Chapter Five

Evaluation of Hierarchically Managed DSM

The hierarchically managed DSM system optimizes the management of shared data based on the assumption that only a small percentage of the shared address space of a process is actually shared at any time. This optimization can potentially enable the hierarchical scheme to achieve the performance advantage of uniform block-based schemes without incurring their large storage overhead (see Chapter 3). We have evaluated the performance of the hierarchical management scheme using trace-driven and execution-driven simulations. We discuss our performance evaluations and present results from these evaluations in this chapter.

Our initial studies evaluated the performance of the hierarchical scheme using trace-driven simulations. The trace-driven simulator does not implement the complete coherence protocol and executes accesses atomically (Section 4.4 in Chapter 4). By using address traces from real workloads with our trace-driven simulations, we obtained initial validation of our premises about sharing behavior in real applications.

Subsequently, we have evaluated the performance of the hierarchical scheme using execution-driven simulation. The execution-driven simulation includes a complete implementation of the coherence protocol and accurately models the behavior of the processors and the network. Since the execution-driven simulation also <u>executes</u> real applications (rather than using address traces from a prior execution on a different architecture), the results from this simulation more accurately reflect the performance of a real DSM system.

For both evaluation studies, the simulations also implemented the one-level scheme to allow the performance of the hierarchical scheme to be compared with the performance of the one-level scheme. Our simulations also provide the flexibility to study the effectiveness of the hierarchical scheme over a range of block sizes and mapping table sizes. We first describe the evaluation based on execution-driven simulation and its results in Section 5.1. In Section 5.2, we describe the results from our trace-driven simulations.

5.1. Evaluation based on Execution-driven Simulation

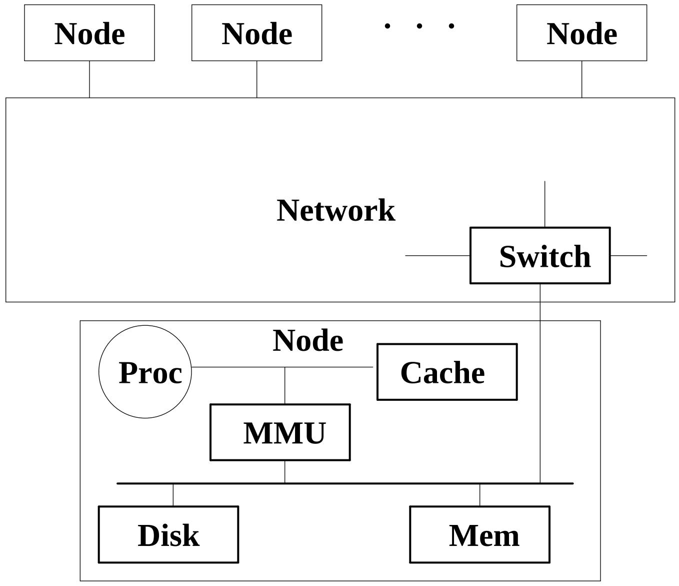

Figure 5.1: Model of the execution-driven simulation. A system with 64 nodes and a 2-D mesh network was simulated.

As described in Section 4.4.2 of Chapter 4, our execution-driven simulation is based on Proteus [Brew92b], which models the system as a number of nodes connected by an interconnection network (Figure 5.1). Each node model in the simulated system is comprised of a processor, cache, memory, disk, a memory management unit (MMU), and an interface to the network. The processor is ‘‘simulated’’ by directly executing the application code on the simulation host. The cache, memory, disk and MMU on each node are simulated by a *memory engine*. The memory engine implements the mechanisms needed to access the shared address space (such as address translation, disk access, coherence protocol) and models their respective latencies. The network interface and the network are simulated by the *network engine*. This engine implements message sending, transmission across the network and reception.

We performed our evaluations for a system with 64 nodes. The behavior of the processor is emulated by executing all application instructions that would be executed by the processor directly on the host machine (Sun Sparc) on which we ran our simulations.

The network is modeled as a packet-switched bidirectional 2-D mesh (8×8). The network switches use infinite output port buffers. In Proteus, the bandwidth of the network links is parameterized as a function of the processor cycle frequency in the processor model. We used the network engine over a range of link bandwidths. For a processor operating at P MHz, we have varied the link bandwidth from P MB/s to $8*P$ MB/s (link bandwidths from 200 MB/s to 1.6 GB/s for a 200 MHz processor, for example). Control messages sent through the network have a size of 5 bytes. The size of data messages is the size of the data plus 5 bytes.

The memory engine models the memory and disk on each node. The local disk controller component of the memory engine controls access to the disk. The disk controller models a latency of 1000 processor cycles to access data from the disk, for all size of data accessed (with current technology, latencies of 1,000,000 cycles can be expected). With the hierarchical scheme, disk accesses are at the granularity of pages. However, with the one-level scheme, disk accesses are at the granularity of blocks. While a real system would access disk only at the granularity of pages, the one-level scheme implemented in our simulations does not include the page-level tables necessary for supporting page-level disk accesses. The implementation of page-level disk accesses with block-level tables is tricky since the eviction of a block to disk from one node necessitates the coordinated eviction of all other blocks of the page from all other nodes. In addition, if page-level disk accesses are supported without page-level tables, a node requesting a block of data may need to allocate an entry for each block of the page being accessed to map all the data received from disk. A practical implementation of the one-level scheme should extend it with page-level tables to support page-level disk accesses (making the scheme pseudo-hierarchical).

The memory engine also implements the memory management unit (MMU) which supports distributed shared memory. The MMU model implements all the mapping tables and our distributed directory coherence protocol for hierarchical management (the MMU also implements the one-level scheme which is described in the next sub-section). The MMU models the presence of a cache on each node in a simple way by assuming that a fixed proportion of shared accesses that hit in the local node's memory hit in the local cache. Our simulations used a cache hit rate of 97% and a cache hit time of one processor cycle. The MMU does not model the Translation Lookaside Buffer (TLB) in detail. However, the average latency of accessing the Present Pages Table (PPT) and the Shared Pages Table

(SPT) are estimated assuming that entries from the PPT and SPT are cached in the TLB. The average PPT access latency is estimated based on a raw PPT access time of 20 processor cycles, a TLB hit access time of one processor cycle and a TLB hit rate of 90% for PPT entries. The average SPT access latency is estimated based on a raw SPT access time of 20 processor cycles, a TLB hit access time of one processor cycle and a TLB hit rate of 85% for SPT entries. Table 5.1 tabulates the latency of accessing various tables in our simulation.

Access Latencies of Coherency Tables			
Table	Acronym	Latency	Comments
Present Pages Table	PPT	3 cycles	cached in TLB from memory
Shared Pages Table	SPT	4 cycles	cached in TLB from memory
Copy Directory	CoD	7 cycles	look-up is pipelined; updated in parallel with message sending
Migrated Pages Table	MPT	40 cycles	cached in memory from disk

Table 5.1: Table Access Latencies.

Our simulations of the hierarchical scheme simulated a page size of 1024 bytes (with varying block sizes) while our one-level scheme simulations simulated several page sizes from 16 bytes to 1024 bytes. Table 5.2 lists the sizes of the PPT, SPT and the copy directory (on each node) in our simulations of the hierarchical and one-level schemes. For the one-level simulations, the number of entries in the PPT, SPT and copy directory are tabulated for each page size that was simulated. At small page sizes, the number of entries in the PPT and SPT are higher since each entry maps a smaller fraction of the local memory. The total amount of memory mapped by the PPT and SPT on each node (product of the number of entries in the table and the page size) are also shown in Table 5.2 (columns labeled

‘‘Size of Memory Mapped’’) for each page size. As the table shows, the amount of memory available in each node (the size of memory mapped) is not the same for all page sizes simulated. At small page sizes, the amount of memory available is smaller in our simulations. This is because the large number of PPT and SPT entries required to map larger memory at small page sizes increases the memory requirements of the simulator posing a burden for our simulation host machine. In all our simulations, we used a full copy directory with 64 entries (last column of the table). The Migrated Page Table (MPT) was allowed to grow as large as necessary to maintain an entry for each page of shared data on disk.

Scheme	Page Size (bytes)	Present Pages Table		Shared Pages Table		Number of entries in Copy Directory
		Number of entries	Size of Memory Mapped	Number of entries	Size of Memory Mapped	
Hierarchical	1024	512	512 KB	128	128 KB	64
Onelevel	16	8192	128 KB	2048	32 KB	64
	32	4096	128 KB	1024	32 KB	64
	64	2048	128 KB	512	32 KB	64
	128	2048	256 KB	512	64 KB	64
	256	1024	256 KB	256	64 KB	64
	512	1024	512 KB	256	128 KB	64
	1024	512	512 KB	128	128 KB	64

Table 5.2: PPT, SPT and Copy directory sizes in simulations. Our hierarchical scheme simulations used a page size of 1024 bytes. The page size was varied from 16 bytes to 1024 bytes in our one-level scheme simulations.

Proteus measures the execution time of the application on the simulated architecture as the sum of the time spent executing local instructions and the time spent executing shared accesses. As described in Section 4.4.2 in Chapter 4, to measure the time spent executing local instructions, the Proteus compilation process augments the assembly code of the application to count the number of instructions executed by the application. The Proteus compilation process also converts all accesses to shared data in the application to calls to the memory engine which services these accesses. The time spent executing shared accesses is the latency within simulator engine modules such as memory and network. During simulation, the cycles spent in the execution of application basic blocks and the cycles used up in simulator engine modules are both measured. In addition, the simulator engine modules are instrumented to measure several behavior and performance characteristics of the application and the system.

We describe our implementation of the one-level scheme and the characteristics of our benchmark applications in the following two sub-sections before presenting performance evaluation results from our simulations.

5.1.1. One-level Scheme Implementation

Our Proteus memory engine also implements the one-level scheme to allow the performance of the hierarchical and one-level schemes to be compared. The one-level scheme manages memory at a single granularity which can be as large as a page (Section 3.1.1), or as small as a cache block (Section 3.1.2). We evaluate page-level and block-level schemes by varying the size of the block in the model. In the rest of this subsection, we describe the key differences between the one-level scheme implementation and the implementation of the hierarchical scheme.

