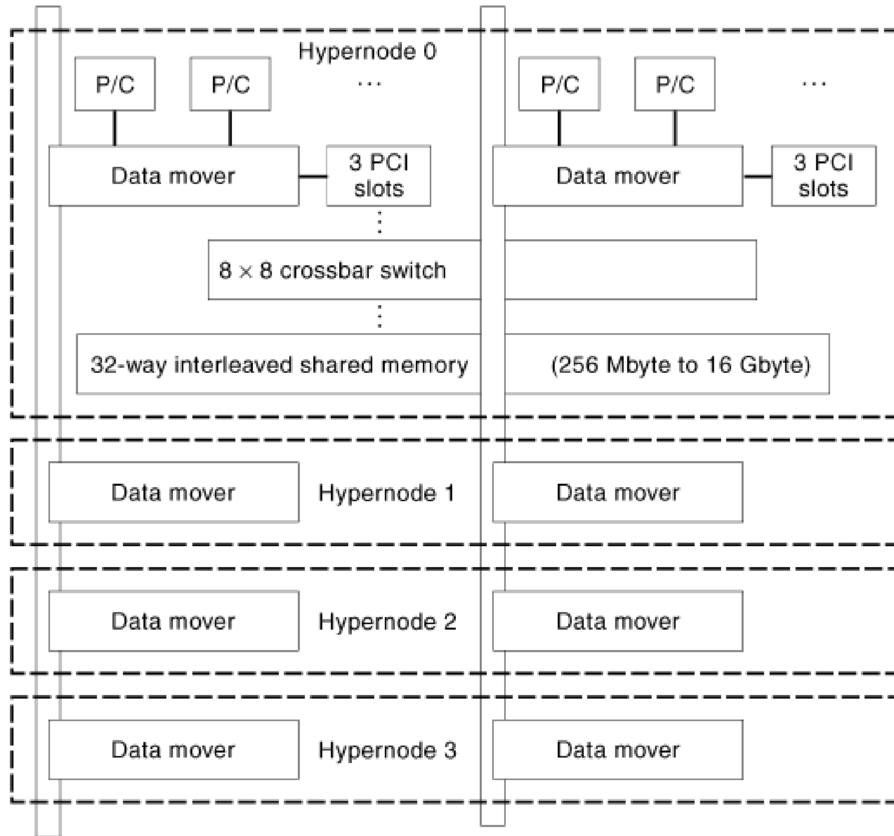

Fig. 8. Architecture of the HP/Convex Exemplar X-Class SPP (P/C: Processor/cache; CTI:coherent toroidal interconnect: PCI: peripheral component interconnect. [Source: Brewer and Astfalk (8).]

block. Physical memory is partitioned into three areas: One is globally accessible, the second is accessible only within the node, and the third is used as cache buffer for data shared among the hypernodes. Global data-transfer requests cross through the switch (8 × 8 crossbar switch) within the hypernode and exit through the appropriate ring network to the target external hypernode. The Exemplar's two-tiered cache-coherency mechanism uses a directory-based full-mapped scheme within each hypernode. The coherency mechanism is a distributed linked list that identifies the copy state of each global block. A partition of physical memory, the CTI cache, is used to maintain the portion of the linked list and the values of variables copied from other hypernodes.

IBM RP3. Figure 9 shows a structural block diagram of the IBM RP3. The original design of the RP3 had two distinct networks between processors and memory modules—one a conventional switching network designed for low latency and high bandwidth, and the other a combining network (20) (i.e. the network that combines the accesses to the same remote memory module). The idea behind using two networks was to direct the noncombinable accesses to the fast, conventional network, while the combinable accesses were routed through the combinable network. The higher latency of the combining network was applicable only to the requests that could be combined; the majority of the requests were not affected by the additional latency. In later designs, the combining network was dropped from the implementation because of cost. The switching network, as shown in Fig. 9, has inputs and outputs on the same side. The global memory is spread among

