Chapter Three

Hierarchical Management of DSM systems

Distributed shared memory has been implemented [Li89b, Leno93, Agar95, Kusk94], using directory-based cache coherence algorithms [Agar88, Leno90]. With single-level distributed shared memory schemes, there is a fixed size block, which is the unit of mapping, the unit of transfer, and, thus, the granularity of sharing. If the block is ‘‘large’’ (several thousand bytes), the scheme may be refered to as a page-level scheme. If the block is ‘‘small’’ (cache block size — tens of bytes), the scheme may be refered to as a block-level scheme. With a page-level scheme [Li89b], the large unit of sharing results in long delays for interprocessor communication even when the object being (logically) transferred is small. In addition, the large unit of mapping results in reduced system performance due to unnecessary data movement caused by ‘‘false sharing’’ [Bolo89]. Hence, page-level DSM schemes do not provide adequate support for fine-grain parallelism.

The difficulties with page-level coherency management motivate the use of block-level schemes. In these schemes, the unit of transfer is a block. Different blocks of a page can be simultaneously mapped to the memories of different nodes. With schemes that support sharing at the fine granularity of cache blocks [Chai90, Leno90] the number of blocks that must be managed is very large. The mapping tables required to manage the large number of blocks leads to unacceptably high storage overhead.

We propose hierarchical management as a solution to the problems associated with page-level and block-level DSM schemes. Our scheme is based on the assumption, which is validated by execution-driven and trace-driven simulations, that at any one time only a small percentage of the address space of every process is actually shared. With hierarchical management, the mapping tables are partitioned into tables for strictly local, non-shared pages and tables for shared pages. Special fine-grain support is provided only in the tables for shared pages. These special tables make it possible to specify different locations for different blocks in the page. Since the shared pages are few, this reduces storage overhead significantly over providing fine-grain support for all pages. Pages are dynamically identified as private or shared. During normal execution, pages are dynamically moved between the different mapping tables, as the need arises. The proposed scheme introduces the use of *ownership* [Katz85] at both the page level and the block level in order to provide block-level sharing and also support dynamic page migration to reduce message traffic and the size of mapping tables.

The basic system organization and single-level management of DSM are discussed in Section 3.1. A brief review of Li's [Li89b, Li89a] page-level scheme is presented in Section 3.1.1. A simple block-level scheme is presented in Section 3.1.2. The sizes of tables required for block-level management of DSM are computed in Section 3.1.4. The proposed two-level management scheme is described in Section 3.2. The storage overhead of the scheme is evaluated in Section 3.2.3. Other schemes for multi-level management of DSM are discussed in Section 3.3.

3.1. Single-Level Management of DSM

A high-performance node of a message-passing multicomputer consists of: an application processor, local memory, cache memory to speed up access to the local memory, a memory management unit (MMU) that manages translation and protection for the local memory, and a communication coprocessor that handles most of the tasks involved with sending, receiving, and forwarding messages [Atha88, Chow88].

Distributed shared memory can be supported with single granularity of management on a multicomputer with a number of these nodes. When an address requested by the processor is not available locally, the node should send a message to a remote node requesting a copy of the data. When a node receives such a remote request, the granule of data that was addressed should be accessed from local memory and sent to the requesting node. Mapping information must be maintained for each granule of data. Also, the nodes must implement a coherence protocol to maintain each granule of data coherent.

DSM can be supported by relying on the application processor to perform all the necessary operations [Li89b, Bolo89]. However, for maximum system performance, the application processor should not be involved in the ‘‘emulation’’ of shared memory — special-purpose hardware should service remote requests for pages and convert local accesses to non-local pages into request messages to remote nodes. Just as the communication coprocessor handles the various tasks involved in message passing, a dedicated coprocessor, called a memory-to-messages / messages-to-memory transducer (MMT) will provide support for shared memory emulation. The resulting organization of the multicomputer node is shown in Figure 3.1. The virtual cache satisfies most of the local processor's memory requests. The MMU performs the address translation in the event of a cache miss or a remote request. A translation lookaside buffer (TLB) in the MMU is used to speed up the translation.

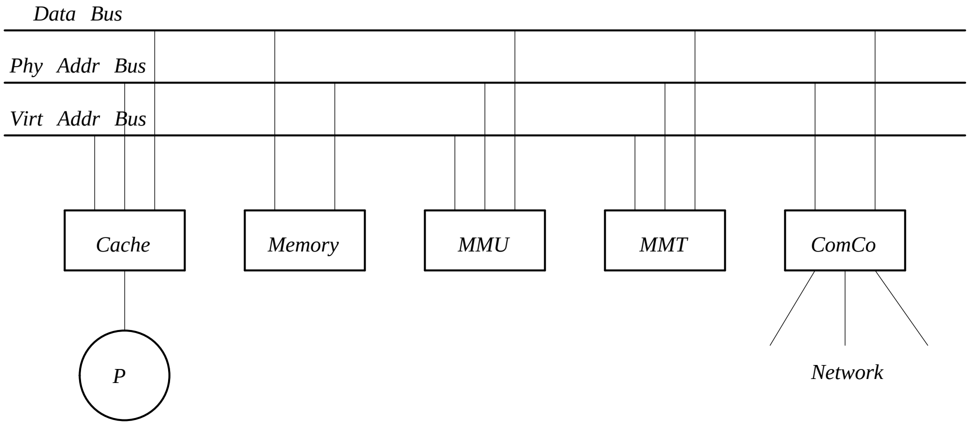


Figure 3.1: A multicomputer node.