In the one-level scheme data is mapped and transferred at only one granularity (we refer to it as a *block*, whose size is variable). Hence, the coherence protocol

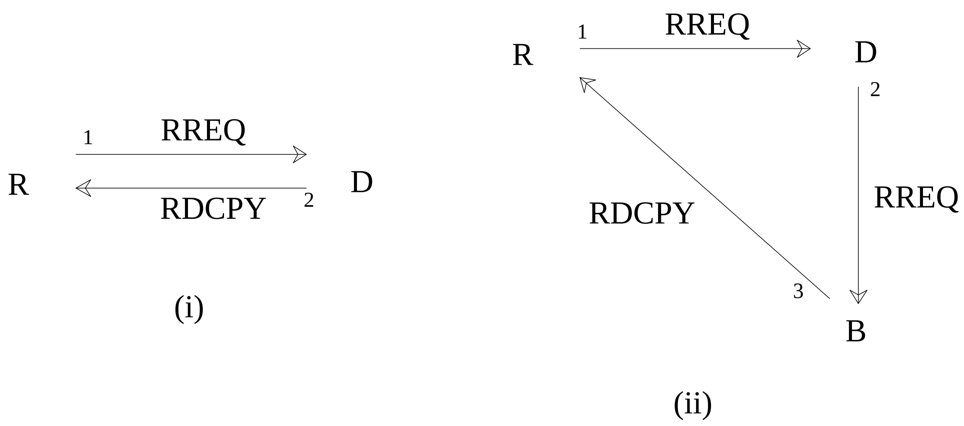

Figure 5.2: One-level scheme protocol flow sequences for read requests. *R* is the read requester, *D* is the default owner, and *B* is the block owner. Protocol can flow as in case (i) when the default owner is also the block owner, and as in case (ii) otherwise.

used for this implementation allows only one level of ownership. Every block of data in physical memory is owned only at the block level by a node referred to as the block owner. A statically assigned default owner for the block maintains the identity of the block owner. The default owner in the one-level scheme serves the function of both the default owner and the page owner in the hierarchical scheme.

The protocol transactions for read and write requests differ from the transactions in the hierarchical scheme primarily in the absence of a second-level page owner. When a node has a read or write miss, the request is sent to the default owner. The default owner identifies the current block owner using its MBT (Migrated Block Table). Read requests are forwarded by the default owner to the current block owner, who directly sends a reply to the requester. In case of a write request, the MBT entry is locked at the default owner. The default owner sends a message to the current block owner requesting it to write the block back. Upon receiving the request, the block owner, as in the hierarchical scheme, invalidates all

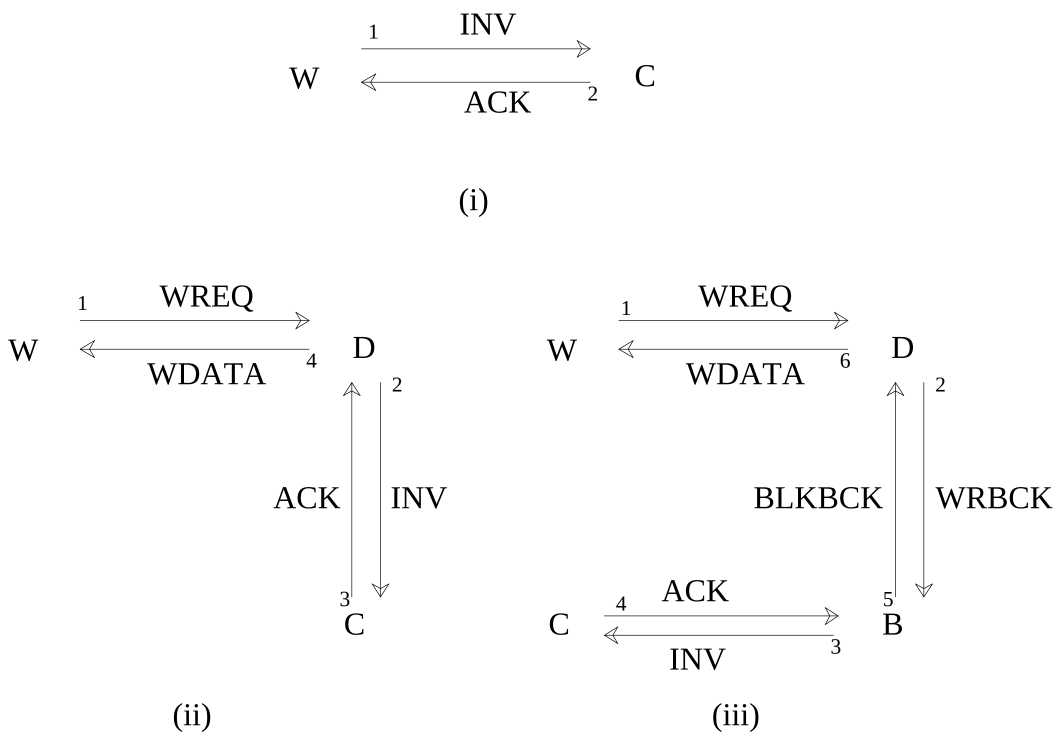

Figure 5.3: One-level scheme protocol flow sequences for write requests. *W* is the write requester, *D* is the default owner, *B* is the block owner, and *C* is some node that has a read copy of the block. Protocol can flow as in case (i) when the *W* is the block owner. Case (ii) will occur when the default owner is also the block owner. Case (iii) also occurs when the block is owned by a node other than the default owner.

read copies of the block. Once all copies are invalidated, the block is written back to the default owner. Upon receiving the write-back, the default owner updates its MBT and the block data and ownership are sent to the requester. We illustrate the typical protocol flows for reads and writes in Figure 5.2 and Figure 5.3 respectively.

The protocol message types and protocol states in the implementation are modified slightly to take advantage of some simplifications allowed by the one-level scheme. Since only one size of data is transferred, messages are not needed to support both blocks and pages. Page-related messages such as *PGOWN*, *PGINV*, *PAGE* and *PGBCK* are not used in the one-level scheme implementation. A block that is absent in a node has no entry associated with it on that node in the one-level scheme (in the hierarchical scheme, an entry is present if some other block of the page is present). Hence, the block access_permit *ABS* is not used. The absence of an entry in the Present Block Table (PBT) provides the same information.

5.1.2. Application Benchmarks

Our execution-driven simulation results are based on the execution of three scientific applications from the SPLASH-2 application suite — Cholesky, LU and Radix [Sing92, Woo95]. These SPLASH-2 applications were ported to use the Proteus API for shared memory programming (Section 4.4.2 in Chapter 4) by the author of [Kopp95]. The port modified the application source code to explicitly identify accesses to the shared address space. Also, synchronization points in the applications were modified to use the test-test-and-set semaphore locks (which are implemented through reads and writes to the shared address space) and barriers (which are implemented through messages) provided by Proteus.

The Cholesky application performs parallel Cholesky factorization of a sparse positive definite matrix. A sparse matrix is factored into the product of a lower triangular matrix and its transpose. The program primarily performs the numerical factorization step of Cholesky factorization [Sing92], which determines the actual numerical values of the non-zero entries in the lower triangular matrix. The numerical factorization is efficient due to the use of supernodal elimination techniques. In our simulations, the application processed the tk14.O input data set provided in the SPLASH-2 suite.

The LU kernel [Woo95] factors a dense matrix into the product of a lower triangular and an upper triangular matrix. The dense 256×256 matrix is divided into a 32×32 array of 8×8 blocks to exploit temporal locality on sub-matrix elements. Block ownership is assigned using a 2-D scatter decomposition, with blocks being updated by the processors that own them. The kernel version is a less-optimized version that does not allocate elements of a block contiguously.

The Radix kernel [Woo95] implements an integer radix sort based on the method described in [Blel91]. The algorithm is iterative, performing one iteration for each radix *r* digit of the keys. In each iteration, a processor passes over its assigned keys and generates a local histogram. The local histograms are then accumulated into a global histogram. Finally, each processor uses the global histogram to permute its keys into a new array for the next iteration. This permutation step requires all-to-all communication. The permutation is inherently a sender-determined one, so keys are communicated through writes rather than reads. The application sorted 256 K keys with maximum key value 512 K using a radix of 1024 in our evaluations.

Table 5.3 summarizes the problem size solved by the applications and the amount of disk storage available to the application. The second column of the table specifies the size of the problem solved by the application. All shared data used by the application in solving this problem is mapped to the disks in the system (one per node). Pages of shared data are mapped across the disks in the system with a round-robin policy. During execution, the application must access the data

Application	Problem Size	Disk storage per node in simulation	Shared Data used by application
Cholesky	tk14.O	1 Mbyte	18.02 Mbytes
LU	256×256 matrix, 8×8 blocks	256 Kbytes	719 Kbytes
Radix	256 K keys, 512 K maximum key value, radix 1024	256 Kbytes	3.08 Mbytes

Table 5.3: Memory usage in application benchmarks. The third column presents the amount of storage available in the disk of each node in the simulation. The simulated system had 64 nodes. The last column lists the actual amount of memory used by the application for storing its shared data.

from the disk maintaining that page. The third column of the table lists the amount of disk storage provided in each node in the simulation. The last column of the table shows the total amount of shared data that was accessed by the application during the simulation.

Table 5.4 characterizes the applications by the number of instructions executed, the total number of read accesses to shared data and the total number of write accesses to shared data. The table lists these measures for simulations with 64-byte and 256-byte blocks, both for the one-level scheme and for the hierarchical scheme. LU and Radix are roughly equal in size in the number of instructions executed (about 400 million). Cholesky is a larger application executing four times as many instructions. As the table shows, the number of instructions executed can be different in simulation runs with different block sizes. Since the latency of shared accesses is influenced by the block size, processes can arrive at synchronization barriers and locks in different sequence in simulations with

Application	Scheme	Block Size (bytes)	Number of Instructions (x 10^6)	Number of Shared Reads (x 10^6)	Number of Shared Writes (x 10^6)
Cholesky	Onelevel	64	1433.47	69.23	7.74
		256	1401.09	67.83	7.74
	Hierarchical	64	1390.37	67.43	7.74
		256	1362.38	65.75	7.74
LU	Onelevel	64	460.12	12.09	5.72
		256	469.58	12.09	5.72
	Hierarchical	64	458.01	12.09	5.72
		256	466.58	12.09	5.72
Radix	Onelevel	64	417.29	11.86	6.51
		256	421.06	11.96	6.51
	Hierarchical	64	416.75	11.86	6.51
		256	420.64	11.95	6.51

Table 5.4: Number of instructions and shared accesses in the applications.

different block sizes. As a result, the number of instructions executed in checking whether all processes have reached the barrier or in suspending and waking up processes may be different in simulations with different block sizes. With Cholesky, such differences in the number of instructions executed can also occur due to the additional synchronization structures (e.g., to signal that a task is ready for processing) that are implemented in the application. The number of instructions executed spinning on these structures (waiting for a task to become ready, for example) can be different in simulations with different block sizes. In all three applications, shared accesses constitute a small fraction of the application's instructions. The percentage of instructions which are shared accesses is only 5% in Cholesky, 3.7% in LU and 4.2% in Radix.