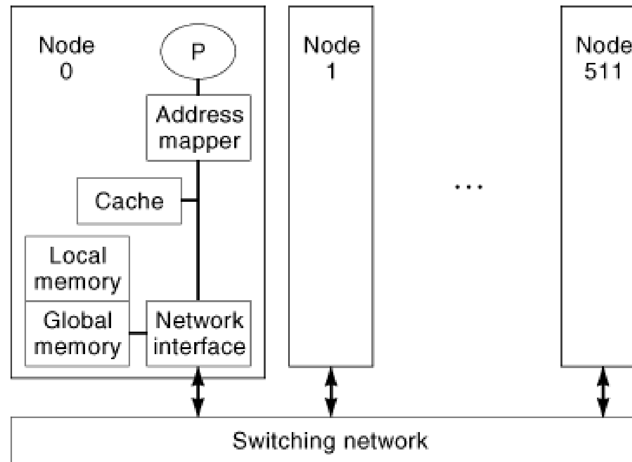

Fig. 9. Block diagram of the IBM RP3.

the processors so that each processor has one independent block of memory, some part of which is used as global memory while the remainder is used for local data. The processor is connected to the network through an address mapper, a cache, and an interface for routing requests to local or global memory or to the network. The mapper maps memory addresses to the cache and the local and/or global memory units.

The global address space in the RP3 is evenly distributed across all modules. This is done to balance access requests across the modules and is implemented by using the least significant bits of memory address to specify the module that contains the data. As a result, references to items close to each other in the logical address space are scattered more or less uniformly to all physical memory modules. Local memory, however, is treated in a different way, as it has to be physically close to its associated processor; hence it uses the most significant, and not the least significant, bits to select a physical memory. Thus, items that lie close to each other in the address space of local memory lie in the same physical memory module.

Local data in the RP3 are not private to a processor, that is, they are accessible to other processors. The main objective of the local address space is, however, to store local, unshared data. The RP3 has an additional degree of freedom in that the boundary between the local and global address subspaces is software-controllable.

The MIT Alewife Machine. The MIT Alewife machine (9) is an example of a CC-NUMA shared-memory programming environment on a scalable hardware base. Figure 10 shows an overview of the MIT Alewife architecture. Each node consists of a processor, a floating-point unit, 64 kbyte of direct-mapped cache, 8 Mbyte of DRAM, a network router, and a custom-designed communication and memory management unit (*CMMU*). The nodes can communicate using either shared memory or message passing via the single-chip CMMU. The CMMU is the heart of an Alewife node and is responsible for coordinating message passing and shared-memory communication. It implements a scalable cache-coherence protocol and provides the processor with a low-latency network interface.

Shared memory is distributed in the sense that the shared address space is physically partitioned among nodes. Cache lines in Alewife are 16 bytes in size and are kept coherent through a software extended directory protocol. Each of the 16 byte memory lines has a home node that contains storage for its data and coherence directory. All coherence operations for given memory line, whether handled by hardware or by software, are coordinated by its home node. Each node contains the data and coherence directories for a 4 Mbyte portion of shared memory.