Upon request from the MMU, the MMT composes messages and forwards them to the ComCo for transmission to the network. On receipt of a message from the network, the ComCo writes the message either into physical memory or into the MMT. Messages that are responses to requests sent out earlier into the network, are written into a physical memory location specified by the MMU at the time of the request, using DMA. Unsolicited messages such as write-backs, invalidates, remote requests are written to the MMT. The MMT coordinates with the MMU in servicing messages passed on by the ComCo, and writes/reads the physical memory using DMA. The virtually addressed cache snoops on the transactions between the MMU and MMT to maintain its data coherent with those in physical memory.

Each node may be time-shared between multiple processes. A set of processes that share their address space forms a *task*. The system may be executing multiple tasks simultaneously. Unique task identifiers partition the total virtual system space into disjoint task virtual spaces. Processes belonging to different tasks execute in disjoint address spaces and cannot share data.

Single-level management of DSM can be implemented with large granules the size of a ‘‘page’’ or with smaller granules the size of a ‘‘cache block.’’ As discussed earlier, the performance of single-level managed DSM and the storage overhead associated with it are dependent on the size of the granule. We discuss single-level management at the granularity of ‘‘pages’’ and at the granularity of ‘‘cache blocks’’ in the next two sub-sections.

3.1.1. Management at the Granularity of ‘‘Pages’’

Distributed Shared Memory where sharing and coherency are managed at the single granularity of *pages* (thousands of bytes) can be implemented using the normal page fault mechanism of the MMU [Li89b, Bolo89]. Li [Li86a, Li89b] has implemented coherent shared virtual memory on networked workstations and on a multicomputer [Li89a]. The scheme is directory-based [Agar88] where, with the most promising of the proposed alternatives, the directory is distributed among the nodes. Nodes must obtain exclusive ownership of a page before modifying it, invalidating all other copies of the page. Multiple read-only copies can be distributed throughout the system. With the ‘‘distributed fixed manager algorithm’’ each virtual page is initially managed by a *default manager* whose identity is determined by a simple function of the virtual page number.

3.1.2. Management at the Granularity of ''Cache Blocks''

As discussed earlier, the use of pages as the unit of mapping and the unit of transfer can result in low performance due to the long latencies and the bandwidth requirements for transferring pages and due to false sharing. A simple solution to this problem is to use the same basic algorithms with much smaller pages, henceforth referred to as *blocks*, as the unit of mapping, transfer, and ownership [Chai90, Leno93, Kusk94]. Such a scheme requires address translation to map any virtual block number to a local block frame, a remote node, or secondary storage. Unfortunately, in a large multicomputer, the total number of blocks in all the virtual address spaces of all the tasks in the system will be too large for address translation based on conventional page tables (block tables).

The above problem can be solved by using an *inverted table* [Chan88] for translating virtual addresses to local physical addresses. The inverted block table on each node, called the *present block table* (PBT), contains one entry for each block frame in the local physical memory (Figure 3.2). Hence, the size of the PBT on each node is proportional to the size of the physical memory, rather than virtual memory, on the node. Hashing is used to provide fast associative lookups in the PBT [Chan88]. If a local or remote memory access does not ''hit'' in the PBT, the default owner (manager) of the block is computed and a remote request for the block is sent to that processor. The default owner of a block may refer to a Migrated Block Table (MBT) to determine whether the block has migrated to another processor or if it is on disk.

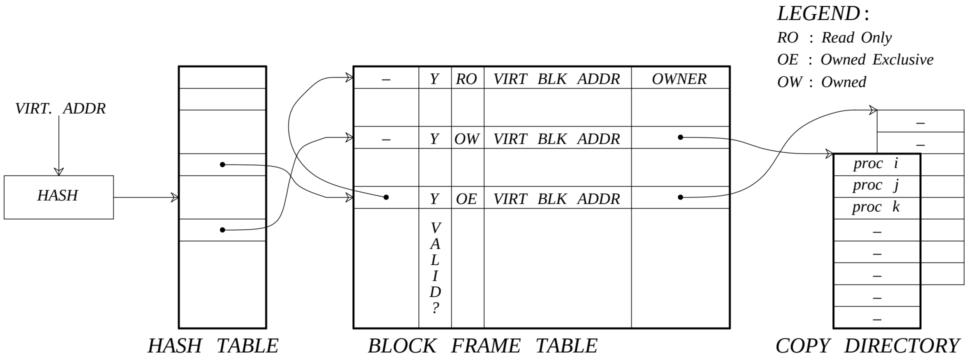


Figure 3.2: Address translation tables for the block-level (one-level) scheme.