Application	Scheme	Block Size (bytes)	Number of Instructions in Parallel Phase (x 10^6)	Number of Shared Reads in Parallel Phase (x 10^6)	Number of Shared Writes in Parallel Phase (x 10^6)
Cholesky	Onelevel	64	1227.63	65.23	2.72
		256	1203.06	63.83	2.72
	Hierarchical	64	1193.49	63.43	2.72
		256	1165.50	61.75	2.72
LU	Onelevel	64	451.91	11.96	5.59
		256	461.66	11.96	5.59
	Hierarchical	64	450.18	11.96	5.59
		256	458.75	11.96	5.59
Radix	Onelevel	64	417.03	11.33	6.51
		256	420.86	11.43	6.51
	Hierarchical	64	416.53	11.34	6.51
		256	420.44	11.42	6.51

Table 5.5: Number of instructions and shared accesses in the parallel phase of the applications. Applications have an initial serial phase when a single process executes initialization code and spawns other processes. Most of the instructions and shared accesses are in the parallel phase of the application.

The execution of all three applications have an initial serial phase before the parallel phase of execution. In the serial phase, a single process executes initialization code and then spawns all other processes for the parallel phase. Table 5.5 lists the number of instructions executed in the parallel phase of the execution and the number of read and write accesses to shared address space in the parallel phase. The instruction count and access count measures for the parallel phase are the sum of the instruction counts and the access counts, respectively, of all the processes that executed during the parallel phase. The table presents measurements from simulation runs with 64-byte blocks and 256-byte blocks for the one-level and the hierarchical schemes. Comparing with the numbers tabulated for the overall execution in Table 5.4, it can be seen that all three applications execute most of their instructions in their parallel phase. Cholesky executes 86% of its instructions in the parallel phase, while LU and Radix execute nearly all their instructions in the parallel phase. While roughly 94% of Cholesky's read accesses to shared data are in its parallel phase, only 35% of its write accesses to shared data are in the parallel phase. The majority of Cholesky's shared write accesses occur in the initialization. 99% of LU's shared reads and 98% of its shared writes occur in its parallel phase. In Radix, the array to be sorted is initialized only in the parallel phase. Hence, all its shared writes are in the parallel phase.

To study the distribution of the workload amongst the processors, we have measured the number of instructions, the number of shared reads and the number of shared writes executed by each processor in the parallel phase. The histograms in Figure 5.4, Figure 5.5 and Figure 5.6 show the breakdown of instructions, shared reads and shared writes, respectively, amongst the processors (for the parallel phase). In each figure, the breakdown is shown for all three applications on the one-level managed and hierarchically managed DSM systems with 64 byte blocks.

Figure 5.4 shows that the total number of instructions executed is very well distributed among the individual processors in the Cholesky application (mean of 19.17×10^6 cycles and a standard deviation of 0.37×10^6 cycles). The breakdown of instructions is less balanced in the LU kernel (mean of 6.79×10^6 cycles and a standard deviation of 0.91×10^6 cycles). The breakdown of instructions is least

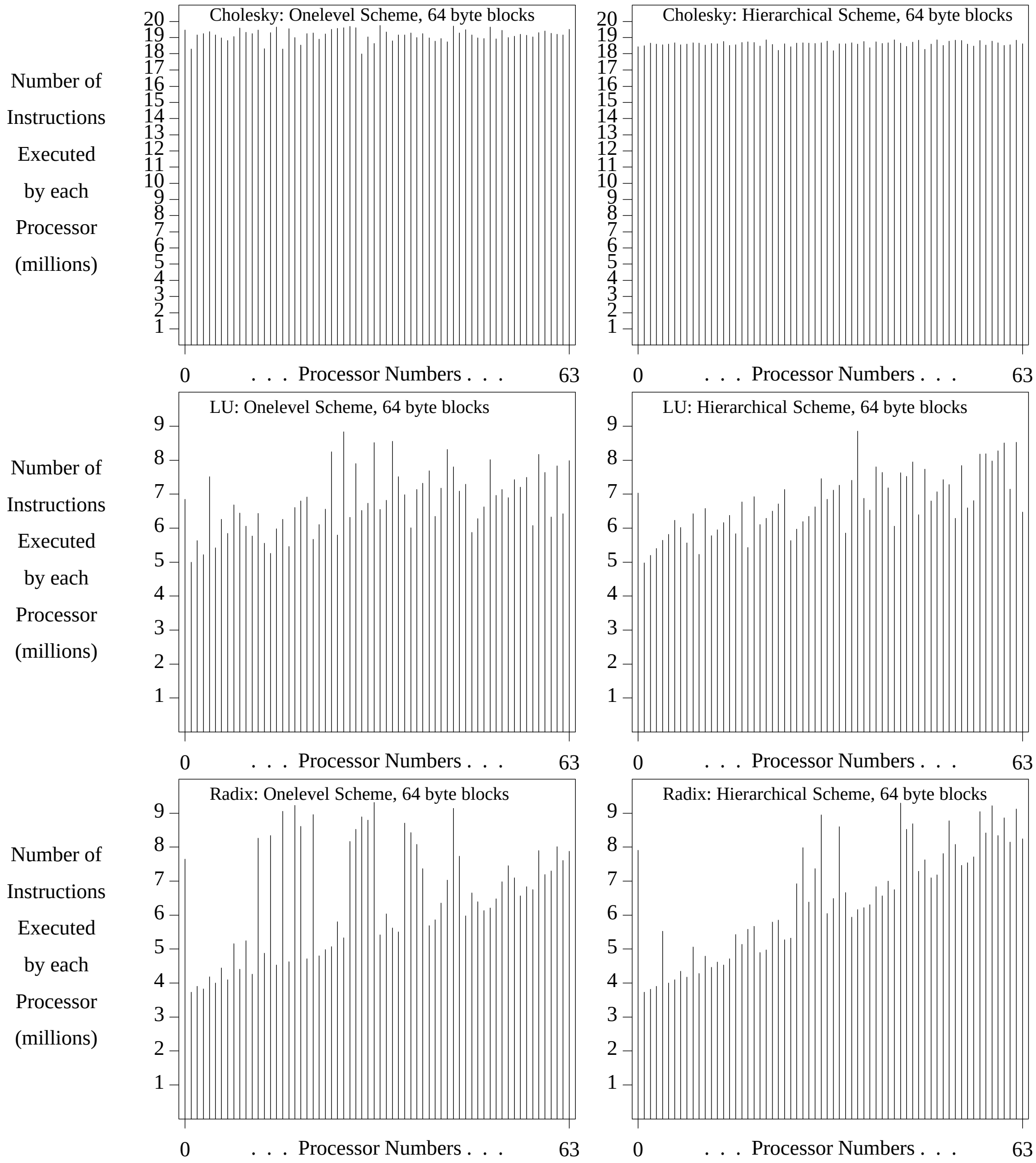

Figure 5.4: Distribution of application instructions amongst processors.

uniform in the Radix kernel (mean of 6.54 $\times 10^6$ cycles and a standard deviation of 1.65 $\times 10^6$ cycles). The figure also shows a clear pattern in the distribution of instructions among the processors in Radix. In the Radix application kernel, all

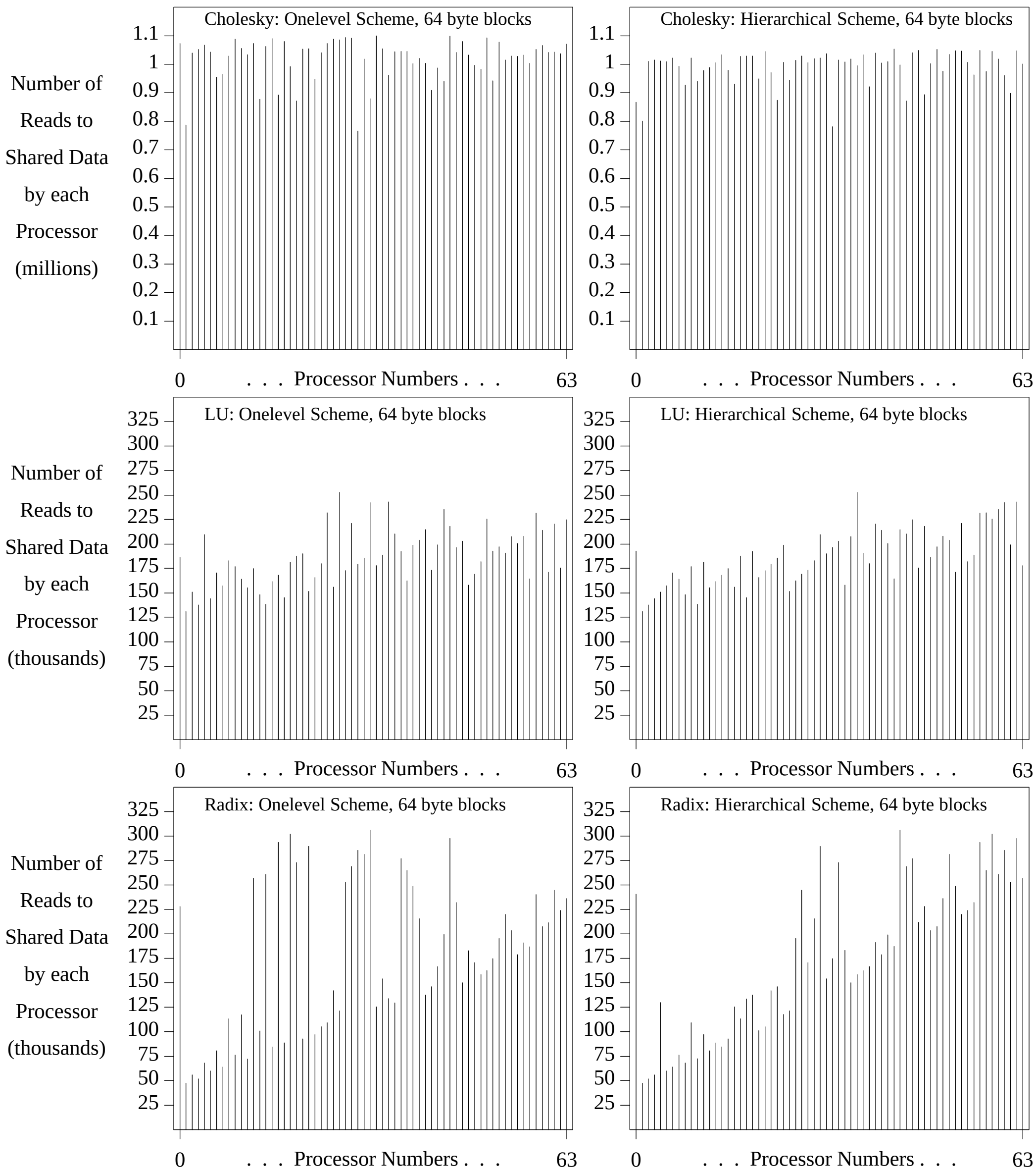

Figure 5.5: Distribution of application shared reads amongst processors.