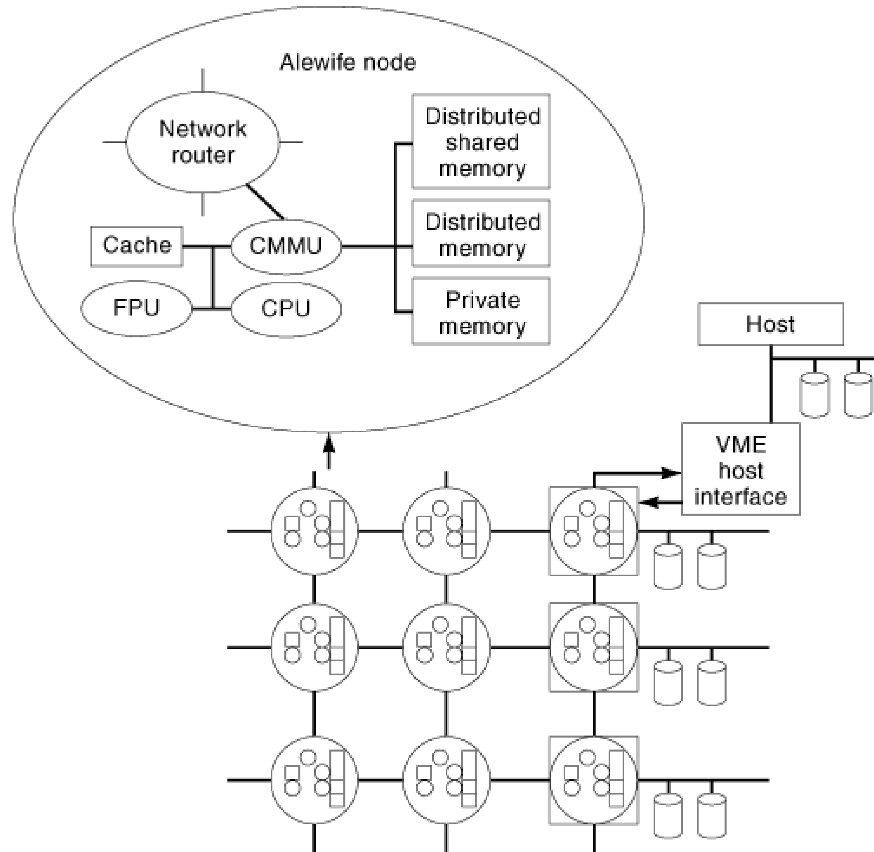

Fig. 10. The Alewife architecture (CMMU: communication and memory management unit; FPU: floating-point unit).

Alewife provides four classes of architectural mechanisms that implement an automatic locality management strategy, which seeks to maximize the amount of local communication by consolidating related blocks of computation and data, and attempts to minimize the effects of nonlocal communication when it is unavoidable. The four classes are:

- *Coherent Caches for Shared Memory.* Though the systems' physical memory is statically distributed over the nodes in the machine, Alewife provides programmers the abstraction of globally shared memory. The memory hardware helps manage locality by caching both private and shared data on each node.
- *Fine-Grained Computation.* Alewife supports fine-grained computation by including fast user-level messages.
- *Integrated Message Passing.* While the programmer sees a shared-memory programming model, for performance reasons much of the underlying software is implemented using message passing. The hardware supports a seamless interface.
- *Latency Tolerance.* The mechanisms of block multithreading and prefetching attempt to tolerate the latency of interprocessor communication when it cannot be avoided. These mechanisms require caches that continue to supply instructions and data while waiting for the prefetched data or during miss (called *lockup-free* caches).

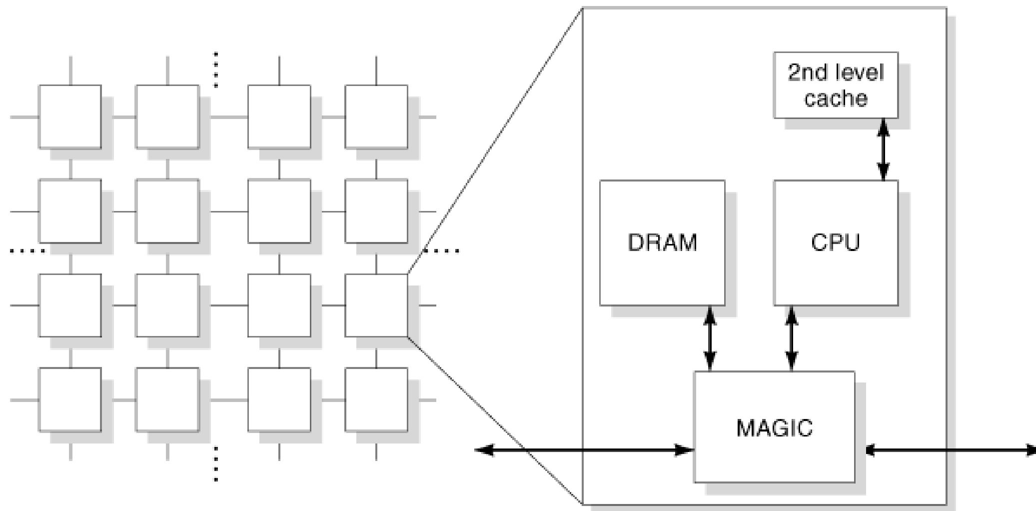

Fig. 11. FLASH system architecture. [Source: Ofelt et al. (10).]

The MIT Alewife machine implements a complete programming environment consisting of the hardware, compiler, and operating system, all combined to achieve the goal of programmability by solving problems like scheduling computation and moving data between processing elements. Features of this environment include: globally shared address space, a compiler that automatically partitions regular programs with loops, a library of efficient synchronization and communication routines, distributed garbage collection, and a parallel debugger.