3.1.3. Limited Copy Directories

If a block is in read-only state, its PBT entry specifies the block owner. If the block is owned non-exclusively, the same field in the PBT is used to point to the *copy directory*, that contains the identities of the processors that possess read-only copies of that block [Li89b]. This information is used for invalidating all read copies prior to any write to the block. A very large copy directory may be needed to accommodate blocks that are read shared by many processors. There are several approaches to limiting the size of the copy directory. With broadcast protocols, if the number of read copies exceeds some fixed bound, a system wide broadcast is required to invalidate a block [Agar88]. The disadvantage of this approach is that many multicomputer interconnection networks do not provide efficient support for broadcasts. An alternative to broadcast protocols is to invalidate one of the read-only copies of the block whenever a new read request is received by the block owner and the size of the copy directory has already reached its limit. With such an *eviction* policy [Chai90], upon receipt of a read request that would ‘‘overflow’’ the copy directory, the block owner picks a victim from the copy directory, invalidates it, and uses the freed slot in the copy directory for the new request.

The performance impact of limiting the size of the copy directory has been evaluated based on the access-burst model [Dubo88]. Critical section accesses and the locality of references in shared blocks are modeled by assuming that accesses by one processor to a shared writable block are done in independent uninterrupted bursts called access bursts. An access burst can modify the block with a fixed write probability w. If a copy directory overflow requires a block to be invalidated, the performance degradation due to the limited number of entries in the copy directory can be measured as the increase in the frequency of invalidates over an infinite copy directory. For an unlimited copy directory, the average number of access bursts before an invalidate is $\sum_{i=1}^{i=\infty} i\,(1-w)^i\, w$, which reduces to $(1 - w)/w$.

With the limited copy directory, invalidations can be caused by write access bursts and by copy directory overflows. To derive an expression for the average number of access bursts before an invalidate for the limited copy directory, we partition the problem into three cases:

1) when the number of access bursts before invalidate is *less* than the maximum number of entries in the copy directory,
2) when the number of access bursts before invalidate is *greater than or equal* to the maximum number of entries in the copy directory and the invalidate is caused by a write access burst, and
3) when the number of access bursts before invalidate is *greater than or equal* to the maximum number of entries in the copy directory and the invalidate is

caused by a copy directory overflow.

In case 1, the invalidate is only due to a write access burst. Hence, the contribution of case 1 to the average number of access bursts before invalidate is $\sum_{i=1}^{i=v-1} i(1-w)^i w$ where v is the maximum number of entries in the copy directory. The invalidate in case 2 is also caused only by a write access burst. However, since the number of access bursts in case 2 is greater than the maximum number of entries in the copy directory, this case is possible only when utmost v processors participate in the access bursts, irrespective of the number of access bursts. The contribution of case 2 to the average number of access bursts before invalidate can be expressed as

$$\sum_{i=v}^{i=\infty} i(1-w)^i \left[\sum_{j=1}^{j=v} P(j,i,N)w \right]$$

N is the number of processors sharing the block and $P(p,b,N)$ is the probability that exactly p out of the N processors have copies of the block after b read bursts. The inner summation in this expression is the probability that utmost v processors participate in a sequence of i access bursts. Case 3 accounts for the only other scenario possible — when the invalidate is caused by a copy directory overflow. For this case, exactly v processors must have already participated in the access bursts when a read access burst is initiated by a new processor. The contribution of case 3 to the average number of access bursts before invalidate can be expressed as

$$\sum_{i=v}^{i=\infty} i(1-w)^i P(v,i,N)(1-w)\frac{N-v}{N}$$

$P(v,i,N)$ is the probability that the copy directory is fully occupied after i access bursts. $(1-w)\frac{N-v}{N}$ is the probability of a read access burst by a processor not in the copy directory.

The average number of access bursts before an invalidate for a limited copy directory B is the sum of the contributions of the three cases. Hence, for N processors in the system, and a write burst probability of w, the average number of access bursts before an invalidate for a copy directory with v entries $B(N, w, v)$ is

$$\sum_{i=1}^{i=v-1} i(1-w)^i w + \sum_{i=v}^{i=\infty} i(1-w)^i \left[\sum_{j=1}^{j=v} P(j,i,N)w + P(v,i,N)(1-w)\frac{N-v}{N} \right]$$