processes are ranked in a fixed order. In every iteration of the sort, each process determines the location of its keys by reading the local histograms generated by each process with a lower rank. This phase results in a non-uniform distribution of

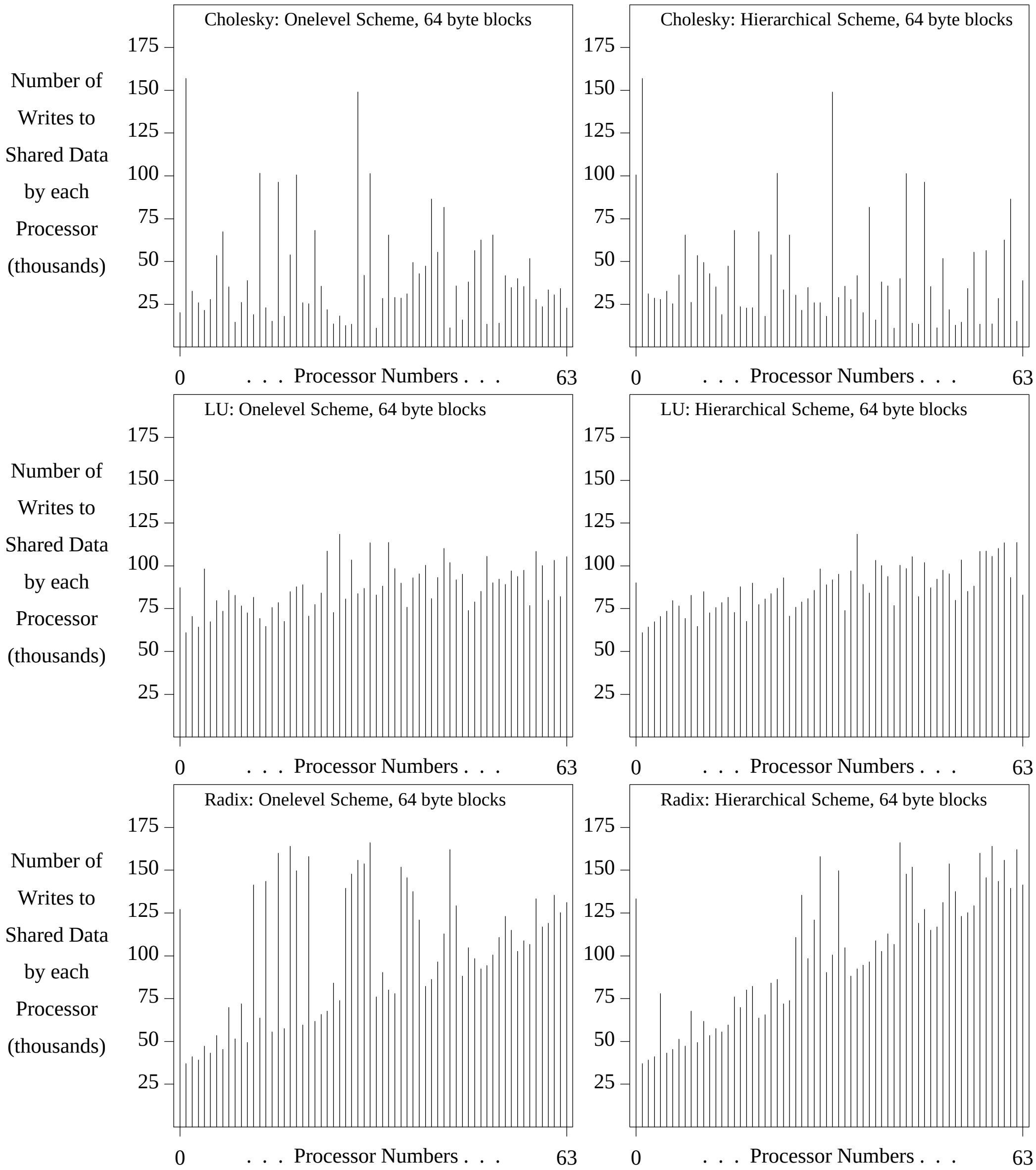

Figure 5.6: Distribution of application shared writes amongst processors.

workload, where the workload of a process is proportional to its rank in the order. The figure shows the workload changing nearly linearly among the processors (it is not exactly linear because the rank number of a process and the number of the processor where it executes are different).

Figure 5.5 and Figure 5.6 show the breakdown of read and write accesses to shared address space amongst the processors. The distribution of shared read and write accesses among the processors in LU and Radix is very similar to the distribution of instructions among the processors in these applications. In the Cholesky application, accesses to shared data, especially write accesses, are less evenly distributed among the processors than instructions.

Most of our evaluation has studied the behavior of the system and application in the parallel phase of execution. In the discussion of our simulation results in the following sections, we present our results for the parallel phase of application execution.

5.1.3. Measurement of the Shared Working Set Size

In many applications with a large shared data set, only a small fraction of the references may be to shared data [Dare87]. In addition, even when shared data is accessed by a process, the shared pages in the working set of the process may be dynamically private to the process [Pink92]. As a result, only a small fraction of the shared data set may actually be shared by any process at any time. Based on this observation, the hierarchical scheme allows only a small fraction of data in the physical memory of any node to be shared at any time. The Shared Pages Table (SPT) which manages the dynamically shared component of the application's working set has fewer entries than the Present Pages Table (PPT). Hence, only a small fraction of the data mapped to a node's physical memory can be dynamically shared with other nodes. If the number of SPT entries is insufficient to maintain the shared working set of the application, frequent SPT entry thrashing can lead to performance degradation.

To evaluate the validity of our premise about parallel application behavior, we have studied application performance at different SPT sizes. We studied the applications on a system simulating the one-level scheme with small 16-byte pages. The small size of the page minimizes false sharing, allowing us to study the effect of true-sharing in the application. We simulated the one-level system for this study since the number of SPT entries is a better measure of the memory available for the application's shared working set with the one-level scheme than with the hierarchical scheme. While SPT entries in the hierarchical scheme allow data to be shared in units smaller than a page (block-level sharing), each SPT entry corresponds to a page of data. Hence, in the hierarchical scheme, when an SPT entry is used by a block of a given page, any unused space in the page frame is not available for use by blocks of other pages.

We ran several simulations of the execution of each application on a one-level system with 16-byte pages. The number of entries in the Shared Pages Table (SPT) was varied in these simulation runs. In all runs, 128 Kbytes of physical memory and 8 K Present Pages Table entries were available on each node. Figure 5.7 presents the results from these simulations.

Figure 5.7 plots the execution time of each application as a function of the number of entries in the SPT. Application performance is lower (longer execution time) at smaller SPT sizes due to higher contention for SPT entries. However, for all three applications, small SPTs with only 256 entries can support application performance within 20% (within 3% in the case of LU and Radix) of the

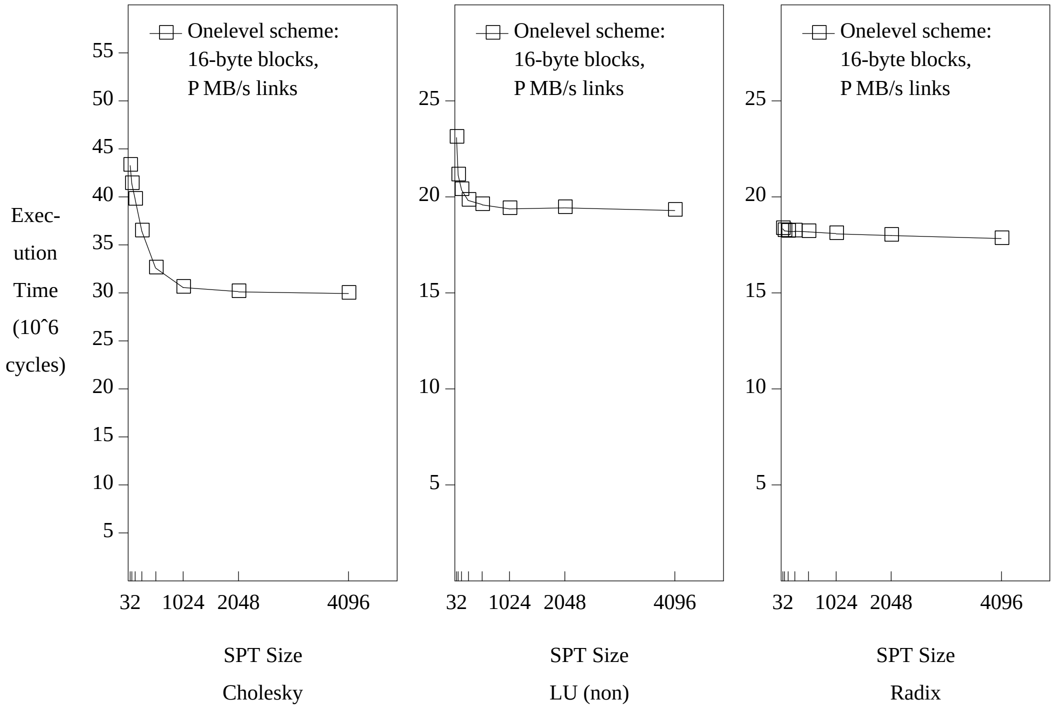

Figure 5.7: Small Shared Working Set. The application's execution time is plotted as a function of the number of entries in the Shared Pages Table (SPT), for each of the applications executed in our simulation. The scheme simulated is the one-level scheme with 16 byte blocks. The number of entries in the Present Pages Table was 8 K. The applications perform well even when the amount of shared data mapped by the SPT is a small fraction of of the total amount of shared data used by the application (listed in Table 5.3).

performance with 4 K SPT entries. This shows that most of the data that is actually shared at any time can be managed with 256 SPT entries on each node. The amount of shared data mapped by 256 SPT entries is 4 Kbytes (16-byte blocks), which is a small fraction of the total shared data used by the applications (see Table5.3). Thus, Figure 5.7 verifies that, due to application sharing behavior, for many applications there is a relatively small performance penalty for using an SPT which is much smaller than the PPT.

5.1.4. Measurement of the Effectiveness of the Copy Directory Organization

To support block-level sharing, the hierarchical scheme maintains copy directory information for each block of every shared page. In Section 3.2.2 of Chapter 3, we discussed two possible ways to organize the copy directory. The straightforward organization, called the Processor List per Block or PLpB organization, maintains an independent list of processors for each block of a shared page. We proposed an alternative organization, called the Block Map per Processor or BMpP organization (Section 3.2.2) for the hierarchical scheme. In the BMpP organization, the copy directory information associated with all blocks of a page at a node are unified in a compact copy directory common to all blocks of the page on that node. The BMpP copy directory maintains the list of processors that share any block of the page and maintains a bit map for each processor identifying the blocks held by that processor. In this sub-section, we discuss our results comparing the effectiveness of these two copy directory organizations. We also discuss the feasibility of bounding the number of entries in the BMpP copy directory.