The Stanford FLASH Multiprocessor. The Stanford FLASH multiprocessor (10), like Alewife, emphasizes efficient integration of both cache-coherent shared memory and low-overhead user-level message passing. FLASH, as shown in Fig. 11, is a single-address-space machine consisting of a large number of processing nodes connected by a low-latency, high-bandwidth interconnection network. Every node is identical, containing a high-performance off-the-shelf microprocessor and its caches. These caches form a portion of the machine's distributed memory on a node-controller chip MAGIC (for Memory and General Interconnect Controller). The MAGIC chip forms the heart of the node, integrating memory controller, I/O controller, network interface, and a programmable protocol processor. This integration allows for low hardware overhead while supporting both cache coherence and message-passing protocols in a scalable and cohesive fashion. MAGIC includes a programmable protocol processor, which offers flexibility. The hardwired data movement logic achieves low latency and high bandwidth by supporting highly pipelined data transfers without extra copying within the chip. MAGIC separates data-movement logic from protocol state-manipulation logic, which ensures that it does not become a latency or bandwidth bottleneck. FLASH's base cache coherence protocol is directory-based and has two components: a scalable directory data structure and a set of handlers. For the scalable directory structure FLASH uses dynamic pointer allocation, wherein each cache-line-sized block (128 bytes) of main memory is associated with an 8 byte state word called the *directory header*. This header is stored in a contiguous section of main memory devoted solely to the cache coherence protocol. A significant advantage of dynamic pointer allocation is that the directory storage requirements are scalable. Overall the directory occupies 7% to 9% of main memory, depending on the system configuration.

Mostly Software Page-Based Distributed Shared-Memory Systems. An alternative approach, making use of software for implementation, has seen the evolution of quite a number of page-based DSM systems. These techniques make use of the virtual-memory hardware in the underlying system to implement the shared-memory consistency models in software to resolve conflicting memory accesses (memory accesses to

the same location by different processors, at least one of which is a write access). Examples of mostly software page-based DSM systems include TreadMarks (2), Brazos (4), and Mirage+ (11).

The advantage of page-based DSM systems is that they eliminate the shared-memory hardware requirement, making them inexpensive and readily implementable. These systems are found to work well for dense matrix codes (2). As the coherence policy is implemented in software, it can be optimized to make use of the operating system to implement coherence mechanisms. The use of the operating system, however, makes them slow compared to hardware coherence mechanisms. Additionally, the coarse sharing granularity (large page size) results in false sharing and longer communication time per page. One solution is to have multigrain systems, using fine-grain shared memory within an SMP and page-based distributed-shared memory across the SMPs.

A key issue in page-based DSM systems is *write protocols:*

Write-Update and Write-Invalidate Protocols There are two approaches to maintaining the memory coherence requirement. One approach is to ensure that a processor has an exclusive access to a datum before it writes that datum. This type of protocol is called a *write-invalidate* protocol, because it invalidates all other copies on a write. This is by far the most common protocol. The other alternative is to update all the cached copies of a datum when it is written. This type of protocol is called a write-update protocol.

Single- and Multiple-Writer Protocols Most hardware cache and DSM systems use single-writer protocols. These protocols allow multiple readers to access a given page simultaneously, but a writer is required to have sole access to a page before performing any modifications. Single-writer protocols are easy to implement because all copies of a given page are always identical, and page fault can always be satisfied by retrieving a copy of the page from any other processor that currently has a valid copy. This simplicity often comes at the expense of high message traffic. Before a page can be written, all other copies must be invalidated. These invalidations can then cause subsequent access misses if the processors whose pages have been invalidated are still accessing the page's data. *False sharing* occurs when two or more unrelated data objects are located in the same page and are written concurrently by separate processors. Since the consistency unit (usually a virtual-memory page) is large in size, false sharing is a potentially serious problem and causes the performance of the single-writer protocol to further deteriorate due to interference between unrelated accesses. Multiple-writer protocols allow multiple processors to have a writable copy of the page at the same time.