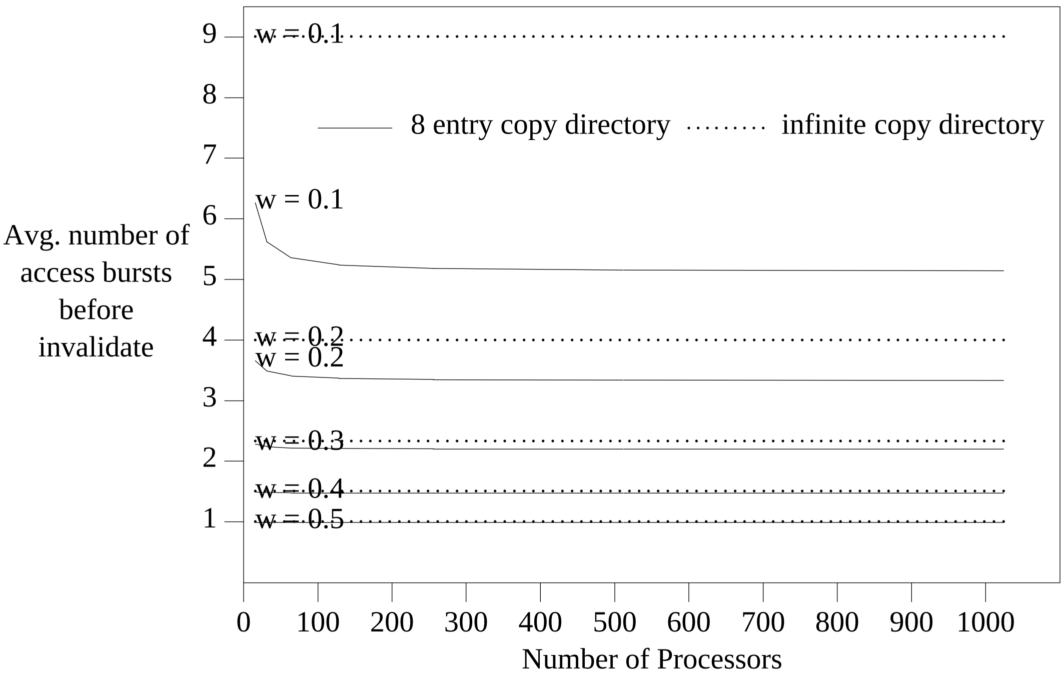


Figure 3.3: Number of Access Bursts prior to invalidate for the infinite and the finite copy directories.

Figure 3.3 shows the relationship between the number of processors and the average number of access bursts before an invalidate for various values of w, for both and infinite copy directory and for a copy directory limited to eight entries. For write probability ranges observed in practical applications (on the order of 0.3) [Bayl89, Dubo88], the figure indicates that a copy directory with a maximum of eight entries performs nearly as well as the infinite copy directory.

3.1.4. Storage Overhead Example

As discussed above, large mapping tables and directories are a critical problem with single level management at a fine granularity. As an example of the sizes of the mapping tables and directories, consider a multicomputer with 1024 nodes, each with 128 MBytes of physical memory. The virtual address space is 1 TB (2^{40}). A page in this system is 1 KB and a block 32 bytes. The PBT on each node has 4 M entries, each about 82 bits wide. Hence, the PBT requires 40 MB of the local memory. With 8 entries in the copy directory of each of the 4 M blocks in the PBT, the total size of the copy directory in one processor would be approximately 40 MB. The MBT does not have to be resident in local memory but its total size can become unmanageable.

3.2. Hierarchical Management of Distributed Shared Memory

With uniform block-level sharing, all ownership and mapping information is at the granularity of blocks. For every block in physical memory, space is allocated to maintain its access rights and copy directory. The hierarchical scheme proposed in this section improves upon the one-level scheme by optimizations based on the following two observations. First, a large portion of physical memory is used exclusively by the local processor for long periods of time. Hence, the one-level scheme is wasteful in allocating storage space for access rights and copy directory for each block in the local memory. Second, private segments are likely to occur in contiguous chunks much larger than a block. A private page, that would require only one mapping entry in a page level scheme, would need as many entries as the number of blocks per page, in a block level scheme. Thus, the block level scheme also results in an unnecessarily large number of mapping and directory entries for private segments of the memory.

3.2.1. The Two-Level Management Scheme

The hierarchical DSM management scheme distinguishes between private and shared segments (pages) in physical memory and allocates space for coherency state information and mapping only for the shared pages. The private segments are managed at a page level granularity. Pages dynamically change state, on demand, between private and shared status, transparent to the application. To eliminate the overhead that is caused by false sharing, ownership of shared segments is maintained at both the page and block levels.

A private page located on its default owner (manager) does not require storage of any extra mapping information. On a remote request for a block of a privately owned page, the page changes to the shared state. For each shared page in the system, information is maintained regarding two levels of ownership: page level and block level. The processor owning the shared page maintains the block ownership information for all blocks of that page. The semantics of ownership of a block of a shared page are the same as ownership in the one-level scheme. Specifically, the processor owning a block can provide read-only copies, invalidate read-only copies, or transfer ownership of the block to another processor. If the block owner provides read-only copies of a block, it must maintain the copy directory for that block.