In the hierarchical scheme, a node that owns any block of the page (block owner) maintains in its copy directory for the page the identities of nodes having read-only copies of any block of the page that it owns locally. The page owner maintains the same information if it owns any blocks of the page, and also maintains the identities of all nodes that own any blocks of the page in its copy directory. The simulation measures the number of nodes in the copy directory when the copy directory is maintained independently for each block (PLpB organization) and when a common copy directory is maintained for all blocks of the page (BMpP organization). These numbers are measured independently for the copy directory in block owners and for the copy directory in the page owner.

During execution, a copy directory is initialized for a page on a node when a block of the page is first shared from that node by another node. When other nodes stop sharing these blocks, the copy directory is deallocated. Within the lifetime (time between initialization and deallocation) of each copy directory, the number of nodes in the copy directory varies with time. During the lifetime of each copy directory, the simulator tracks the changes in the number of nodes in the copy directory and the time interval between these changes. For the lifetime of each copy directory that is used, the simulator computes the time-weighted average number of nodes in the copy directory if the copy directory is organized as a PLpB copy directory and if it is organized as a BMpP copy directory. The simulator also measures the lifetime of each copy directory. At the end of execution, the average number of nodes in a copy directory is computed for each copy directory organization. This average is computed by averaging the numbers measured for each copy directory that was used during the simulation over all copy directories that were used weighted by the lifetime of each copy directory. The average is computed independently for copy directories in block owners and for copy directories in page owners.

The effectiveness of the PLpB and BMpP copy directory organizations can be compared by comparing the storage required with each organization to maintain the average number of copy directory entries measured during simulation. The size of the copy directory needed to accommodate the entries can be easily calculated for both copy directory organizations. The size of an entry in the PLpB organization is simply the size of a node identifier (6 bits for a 64-node system). The size of the copy directory of a block in the PLpB organization is the product of the size of the entry and the average number of entries measured by the simulation. Since a copy directory must be maintained for each block of a page in the PLpB organization, the total size of PLpB copy directories for a page is the product of the size of the block copy directory and the number of blocks per page. The size of an entry in the BMpP organization is the size of the node identifier plus the number of blocks in a page (to store the bit map). The total size of the BMpP copy directory for any page is the product of the size of an entry and the measured number of entries per copy directory.

We performed this simulation and analysis of copy directory organization effectiveness with different block sizes. We simulated a 64-node hierarchical DSM system using 1 KB pages. The results of this analysis are shown in Figure 5.8. The figure shows the computed average size of the copy directory for a page for both copy directory organizations (independently for the block owner and the page owner) as a function of the block size. The BMpP organization of the copy directory requires less storage than with the PLpB organization, especially with small blocks. With both organizations, the storage required for the copy directory is higher at smaller block sizes. At small block sizes, false-sharing is lower, which can lead to fewer entries in the copy directory. However, smaller blocks result in a larger number of blocks per page, which can lead to higher storage overhead for the copy directories. Specifically, more blocks require more copy directory entries with the the PLpB organization (one per block) and require larger block maps with the BMpP organization (one bit per block).

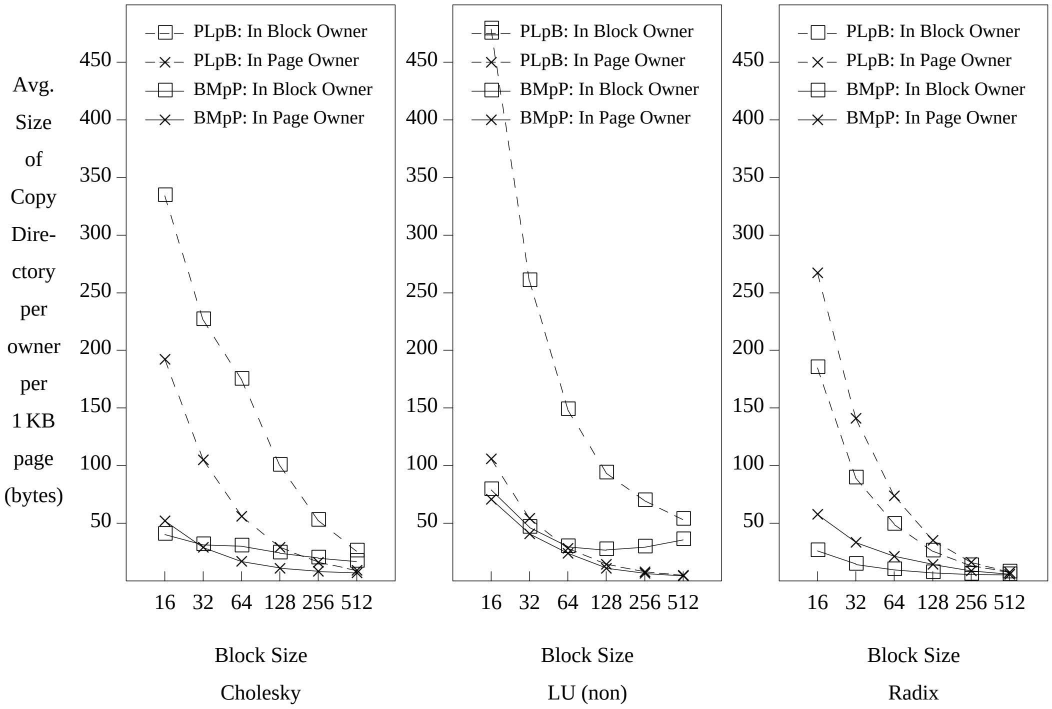

Figure 5.8: BMpP organization of the copy directory is more efficient. The simulation measured the average number of nodes in the copy directory, when it is maintained as an independent processor list per block (PLpB), and when it is maintained for the entire page as a block map for each processor(BMpP). The size of the copy directory needed to accommodate these number of nodes can be computed, and is plotted here for both organizations. The BMpP copy directory can be smaller in maintaining the same information.

Figure 5.9 shows the average number of nodes in the BMpP copy directory as a function of block size. The average is shown independently for block owners and page owners. At larger block sizes, the number of nodes having read copies of a block is higher due to false sharing, requiring more entries in the copy directory at the block owner (marginal in the Radix application). However, the number of entries in the page owner's copy directory decreases with increasing block size. As discussed earlier, this is because for a fixed page size (1 KB in the simulations) the

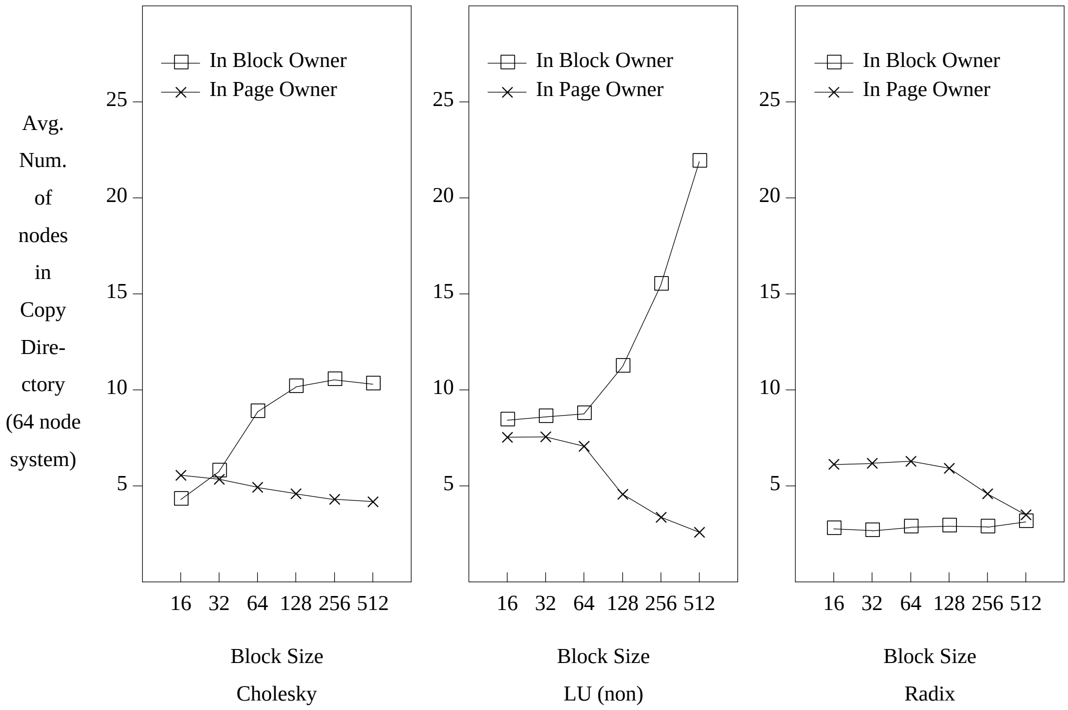

Figure 5.9: Copy Directory can be bounded. The graph plots the number of nodes in the block owner and page owner copy directories in a 64-node system implementing the hierarchical scheme with 1 Kbyte pages. The number of nodes in the copy directory is small at small block sizes, making it reasonable to bound the number of entries in the copy directory.

number of blocks per page (and, hence, the number of potential block owners) decreases with increasing block size. At small block sizes, the average required size of the copy directory is small (≈ 5 nodes in a 64-node system) in both the page owner and the block owners. Since the small average indicates that only a few nodes occupied the directory during a large proportion of the lifetime of the copy directory, it may be reasonable to bound the size of the BMpP copy directory.