TreadMarks. TreadMarks (2) supports parallel computing on networks of workstations (*NOW*s) by providing the application with a shared-memory abstraction. The TreadMarks application programming interface (*API*) provides facilities for process creation and destruction, synchronization, and shared-memory allocation. Synchronization, a way for the programmer to express ordering constraints between the shared-memory accesses of different processes, is implemented with *critical sections*. TreadMarks provides two synchronization primitives: barriers and exclusive locks. Barriers are global in the sense that on calling a barrier, a process is stalled until all the processes in the system have arrived at that barrier. In the case of locks, a lock-acquire call acquires a lock for the calling process, and a lock-release call releases it.

TreadMarks uses a *multiple-writer* protocol. The shared page is initially write-protected. When a write occurs in a processor (say P1), TreadMarks creates a copy of the page, or *twin*, and saves it as a part of the TreadMarks data structure on P1. It then unprotects the page in the user's address space so that further writes to that page occur without software intervention. Later on, when P1 arrives at a barrier, there is an unmodified twin and a modified copy in the user's address space. By making a word-by-word comparison of the two, a run-length encoding of the modifications of the page, called a *diff*, is created. Once the diff is created, it is sent to all the processors sharing that page. These processors then modify the page, discarding the twin. The same

sequence of events takes place on every other processor. Once the diff is received, the entire sequence of events is local to each processor and does not require any message exchanges, unlike in single-writer protocols.

Brazos. Brazos (4) is a page-based DSM that makes use of relaxed consistency models and multithreading on a network of multiprocessor computers. It executes on x86 multiprocessor workstations running, Microsoft Windows NT 4.0. Brazos is based on selective multicast in a time-multiplexed network environment such as Ethernet. Selective multicast is used in Brazos to reduce the number of consistency-related messages and to efficiently implement its version of scope consistency. One disadvantage with multicast is the potential harmful effect of unused indirect diffs (i.e. run-length encoding of the modifications of a page). Although receiving multicast diffs for inactive pages does not increase network traffic, it does cause processors to be interrupted frequently to process incoming multicast messages. These messages and subsequent changes are not accessed before the next time that page is invalidated; thus, they detract from user-code computation time. The dynamic copyset reduction mechanism ameliorates this effect by allowing processes to drop out of the copyset for a particular page. This causes them to be excluded from multicast messages providing diffs for the page.

Brazos uses multithreading at both user level and DSM system level. Multiple user-level threads allow applications to take advantage of SMP servers by using all available processors for computation. The Brazos run-time system has two threads. One thread is responsible for quickly responding to asynchronous requests for data from other processes and runs at highest possible priority. The other thread handles replies to requests previously sent by the process.

Brazos implements a version of the *scope consistency* model, which is a bridge between the release consistency model and the entry consistency model. The scope consistency seeks to reduce the false sharing present in page-based DSM systems. False sharing occurs when two or more threads modify different parts of the same page of data, but do not actually share the same data element. This leads to unnecessary network traffic. Scope consistency divides the execution of a program into global and local scopes, and only data modified within a single scope are guaranteed to be coherent at the end of that scope. Brazos implements software only scope consistency that requires no additional hardware support.

Mirage+. Mirage+ (11), developed at University of California, Riverside, allocates time windows during which processors at a node possess a page. At the end of the time window the node may be interrupted to relinquish the page. During the time window, processes at the site(s) having read-only access may read, and processes at the site having write access may read or write, the page. The page may also be unused during the time window. Thus, the time window provides some degree of control over the processor *locality* (the number of references to a given page that a processor will make before another processor is allowed to reference that page). Mirage+ is a write-invalidate coherent system, that is, a store requires that all read-only copies of a page be invalidated before storing to the page with the referenced location.

Mirage+ defines one distinguished site, called the *library* site. Requests for pages are sent to the library site, queried, and sequentially processed. All pages must be checked out from the library. Another distinguished site is the *clock* site. It is the site that has the most recent copy of a page. The library site records which site is acting as the clock site. The process of converting a reader to a writer when a page fault occurs is called an *upgrade*. The process of converting a writer to a reader is called a *downgrade*.

Mirage makes use of performance improvement techniques in a networked environment, such as

high-level packet blasting and *compression* High-level packet blasting eliminates the overhead of explicitly handshaking each packet, thus significantly improving the total time for a remote page. Compression works by reducing the number of packets that the system must transmit at each page fault.