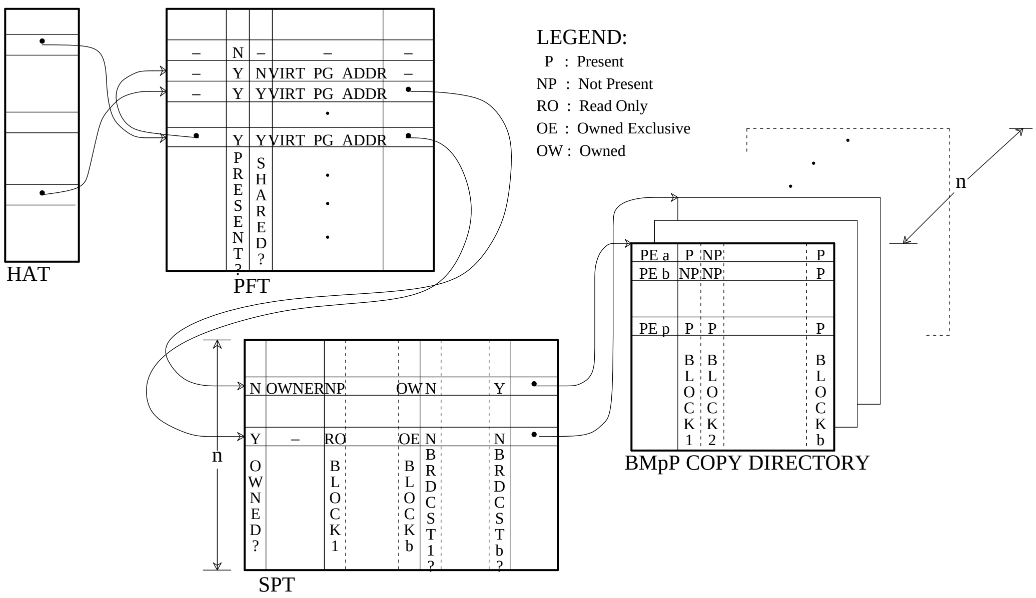


Figure 3.4: Address translation tables for the two-level scheme.

Initially, all pages are private. A remote request for a block of the page changes its state to shared. When a page enters the shared state, all of its blocks are owned by the page owner. Ownership of individual blocks may be transferred to other nodes that need to modify the block. Hence, the owner of a shared page does not necessarily possess copies of all blocks belonging to that page. As with sector mapping in cache memories [Lipt68], space for the entire page must be allocated in the local memory of every processor that has any blocks from that page.

The address translation tables are organized in a two-level hierarchy (Figure 3.4). The first level table indicates which virtual page is mapped in each page frame of the physical memory and whether the page is shared. On a memory request, the first level table identifies if the page containing the block being requested is present in local memory. If the page is private, the block is guaranteed to be present. If the page is shared, the second level table indicates whether the requested block is present with sufficient access permission. For every owned shared page, the second level table maintains the identity of each block owner. If the page is present and shared, but the block is absent, the request is forwarded to the page owner, or to the block owner if this processor is the page owner. As in the one-level scheme, on a page miss the request is forwarded to the default owner. A Migrated Page Table (MPT) maintains the identity of the current owner of default pages that have been migrated.

The first level table, the *present pages table* (PPT), is organized as an inverted table, consisting of a hash anchor table (HAT) and a page frame table (PFT) [Chan88]. Since the PPT maps pages rather than blocks, it is smaller than the PBT in the one-level scheme. For each page in the physical address space, the PFT maintains the virtual page number, a bit to indicate if the page is shared or private and, if the page is shared, a pointer to a second level table entry. The address translation for a private page need not access the second level table.

The second level table, *shared page table* (SPT) maintains information about shared pages. The SPT entry maintains the access rights to each block in the page as well as a flag indicating whether the page is owned locally. If the page-owned bit is not set, the page owner field holds the identity of the processor owning the page. Two state bits for every block in a shared page indicate the access rights to the block: invalid, read only, owned non-exclusive, and owned exclusive (write permission) [Katz85]. Each SPT entry includes a field which may contain a pointer to the copy directory. If the page is owned, the copy directory must indicate the owners of each block which is not owned locally. For every block owned non-exclusively, the copy directory indicates all the nodes that have read-only copies of the block. If neither the page nor the block are owned by the local node, messages destined to the block owner are sent to the the *page* owner, specified in the SPT. The page owner may then forward such messages to the block owner. As in the one-level scheme, the copy directory size can be limited by following a policy of eviction or broadcast. The organization of the copy directory is discussed in Subsection 3.2.2. If the broadcast policy is used, each SPT entry includes one *broadcast bit* per block (Figure 3.4).