5.1.5. Performance Comparison Between the Hierarchical and One-Level Schemes

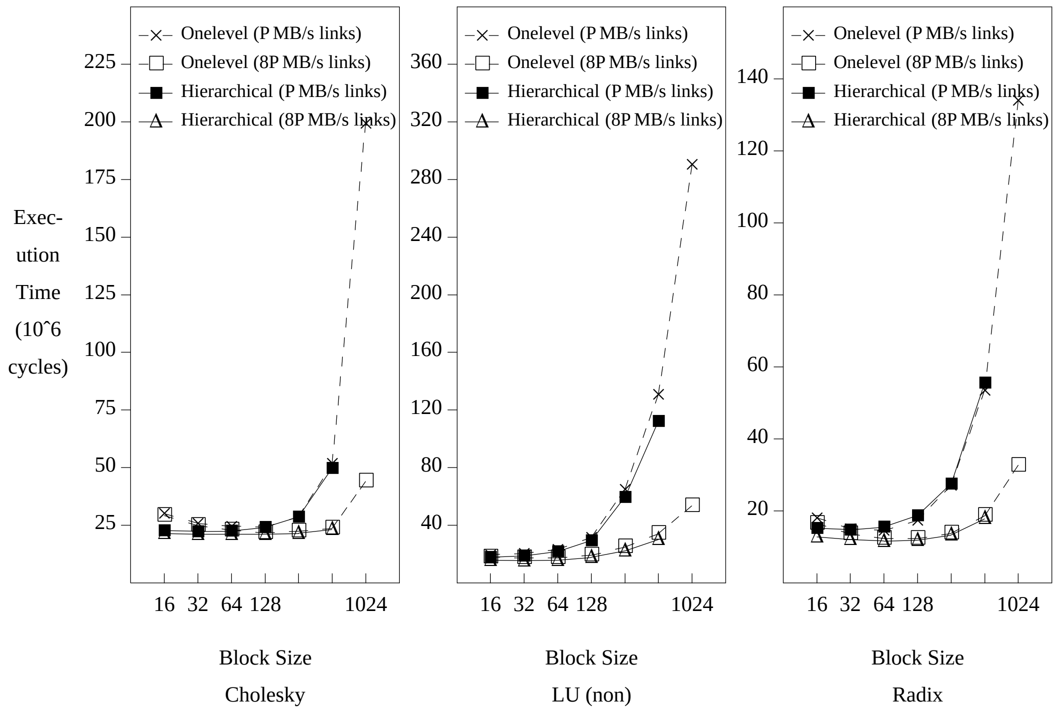

Figure 5.10: Performance of the hierarchical scheme. The graph compares the application execution time with the hierarchical scheme using 1 KB pages, and with the one-level scheme, at various block sizes. Performance is poor with the page-sized one-level scheme (one-level scheme with block size 1024 bytes). The one-level scheme performs well at small block sizes, and the hierarchical scheme matches its performance. When the network is faster, all schemes improve and the qualitative comparison remains the same.

We compare the overall performance of the hierarchical scheme and the one-level scheme in Figure 5.10. For each scheme, the figure shows the execution time of each application as a function of the block size. As expected, when blocks are the size of a memory page (1 KB), the one-level scheme has very poor performance. This is primarily due to the high false sharing and high network latency at large block sizes. Also, the one-level scheme performs very well at small block sizes. The graph also shows that the hierarchical scheme (page size is 1 KB) achieves nearly identical performance to the one-level scheme even for small blocks. As we have seen earlier, the hierarchical scheme incurs a small fraction of the storage overhead incurred by the one-level scheme in supporting this high performance.

We evaluated the hierarchical and one-level schemes at various network link bandwidths. Figure 5.10 shows the comparison for two different networks, one of which has a higher link bandwidth (8P MB/s versus P MB/s link bandwidth) than the other. When the network link bandwidth is higher, the average latency through the network is lower. This leads to overall improvement in application performance with both schemes. However, the qualitative comparison of the schemes remains the same.

Both the hierarchical scheme and the page-sized one-level scheme map data at large page sizes. However, it can be seen from Figure 5.10 that the hierarchical scheme supports higher application performance than the page-sized one-level scheme. This can be attributed to the support for fine-grain block-level sharing in the hierarchical scheme. The smaller block size leads to lower false-sharing, lower network latency and lower network contention. We address the influence of small block sizes in the rest of this sub-section.

When the size of a block is the smallest addressable unit, the fraction of a page shared from a node reflects the true sharing in that page. When the block size is larger than this minimal word size, a larger portion of the page may be shared from the node than required by true sharing due to the larger sharing granularity.

The degree of false sharing is indicated by the increase in the fraction of the page that is shared. To evaluate this false-sharing, we have studied the impact of block size on the average fraction of a fixed sized page that is shared. Since the Block Map per Processor (BMpP) copy directory organization maintains information about the sharing of each page, we make use of its information in this study.

A specific block of a page is actively shared only if the BMpP copy directory of the page identifies some node for that block. The BMpP copy directory at the page owner identifies nodes that have a read or an ownership copy of any block of the page. Hence, the fraction of blocks that have any nodes indicated in the page owner's copy directory identifies the fraction of the page that is shared by any node. The copy directory at the block owner identifies nodes that hold a read copy of blocks owned by the block owner. Hence, the fraction of blocks that have any nodes indicated in the block owner's copy directory identifies the fraction of the page that is read-shared from that block owner.

When a block of a page is first shared from a node during execution, an SPT entry must be allocated for the page on that node. When the page is no longer shared from that node, the SPT entry is deallocated. During the lifetime of the shared page on that node (or lifetime of the SPT entry), different number of blocks of the page may be shared by other nodes. Our simulator measures, for each page shared in the execution, the average fraction of blocks of the page that is indicated as shared in its BMpP copy directory as well as the lifetime of the page. These measures are collected independently for the copy directory at the block owners and at the page owner. The overall average of the fraction of blocks of a shared page that are shared is computed (independently for page owners and for block

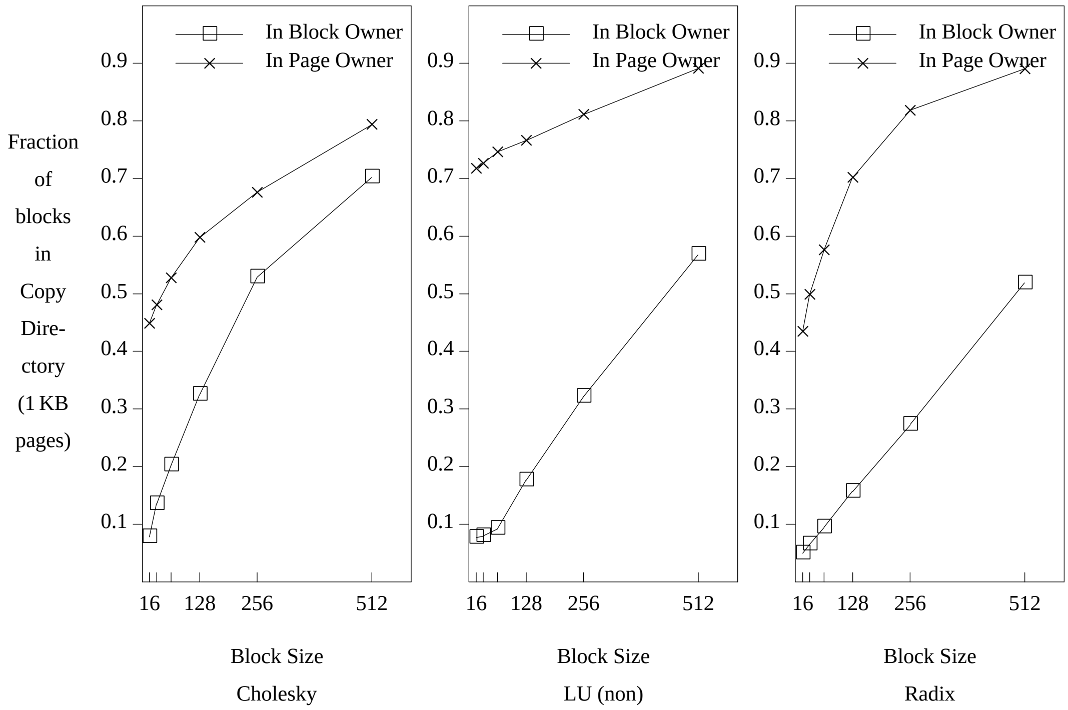

Figure 5.11: False sharing is higher at larger block sizes. A larger fraction of the blocks of the page are shared when the block size is larger (for a fixed page size). The page size used for these simulations is 1 KB.

owners) by averaging all the individual page measures, weighted by the lifetime of each page. For the hierarchical scheme with 1 KB pages, Figure 5.11 shows this computed fraction of blocks of a page (for page owner and the block owner) as a function of the block size.

The fraction of blocks indicated as shared in the block owner copy directory is smaller than the fraction of blocks indicated as shared in the page owner copy directory in Figure 5.11. This is expected since the page owner maintains entries for all blocks that are not locally owned even if they are locally invalid, while the block owner only maintains entries for locally owned blocks that are read shared.

The graphs also show that at small block sizes, under 10% of the blocks are actively read-shared from any block owner. At large block sizes, each block owner manages (owns) a larger fraction of the page and a node that read-shares a block shares a larger fraction of the page. Hence, each block owner shares a higher fraction of blocks at larger block sizes.

Figure 5.11 also shows that the fraction of blocks that are shared in the page owner copy directory is smaller at small block sizes, especially in Cholesky and Radix. This indicates that, in Cholesky and Radix, at larger block sizes, additional data is unnecessarily shared from the page owner by other nodes due to false sharing. In the case of LU, the fraction of blocks that are shared from the page owner is quite high even at small block sizes. The LU application does not allocate the application data accessed by a single process contiguously (see Section 5.1.2). As a result, data belonging to multiple processors can be mapped to the same page. Since ownership of the page is assigned to the node that accesses the page from disk (see Section 4.3.1 in Chapter 4), other nodes which subsequently access the page share the page from the page owner. Hence, the high fraction of blocks shared from the page owner in LU may not be indicative of the true sharing in LU. It is indicative of the high degree of page-level false sharing.

To demonstrate the impact of block size on network latency, we study the average latency of reads that require network communication. Read accesses may need to be retried when they compete against an ongoing write. To ignore the latency contribution due to such access contention, Figure 5.12 shows the average latency of read accesses requiring network communication which do not involve a retry. The read latency is plotted for the one-level and the hierarchical schemes for

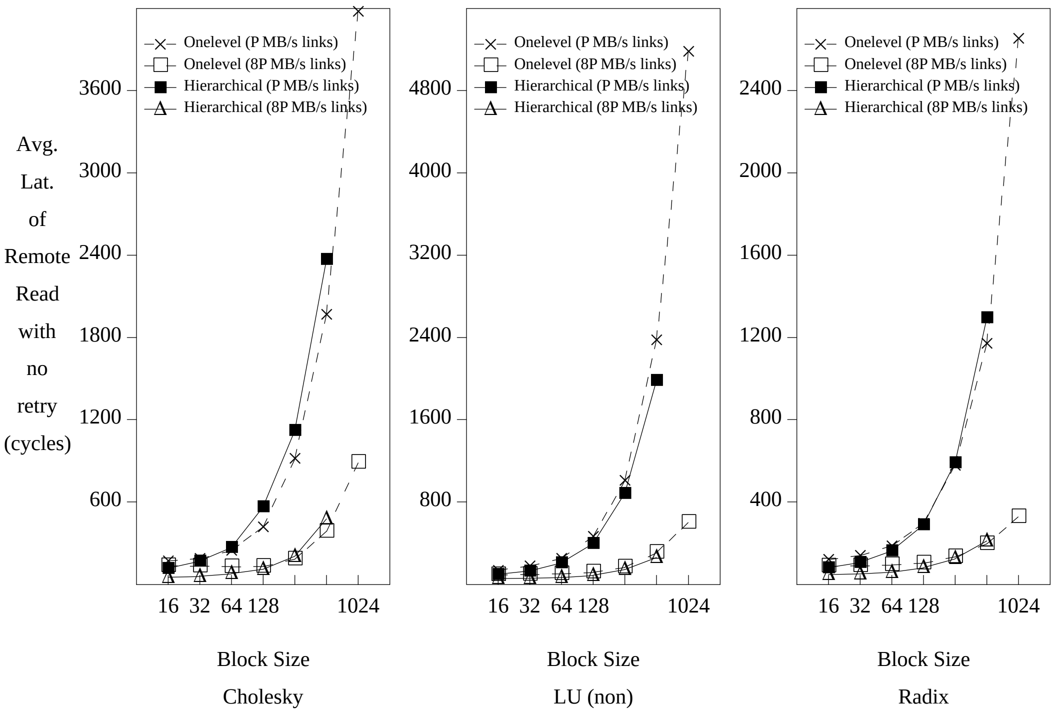

Figure 5.12: Latency of remote read accesses is higher at larger block sizes.

two different network speeds. The response time for read accesses is much higher at large block sizes in all cases. The response time improves significantly in the faster network.