All-Software Object-Based Distributed Shared-Memory Systems. In the all-software (object-based) approach, shared memory is entirely supported in software. Orca, SAM, Midway, CRL, and Shasta are examples of this approach. Shasta is unique in that it uses a fine-grained approach.

Orca. Orca (1) defines an object-based and language-based DSM model. It encapsulates shared data in objects and allows programmers to define operations on these objects, using abstract data types. The Orca language supporting this model was designed specifically for parallel programming on DSM systems. Orca integrates synchronization and data accesses, giving the advantage that programmers, while developing parallel programs, do not have to use explicit synchronization primitives.

Orca migrates and replicates shared data (objects) and supports an *update* coherence protocol for implementing write operations. Objects are updated using function shipping, that is, the operation and its parameters are sent to all machines containing a copy of the object to be updated. The operation is then applied to the local copies. To ensure that the replicated copies are updated in a coherent manner, the operation is sent using totally ordered group communication. All updates are executed in the same order at all machines, so that sequential consistency is guaranteed. Orca is implemented entirely in software and requires the operating system (or hardware) to provide only basic communication primitives. This flexibility of being an all-software system is exploited to implement several important optimizations.

Portability is achieved in Orca by using a layered approach. The system contains three layers, and the machine-specific parts are isolated in the lowest layer. This layer (called Panda) implements a virtual machine, which provides the communication and multitasking primitives needed at run time. Portability of Orca only requires portability of Panda.

SAM. Stanford SAM (12) is a shared-object system for distributed shared-memory machines. SAM has been implemented as a C library. It is a portable run-time system that provides a global name space and automatic caching of shared data. SAM allows communication of data at the level of user-defined data types, thus allowing user control over communication in a DSM machine. The basic principle underlying SAM is to require the programmer to designate the way in which data are to be accessed. There are two kinds of data relationships (hence synchronization) in parallel programs: *values*, with single assignment constraints, and *accumulators*, which allow mutually exclusive accesses to the requesting processors. Values make it simple to express producer–consumer relationships or precedence constraints; any read of value must wait for the creation of the value. Accumulators allow automatic migration of data to the requesting processors, making sure that the data accesses are mutually exclusive.

SAM incorporates mechanisms to address the problems of high communication overheads; these mechanisms include tying synchronization to data access, chaotic access to data, prefetching of data, and pushing of data to remote processors. SAM deals only with management and communication of shared data; data that are completely local to a processor can be managed by any appropriate method. The creator of a value or accumulator should specify the type of the new data. With the help of a preprocessor, SAM uses this type information to allocate space for data, pack the messages, unpack them, and free the storage of the data. The preprocessor can handle complex C data types.

An important mechanism for tolerating communication latency is support for asynchronous access. SAM provides the capability to fetch values and accumulators asynchronously. An asynchronous fetch succeeds immediately if a copy of the value is available on the local processor. If the value is not immediately available, the fetch operation returns an indication of unavailability and the requesting process can proceed with other access or computation. The requesting process is notified when the value becomes available on the local processor. For an asynchronous access to an accumulator, the process is notified when the accumulator has been fetched to the local processor and mutual exclusion has been obtained.

Midway. Midway (3), at Carnegie Mellon University, is also an object-based DSM programming system supporting multiple consistency models within a single parallel program. Midway contains data that may be processor-consistent, release-consistent, or entry-consistent.

Midway programs are written in C, and the association between synchronization objects and data must be made with explicit annotations. Midway requires a small amount of compile-time support to implement its consistency protocols; for example, whenever the compiler generates its code to store a new value into a shared datum, it also generates code that marks the item as *dirty* in an auxiliary data structure. Distributed

synchronization management, implemented in Midway, enables processors to acquire synchronization objects not presently held in their local memories. Two types of synchronization objects are supported: locks and barriers. Locks are acquired in either exclusive or nonexclusive mode by locating the lock's owner using a distributed queuing algorithm. Distributed cache management ensures that a processor never enters a critical section without having received all updates to the shared data guarded by that synchronization object (a lock or a barrier). Midway implements entry consistency with an update-based protocol, thereby requiring interprocessor communication only during acquisition of synchronization objects. Entry consistency guarantees that shared data become consistent at a processor when the processor acquires a synchronization object known to guard the data.