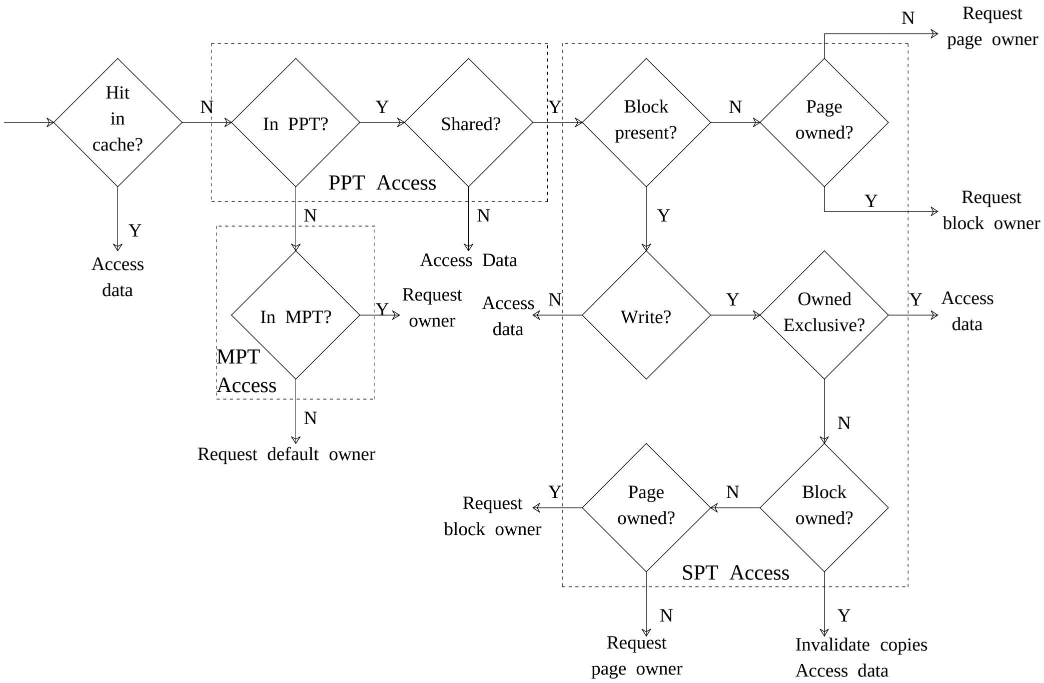


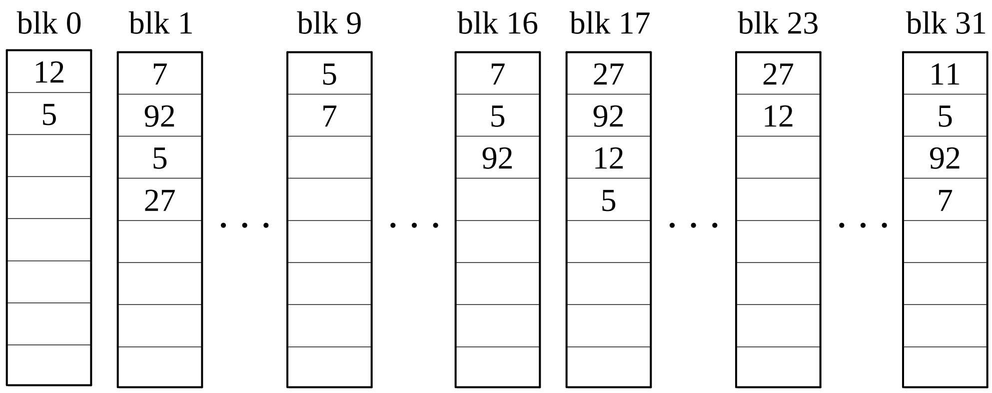
Figure 3.5: Address translation for local requests.

A key to the efficiency of the two-level scheme is that the required number of entries in the SPT is significantly smaller than the number of entries in the PPT. In many parallel applications the proportion of references to shared storage is less than 15% [Bayl89]. Furthermore, only part of the shared space is actually shared during each ‘‘phase’’ of the program. Hence, an SPT with significantly fewer entries than the PPT should be sufficient to accommodate the ‘‘shared working set’’ of most applications. When the SPT ‘‘overflows,’’ an entry can be freed up by flushing an un-owned page back to its owner or by ‘‘recalling’’ all the blocks of an owned page and making the page private. If page migration is supported, an SPT entry may be freed by migrating an owned page to another node which already owns some of its blocks. We expect an SPT which can accommodate about 10% of the working set of the process, roughly 10% of the entries of the PPT, to be sufficient (see Section 5.1.3 in Chapter 5).

Figure 3.5 shows the sequence of actions of the address translation mechanism for local requests. Remote requests are handled similarly but without the initial lookup in the local cache. The messages exchanged between the nodes to maintain coherency are presented in Section 4.3.3 in Chapter 4.

3.2.2. The Copy Directory

As discussed in Section 3.1.2, in most cases the maximum number of entries in the copy directory for each shared block can be bounded with little performance degradation. With the two level scheme, there may be a need to maintain copy directory entries for every block in every shared page. Using a conventional organization of a copy directory [Agar88], a limited number of processor identifiers is maintained for each block of the page (Section 3.1.2). We call this organization



Figures 3.6: Processor List per Block (PLpB) copy directory organization. Each page has 32 blocks, and 8 copy directory entries are maintained for each block.

the *Processor List per Block* (PLpB) organization. This organization is illustrated in Figure 3.6.

For the hierarchical coherency management scheme, we propose a new organization for the copy directory, called the *Block Map per Processor* (BMpP) organization (Figure 3.7). This organization minimizes the required storage for the copy directories based on the assumption that two nodes that share one block of a page are likely to share other blocks from that page. For each shared *page* there is a limited number of BMpP entries. Each entry identifies one remote processor that shares one or more blocks of that page. In each entry there is a bit vector (map) with one bit per block indicating whether the particular remote node has a copy of the corresponding block. The maximum number of entries in the copy directory per shared page is the maximum number of nodes that may share a page at the same time.