When a processor access cannot be completed locally, the processor is blocked until the access can be satisfied. As we have seen above, non-local accesses take longer to complete at larger block sizes. Hence, at larger block sizes, the processor remains blocked on remote accesses longer. This lowers the processor utilization — the fraction of time a processor is not blocked for remote accesses or synchronization. Figure 5.13 shows the processor utilization as a function of block size for all three applications. In all three applications, the

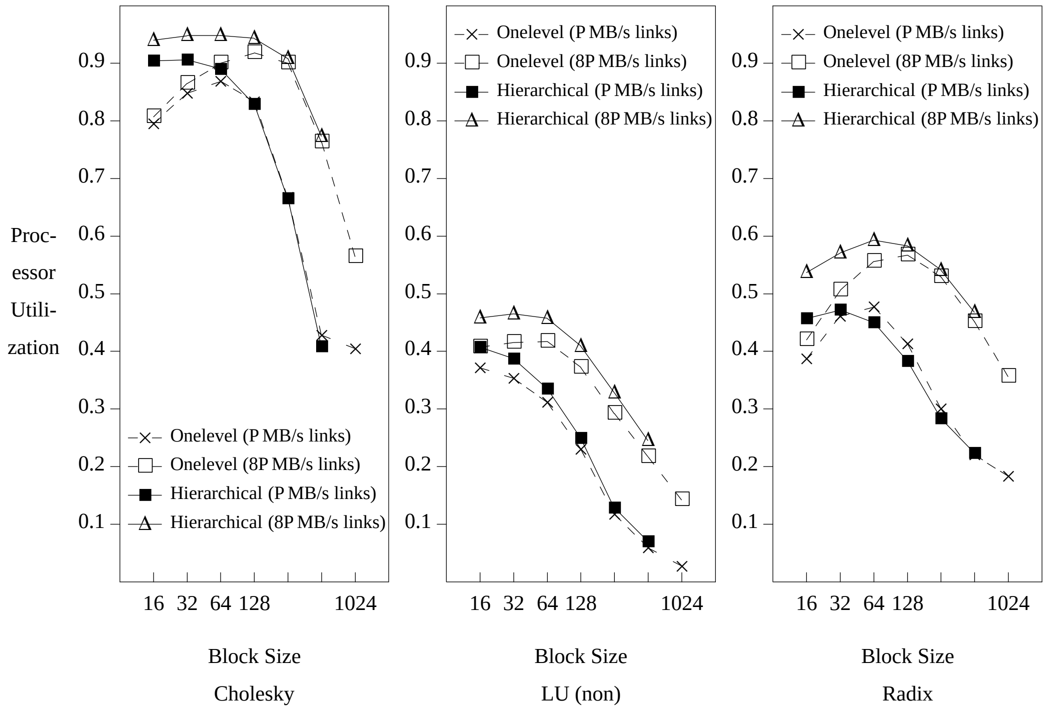

Figure 5.13: Processor utilization is lower at larger block sizes.

processor utilization is very low at large block sizes as expected. At very small block sizes (16 or 32 bytes), more remote accesses may be needed to acquire all data needed for processing than if the block size were 64 or 128 bytes. As a result, the processor utilization also drops at very small block sizes.

The drop in processor utilization at very small block sizes is magnified in the Cholesky and Radix applications with the one-level scheme due to an artifact in the simulation system model. As we discussed in Section 5.1, disk accesses are at the granularity of pages with the hierarchical scheme while disk accesses are at the granularity of blocks with the one-level scheme. At small block sizes, more disk accesses are needed in the one-level scheme to bring in all the data needed in block-sized chunks. The larger number of disk accesses lowers processor utilization in the one-level scheme at small block sizes. This effect is significant in Cholesky and Radix due to a significant number of disk accesses in these applications. In Cholesky, disk accesses are needed since the large data set does not fit in a node's physical memory. In Radix, the disk accesses are needed to initialize the array to be sorted. It should be noted that no real system accesses small blocks (less than 1 K bytes) from disk. Hence, no conclusions regarding real systems may be drawn from the results in Figure 5.13 with respect to performance of the one-level system versus the hierarchical scheme with small blocks.

5.1.6. Characterization and Measurement of Non-Local Read and Write Accesses

When a read or write request cannot be satisfied locally within the node, remote access is necessary to satisfy the request. Messages must be sent to one or more nodes to acquire the requested data with sufficient access permission. The type of the message(s) exchanged and the nodes that must be contacted are specified by the coherence protocol. They depend on the type of request (read or write). For example, consider the reply message received by the requesting node in the coherence protocol for the hierarchical scheme (see Section 4.3.4 in Chapter 4). A node issuing a read request will receive the read copy reply from the owner of the block (block owner) containing the requested data. But, a node issuing a write request will receive the ownership copy reply from the owner of the page (page owner) containing the requested data. The message types and the set of nodes contacted also depend on the dynamic shared state of the required data. If the requesting node has a copy of some block of the page containing the requested data, the request is sent to the owner of the page; otherwise, the request is sent to the default owner of the page. Therefore, each remote request will involve a different sequence of messages and will require a different set of tables to be looked up in different nodes.

Since the coherence protocol used for the hierarchically managed and one-level managed systems are different (see Section 4.3.4 in Chapter 4 and Section 5.1.1), remote requests also have different behavior on the two systems. The hierarchical scheme supports two levels of ownership and, as a result, more levels of forwarding than the one-level scheme. Remote requests can, therefore, involve more messages and more table lookups in the hierarchical scheme than in the one-level scheme.

We have used our simulations to profile the behavior of remote requests based on the different sequences of messages and nodes that are involved in satisfying the request and on the network latency and table lookup latency incurred in satisfying the request. Our study has profiled the behavior of remote requests on both the hierarchical and one-level schemes. In our discussion of the coherence protocol in Chapter 4, we identified different flows of the protocol that are possible in executing remote read and write requests (Figure 4.18 and Figure 4.21 in Chapter 4, for example). To assist in characterizing the behavior of remote requests, we have classified the various coherence protocol flows that are used to satisfy remote requests. In the simulation, we have measured the number of occurrences of each class of protocol flow in satisfying remote requests.

We present measurements of the fraction of shared accesses that require non-local access in Section 5.1.6.1. We introduce our protocol flow classification in

Section 5.1.6.2. Based on simulation results, in Section 5.1.6.3, we present the frequencies of occurrence of different classes of protocol flows for a variety of non-local access scenarios with both the one-level and hierarchical schemes. The latency of a remote access depends on the latency of the messages through the network and the latency of accessing tables at the servicing nodes. In Section 5.1.6.4, we present simulation results that quantify the contributions of the network latency and the table lookup latency to the overall latency of remote requests.

5.1.6.1. Measurement of the Miss Rate of Shared Accesses

A shared read or write request by a processor requires remote access only when the processor's request misses in the local memory of that node. We have measured the miss rate for shared accesses and show our results in Figure 5.14. Figure 5.14 shows the fraction of shared reads that require non-local access and the fraction of shared writes that require non-local access for both the one-level and hierarchical schemes. The results are shown for simulations with different block sizes. The overall miss rate for shared accesses can be calculated from the miss rates for shared reads and shared writes shown in Figure 5.14 and the number of shared reads and writes listed listed earlier in Table 5.5 (Section 5.1.2).

The figure shows that the miss rate for shared accesses is under 10% for Cholesky and Radix for nearly all block sizes. In the case of LU, the miss rate is higher since the data accessed by a process are not allocated contiguously. In all three applications, read accesses have better hit rate than write accesses. This is expected since write accesses hit only when the block is owned exclusively by the

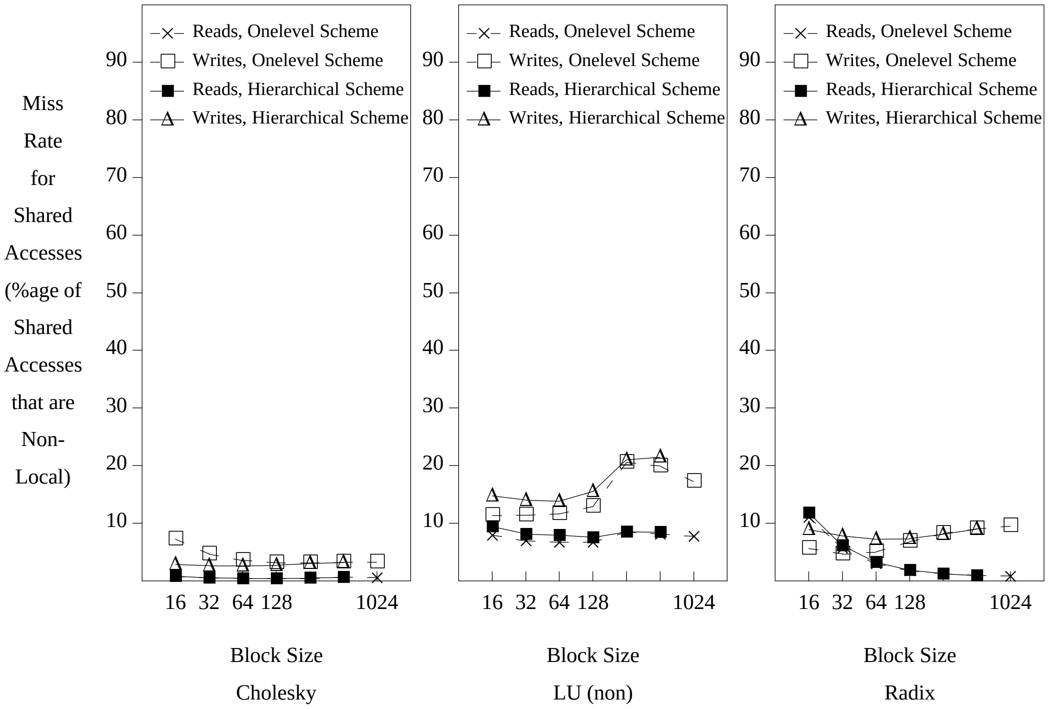

Figure 5.14: Miss Rate for Shared Accesses.

local node. Read accesses do not require exclusive access.