CRL: The C Region Library. CRL (13) is an all-software DSM model that is system- and language-independent. It is portable and employs a region-based approach. Each region is an arbitrary-sized contiguous area of memory identified by a unique region identifier. CRL is implemented entirely as a library. CRL requires no functionality from the underlying hardware, compiler, or operating system beyond that necessary to send and receive messages. CRL considers entire operations on regions of data as individual units and provides sequential consistency for the read and write operations. In terms of individual loads and stores, CRL provides memory coherence through entry or release consistency. CRL employs a fixed-home, directory-based write-invalidate protocol.

CRL is able to use part of main memory as a large secondary cache instead of relying only on hardware caches, which are typically small. Regions, chosen to correspond to user-defined data structures, assist coherence actions to transfer exactly the data required by the application.

Fine-Grained Shasta Distributed Shared Memory. Fine-grained sharing is an alternative all-software approach proposed to overcome both false sharing and unnecessary transmission. Shasta (14) a fine-grained, all-software DSM, was developed at Western Research Laboratory. It supports coherence at fine granularity and thus alleviates the need for complex mechanisms for dealing with false sharing typically present in software page-based DSM systems. To reduce the high overheads associated with software message handling, the cache coherence protocol is designed to minimize extraneous coherence messages. It also includes optimizations such as nonblocking stores, detection of migratory data sharing, issuing multiple load misses in a batch, merging of load, sharing misses to the same line, and support for prefetching and home-placement directives.

Shared data in Shasta have three basic states:

(1) *Invalid* The datum is not valid on this processor.
(2) *Shared* The datum is valid on this processor, and other processors have copies of it.
(3) *Exclusive* The datum is valid on this processor, and no other processor has a copy of it.

Communication is required if a processor attempts to read data that are in invalid or shared state. This is called a shared miss.

The shared address space in Shasta is divided into ranges of memory, called *blocks*. The block size can be different for different ranges of the shared address space (i.e. for different program data). The line size is configurable at compile time and is typically set to 64 or 128 bytes. The size of each block must be a multiple of the fixed line size. Coherence is maintained using a directory-based invalidation protocol, which supports three types of requests: *read, read exclusive*, and *exclusive* (or upgrade). Supporting exclusive requests is an important optimization, since it reduces message latency and overhead if the requesting processor has the line in shared state. Shasta supports three types of synchronization primitives: locks, barriers, and event flags.

A home processor is associated with each virtual page of shared data, and each processor maintains directory information for the shared-data pages assigned to it. The protocol maintains the notion of an *owner* processor for each line, which corresponds to the last processor that maintained an exclusive copy of the line. Directory information consists of two components: a pointer to the current owner processor, and a full-bit vector

of the processors that are sharing the data. The protocol supports dirty sharing, which allows the data to be shared without requiring the home node to have an up-to-date copy. A request coming to the home node is forwarded to the current owner as an optimization; this forwarding is avoided if the home processor has a copy of the data.

To avoid the high cost of handling messages via interrupts, messages from other processors are serviced through a polling mechanism. Polls are also inserted at every loop back edge to ensure reasonable response times.

The protocol aggressively exploits the release consistency model by emulating the behavior of a processor with nonblocking stores and lockup-free caches.

Concluding Remarks

Shared-memory machines built with symmetric multiprocessors and clusters of distributed multiprocessors are becoming widespread, both commercially and in academia (1,3,4,5,7,8,9,10,11,13,16,19, 21). Shared-memory multiprocessors provide ease of programming while exploiting the scalability of distributed-memory architectures and the cost-effectiveness of SMPs. They provide a shared-memory abstraction despite having memory physically distributed across the nodes. Key issues in the design of the shared-memory multiprocessors are cache-coherence protocols and shared-memory consistency models, as discussed. The SMPs typically use snoopy cache-coherence protocols, whereas the DSM machines are converging towards directory-based cache coherence. The More popular consistency models include sequential consistency, release consistency, and scope consistency, each addressing different design goals.