With the PLpB organization, if a processor shares several blocks of a page,

PE id	Bit vector of block access permissions																														
92	0	1							0							1	1						0								1
5	1	1	0	0	0	0	0	0	1	0	0	0	0	0	0	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	1
27	0	1							0							0	1						1								0
11	0	0							0							0	0						0								1
•																															
•																															
•																															

Figures 3.7: Block Map per Processor (BMpP) copy directory organization. Each page has 32 blocks.

the processor identifier is stored multiple times for that one page (for example, node 5 in Figure 3.6). Since every entry in the BMpP copy directory keeps track of all the blocks of the page, the processor identifier is stored once so the total storage needed for good performance is expected to be significantly smaller than with the PLpB copy directory (see Section 5.1.4 in Chapter 5). As with the limited PLpB organization, when a BMpP copy directory is full and a new entry is needed, an existing entry can be evicted by recalling all the blocks in the associated remote node. If necessary, it is possible to use a broadcast bit per block (Figure 3.4) with the limited BMpP directories to deal with blocks which are shared by a large number of processors [Agar88].

3.2.3. Storage Overhead of the Two-Level Management Scheme

Coherency Scheme	Size of Mapping Tables per Node			
	PPT/PBT	SPT	Copy Dir.	Total
Conventional One-Level Scheme	40 MB	-	40 MB	80 MB
One-Level scheme (10% shared blocks)	40 MB	-	6.75 MB	46.75 MB
Hierarchical Scheme (10% shared blocks)	1.28 MB	272 KB	2.75 MB	4.25 MB

Table 3.1: Comparison of the storage overhead in the one-level and two-level schemes. The comparison is shown for a system with 1024 nodes, 1 TB of virtual address space, and 128 MB of physical memory per node. The one-level scheme uses 32-byte blocks. The hierarchical scheme uses 1 KB pages and 32-byte blocks.

With the system parameters discussed in Section 3.1.2, the Present Pages Table (PPT) has just 128 K entries as opposed to 4 M entries in the Present Blocks Table (PBT) discussed in Section 3.1.2. As a result, the total size of the PPT is only 1.28 MB. As discussed in Subsection 3.2.1, the number of entries in the second-level Shared Pages Table (SPT), and thus the number of BMpP copy directories can be significantly smaller than the number of PPT entries. With the broadcast bit shown in Figure 3.4, each SPT entry requires 134 bits of storage. Hence, approximately 272 KB will be required for 16 K entries.

With the proposed Block Map per Processor (BMpP) organization of the copy directory, given a fixed memory size and page size, the total storage needed for the copy directory is sensitive to the size of the block. Larger blocks lead to smaller entries (fewer blocks per page), and thus reduce the memory used for copy

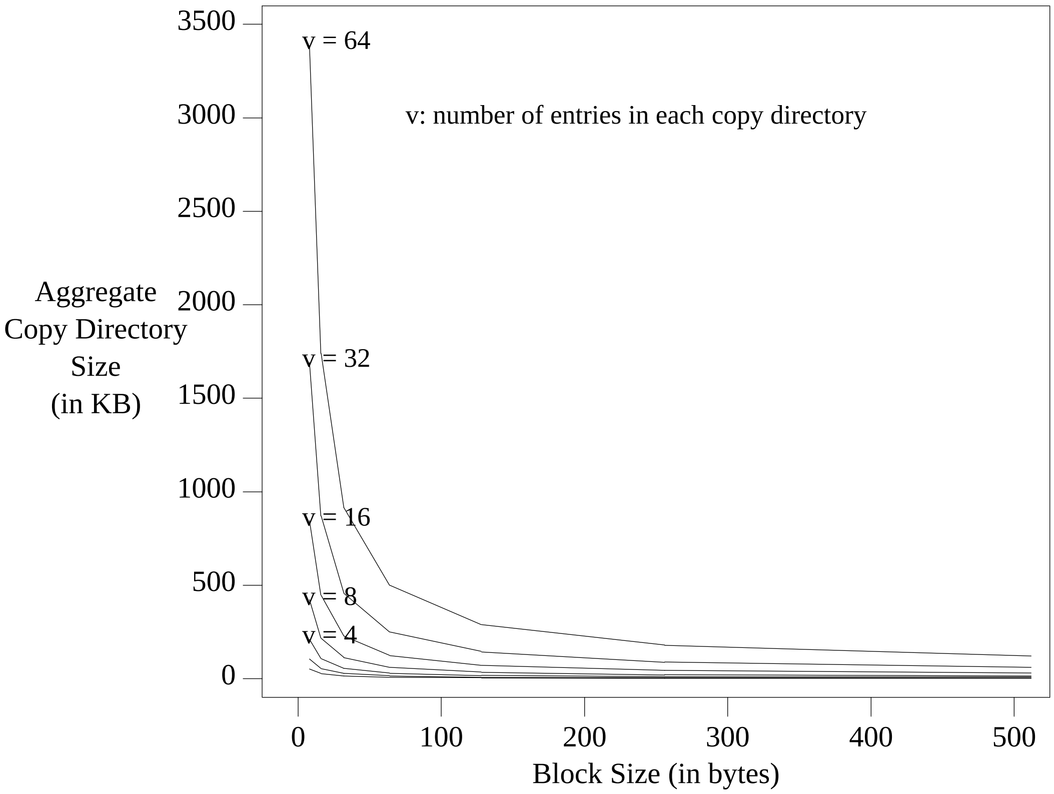


Figure 3.8: Total size of all the copy directories with the BMpP organization on a node versus the granularity of sharing. 1K entries in the SPT. 4 KB pages. 32 MB local memory. The parameter *v* is the number of BMpP copy directory entries per shared page. The total memory required for the copy directories increases rapidly for block sizes below 64 bytes.

directory storage. However, small blocks are needed to minimize false sharing. For the example system, Figure 3.8 shows the size of the copy directory per node versus the block size for different number of entries in the copy directory per shared page. With a block size of 32 bytes and 32 entries per copy directory, the total copy directory size per node is about 2.75 MB.

In the one-level scheme, if all the blocks are allowed to be shared, the total size of memory resident tables — PBT and the PLpB copy directory — is approximately 80 MB (62.5% of local memory) If 10% of the blocks may be shared, the total size of tables is approximately 46.75 MB (36.5% of local memory) (see Table 3.1). With the two-level scheme, the total size of memory resident tables — PPT, SPT, and the BMpP directory — is about 4.25 MB, consuming less than 4% of the local memory.

3.3. Related Work

Two of the key ideas in our proposed scheme have also been proposed by O'Krafka and Newton [O'Kr90]: 1) managing coherency at two levels of granularity, block and sub-block, and 2) exploiting the fact that not all the blocks are actually shared at each point in time. Their sectored directory scheme [O'Kr90] manages data at two levels — a block and a sub-block, similar to sector-mapping in uniprocessor caches. Ownership is associated with each sub-block. The identity of the owner and a modified bit are maintained for each sub-block. To minimize the directory storage overhead, the full-map copy directory is maintained only for each block (instead of each sub-block, as in our scheme). As a result, the sectored directory scheme does not fully exploit the idea of multi-level management since it requires invalidating all the sub-blocks of a block when only one sub-block needs to be invalidated. This limitation results in poor performance [O'Kr90].

The second scheme proposed in [O'Kr90] avoids maintaining a copy directory for every memory block. Instead, copy directory information is maintained in dedicated associative caches (''tag caches'') only for blocks that are present in processor caches. This reduces directory storage overhead over maintaining directory information for all blocks in memory. As an additional optimization, two different caches are used to maintain copy directory information: a large cache with a limited copy directory in each entry, and a small cache with full-map entries. A block is initially allocated an entry with a limited copy directory. If the number of copies exceeds the number of copies supported in the limited copy directory, a full-map entry is allocated, and the limited copy directory entry is freed.

The second scheme in[O'Kr90] requires dedicated associative copy directory caches to exploit the fact that not all blocks are shared at each point in time. Through the use of simple pointers in the PPT, we accomplish the same functionality with simpler hardware that should result in faster operation. Since the pages are large, the number of entries in the PPT is relatively small and the storage overhead for the additional pointers is insignificant (see Subsection 3.2.3). There is no discussion in [O'Kr90] of how the two ideas for reducing the storage requirements for mapping tables can be combined in one efficient scheme, as described earlier in this section. In particular, as discussed in Section 3.2.2, the two-level management scheme allows a new efficient copy directory organization to be used.

Gupta et al. [Gupt90] propose using caches of directory information, which they call *sparse directories*, instead of maintaining a directory entry for every memory block. The sparse directory maintains only entries associated with blocks that are present in the caches. This sparse directory approach is similar to our approach of using inverted mapping tables. However, Gupta et al. [Gupt90] maintain copy directory information for each entry in the sparse directory. They do not explore the benefits of dynamically identifying private and shared data. Our hierarchical scheme identifies dynamically shared data and maintains copy directory information only for shared pages. This refinement leads to additional savings in directory storage overhead.

With both the copy directory organizations discussed in Section 3.2.2, the copy directory is placed at the owner. Since local storage for the copy directory is limited, performance may degrade with such organizations in the presence of heavy sharing in large systems. In the Stanford FLASH multiprocessor [Kusk94] this problem is alleviated using a technique called *dynamic pointer allocation*. In FLASH, copy directory entries for all blocks are allocated from a common bank of copy directory memory. This accommodates the need for large copy directories for heavily shared blocks while not wasting storage for the small copy directories sufficient for lightly shared blocks.

Another solution to the problem of copy directory overflows is to implement the copy directory of a block as a linked list of entries (a *chained directory*) distributed among the processors sharing the block [IEEE93, Gust92, Thap90, Chai90]. The chained directory scales well to larger systems, but incurs additional latency during invalidations in scanning the linked list.

Alewife employs a LimitLESS [Chai91] copy directory organization which handles copy directory overflows in software. Alewife maintains a limited copy directory in hardware. When this limited hardware copy directory overflows, a full-map copy directory is maintained in software. All subsequent requests for this block are serviced by software. Our scheme can also take advantage of such copy directory extensions to improve performance under heavy sharing.