High-level optimizations in a programming model, such as a single global address space and low-latency access to remote data, are critical to the usability of shared-memory multiprocessors. However, they trade off directly with the scalability of architecture and the operating system's performance. In shared-memory multiprocessors, the architecture, operating system, and programming model are heavily interdependent. Current shared-memory multiprocessors are built to achieve very high memory performance in bandwidth and latency (5, 7). An important issue that needs to be addressed is the I/O behavior of these machines. The performance of distributed I/Os and distributed file systems on a shared-memory abstraction needs to be addressed in future designs.

BIBLIOGRAPHY

1. H. E. Bal *et al.* Performance evaluation of the orca shared object system, *ACM Trans. comput. syst.*, **16** (1): 1–40, 1998.
2. C. Amza *et al.* TreadMarks: Shared Memory Computing on Networks of Workstations, *IEEE Computer*, **29** (2): 18–28, 1996.
3. B. Bershad M. Zekauskas W. Swadon The Midway distributed shared memory system. *IEEE Int. Computer Conf. (COMPCON)*, 1993.
4. E. Speight J. K. Bennett Brazos: A third generation DSM system, *Proc. 1997 USENIX*.
5. A. Charlesworth STARFIRE: Extending the SMP envelope, *IEEE MICRO*, **18** (1): 39–49, 1998.
6. *Sun Enterprise X000 Server Family: Architecture and Implementation*, [Online]. Available WWW: http://www.sun.com/servers/white-papers/

7. J. Laudon D. Lenoski The SGI Origin: A ccNUMA Highly Scalable Server http://www-europe.sgi.com/origin/
8. T. Brewer G. Astfalk The evolution of HP/Convex Exemplar, *Proc. IEEE computer Conf. (COMPCON), Spring*, 1997.
9. A. Agarwal *et al.* The MIT Alewife machine: Architecture and performance, *Proc. 22nd Int. Symp. on Computer Architecture (ISCA)*, 1995.
10. J. D. Ofelt *et al.* The Stanford FLASH multiprocessor, *Proc. 21st Int. Symp. on Computer Architecture*, 1994.
11. B. D. Fleisch R. L. Hyde N. Christian Mirage+: A kernel implementation of distributed shared memory for a network of personal computers. *Softw. Pract. Experience*, 1994.
12. D. J. Scales M. S. Lam The design and evaluation of a shared object system for distributed memory machines, *Proc. First Symp. on Operating Systems Design and Implementation*, 1994.
13. K. L. Johnson M. Kaashoek D. Wallach CRL: High-performance all-software distributed shared memory, *Proc. 15th ACM Symp. Operating Systems Principles (SOSP '95)*, 1995.
14. D. J. Scales K. Gharachorloo A. Aggarwal Fine-grain software distributed shared memory on SMP clusters, Research Report 97/3, 1997.
15. H. Burkhardt III *et al.* Overview of the KSR1 computer system, Tech. Rep KSR-TR-9202001, Kendall Square Research, Boston, 1992.
16. B. Falsafi D. A. Wood. Reactive NUMA: A design for unifying S-COMA and CC-NUMA, *Proc. 24th Int. Symp. on Computer Architecture (ISCA)*, 1997.
17. C. Kuo *et al.* ASCOMA: An adaptive hybrid shared memory architecture, *Int. Conf. on Parallel Processing (ICPP'98)*, 1998.
18. A. Geist *et al.* PVM 3 user's guide and reference manual, Technical Report ORNL/TM-12187, Oak Ridge National Laboratory, 1993.
19. Message Passing Interface Forum, MPI : A message passing interface standard, 1994.
20. H. S. Stone *High-Performance Computer Architecture*, 2nd ed., Reading, MA: Addison-Wesley, 1990.
21. B. Verghese *et al.* Operating system support for improving data locality on CC-NUMA computer servers, *Proc. 7th symp. on Architectural Support for Programming Languages and Operating Systems (ASPOLS VII)*, 1996.
22. A. Saulsbury A. Nowatzyk Simple COMA on S3. MP, *Proc. 1995 Int. Symp. on Computer Architecture Shared Memory Workshop*, 1995.

READING LIST

A. Saulasbury *et al.* Simple COMA node implementations, *Proc. 27th Hawaii Int. Conf. on System Sciences*, 1994.

NARENDRA SHAHA
MANISH PARASHAR
Rutgers, The State University of New Jersey