In general, the miss rate for shared reads decreases with increasing block size in both the one-level and the hierarchical schemes. With larger blocks, a larger chunk of data is transfered to satisfy a read miss. Thus, more subsequent accesses are likely to hit on the same block without requiring remote access. However, when the block size is increased beyond 128 bytes, miss rate can increase due to false sharing. This is especially the case in LU, where data modified by different nodes get mapped to the same block when the block size is larger than 128 bytes. When data modified by multiple nodes is mapped to the same block, read copies held by one node can be unnecessarily invalidated when another node writes to a different part of the block. As a result, more shared reads may require non-local access.

Shared writes also show a small decrease in miss rate with increasing block size at small block sizes. However, shared writes demonstrate higher increase in miss rate at large block sizes than shared reads. This is expected since write accesses require an exclusive copy of the block (unlike reads which require only a shared copy). With larger blocks, the block is more likely to be shared by other nodes. As a result, more write accesses may require remote access to invalidate read copies. With LU, the miss rate for shared writes is especially higher with blocks larger than 128 bytes since data modified by multiple nodes get mapped to the same block.

With LU and Radix, at small block sizes, the miss rates are slightly higher in the hierarchical scheme than in the one-level scheme using the same block size. This is most likely due to the fact that the SPT entry contention can be higher in the hierarchical scheme than the one-level scheme at small block sizes. Due to the page-level mapping of the SPT entry in the hierarchical scheme, there are fewer SPT entries for a given amount of storage in the hierarchical scheme. For example, in our simulations, the SPT has 2048 entries when simulating the one-level scheme with 16-byte blocks while the SPT has 128 entries when simulating the hierarchical scheme with 16-byte blocks (see Table 5.2 in Section 5.1). The contention for SPT entries is hence marginally higher in the hierarchical scheme at small block sizes. When there is contention for SPT entries, it is necessary to free SPT entries by evicting shared pages. Since the eviction discards shared data that is currently mapped to the node's local memory, subsequent accesses to the evicted page require remote access leading to a small increase in miss rate.

As will be demonstrated later in this subsection, the miss rate results are also influenced by the different implementation of disk accesses in the one-level and the hierarchical schemes. As we described earlier in Section 5.1, data is accessed from the disk in block-sized chunks in the one-level scheme while data is accessed in page-sized chunks in the hierarchical scheme. Consequently, a shared access miss that is satisfied from disk brings in a page of data in the hierarchical scheme, while it brings in a smaller block of data in the one-level scheme. The larger chunk of data that is transfered to satisfy the miss in the hierarchical scheme can potentially satisfy more subsequent accesses to the page locally, leading to lower miss rate than in the one-level scheme. However, the larger transfer size in the hierarchical scheme may not benefit subsequent accesses when data modified by multiple nodes are mapped to the same page. In such cases, intervening writes by other nodes writing to the page can invalidate sections of the page from local memory forcing subsequent accesses to the page to issue a remote request.

Application	Number of Shared Read Accesses Satisfied by Disk Access				
	Onelevel Scheme: 16 byte accesses	Onelevel Scheme: 32 byte accesses	Onelevel Scheme: 64 byte accesses	Onelevel Scheme: 256 byte accesses	Hierarchical Scheme: 1 KB accesses
Cholesky	68207	35194	18329	4812	1444
LU	25165	12621	6365	1152	123
Radix	64	64	64	64	64

Table 5.6: Number of Shared Reads Satisfied by Disk Accesses. Disk is accessed at the granularity of blocks with the one-level scheme, and at the granularity of pages in the hierarchical scheme.

In Table 5.6 and Table 5.7, we list the number of shared read misses that are satisfied from disk and the number of shared write misses that are satisfied from disk respectively, for various granularities of disk access. As expected, the number of accesses satisfied from disk is higher when granularity of disk access is small. The difference between the number of shared reads satisfied from disk in the one-level and hierarchical schemes is a very small fraction of the total number of shared reads (see the total number of shared reads in Table 5.5 in Section 5.1.2) in the application. Consequently, in Figure 5.14, there is only negligible benefit in terms of read miss rate to the larger transfer size from disk in the hierarchical scheme.

Application	Number of Shared Write Accesses Satisfied by Disk Access				
	Onelevel Scheme: 16 byte accesses	Onelevel Scheme: 32 byte accesses	Onelevel Scheme: 64 byte accesses	Onelevel Scheme: 256 byte accesses	Hierarchical Scheme: 1 KB accesses
Cholesky	142810	71966	36474	9707	2880
LU	3	3	3	3	3
Radix	164097	82048	41024	10256	2564

Table 5.7: Number of Shared Writes Satisfied by Disk Accesses. Disk is accessed at the granularity of blocks with the one-level scheme, and at the granularity of pages in the hierarchical scheme.

The difference between the number of shared writes satisfied from disk in the one-level scheme with 16 byte blocks and in the hierarchical scheme is roughly 5% of the total number of shared writes in Cholesky and is roughly 2.5% of the total number of shared writes in Radix (Table 5.7). In Cholesky, it appears that the larger granularity of disk access in the hierarchical scheme allows more subsequent accesses to be satisfied locally leading to lower miss rate in the hierarchical scheme (see Figure 5.14) than in the one-level scheme at small block sizes. This is not the case in Radix. In the Radix application, after each phase of sorting, the keys being sorted are permuted to their new position after that stage (Section 5.1.2). These shared writes are satisfied by disk accesses. Since the positions updated by different nodes are mapped to the same page, the larger transfer size from disk in the hierarchical scheme does not provide any benefit in terms of local miss rate (there is a benefit of reduced wait for disk accesses). As discussed previously, writes to the same page by other nodes invalidate sections of the page forcing subsequent accesses to issue a remote request. With the LU application, the smaller granularity of disk access in the one-level scheme can actually reduce the number of non-local writes (although more latency is incurred in more disk accesses). In the LU application, processes read their data before writing to them. When the initial read misses are satisfied by transfering small blocks from the disk, the block is likely to be private to the node allowing the subsequent write to hit locally. When data is transfered from disk in the granularity of pages in the hierarchical scheme, the page is installed in the node that receives the page from disk (the node that first reads the page). Read misses in other nodes are satisfied by requesting read copies from the node that received the page from disk (the page owner). Since these copies only allow read access, subsequent writes to the block cannot be satisfied locally and require remote access.

5.1.6.2. Classification of Protocol Flows Used to Satisfy Non-Local Requests

Our classification of the protocol flows used to satisfy remote requests in our coherence protocol is based on the observation that different protocol flows differ primarily in the sequence of messages and node hops needed to reach the owner of the block. Once the owner is reached, the protocol flow follows a fixed pattern for all read requests and another fixed pattern for all write requests. For example, a read request can arrive at the owner of the requested block in one of several ways — directly from the requester, forwarded by the page owner, or forwarded by the default owner. However, when a read request reaches the owner of the required block, in our coherence protocols, a copy of the block is always sent directly to the requester. Likewise, when a write request has reached the current owner of the block, the ownership of the block is always granted through the page owner (in the hierarchical scheme) or through the default owner (in the one-level scheme).

The number of messages exchanged to reach the owner of the block in a protocol flow is indicative of the complexity of the protocol flow. A protocol flow that requires a larger number of these messages incurs higher network latency and potentially incurs higher latency in accessing more coherency tables at intermediate nodes. Hence, we classify protocol flows by the <u>number</u> of messages exchanged in the protocol flow to reach the owner of the block. We refer to the sequence of protocol message and node hops used to reach the owner of the requested block as the *hop-sequence* of the protocol flow. The *hop-sequence type number* of a protocol flow is the number of messages exchanged before the owner of the block is reached (i.e., the number of messages in the *hop-sequence*).

To illustrate this classification, consider a few examples. Suppose a node requires read access to a block that it does not possess locally. If the node knows the identity of the block owner, the read request can be sent to the owner of the block directly. Since only one message was used to reach the owner of the block, the read request is satisfied by a protocol flow with *hop-sequence* type of 1. If the requesting node does not know the identity of the block owner, the read request may be sent to the page owner who forwards it to the block owner. The *hop-sequence* type of this protocol flow is 2 since it uses two messages to reach the owner of the block. Later in this subsection, we discuss in detail more examples of *hop-sequences* of different type numbers that are used to service read and write requests in the one-level and hierarchical schemes.

It should be noted that our classification is based only on the <u>number</u> of messages and not on the types of messages that are sent or the identity of intermediate nodes (page owner, default owner etc.) that are accessed. Hence, different protocol flows that use different sequences of messages or different sequences of nodes have the same *hop-sequence* type numbers if they use the same number of messages. For example, a protocol flow that forwards a read request through the page owner to the block owner has an *hop-sequence* type of 2. A protocol flow that forwards the read request through the default owner to the page owner which also owns the block (the page owner is also the block owner) also has an *hop-sequence* type of 2. Alternative classifications based on the sequence of messages exchanged, on the sequence of nodes accessed or based on the set of tables looked up are also possible. Our classification based on the number of messages is simple and captures the key differences in the protocol flows.

Table 5.8 presents example *hop-sequences* of protocol flows used to satisfy

Hop-sequences of Protocol for Read Access in One-level Scheme					
Type of hop-sequence of protocol	Nodes (Sequence)	Messages (Sequence)	Tables Accessed at Node	Table Access Latency (cycles)	Comments
Type 0					Not possible
Type 1	Requestor → Default Owner → Requestor	RREQ RDCPY	PPT PPT+SPT Cache	3 7 1	 MPT access may be needed
Type 2	Requestor → Default Owner → Page Owner → Requestor	RREQ RREQ RDCPY	PPT PPT+MPT PPT+SPT Cache	3 43 7 1	 MPT access may not be needed Copy Directory updated in parallel
Type 3	Requestor → Default Owner → Page Owner → Default Owner → Requestor	RREQ RREQ RREQ RDCPY	PPT PPT+MPT PPT PPT+SPT Cache	3 43 3 7 1	 MPT access may not be needed Page owner has changed Default owner is also page owner
Type 4	Requestor → Default Owner → Page Owner → Default Owner → New Page Owner → Requestor	RREQ RREQ RREQ RREQ RDCPY	PPT PPT+MPT PPT PPT+MPT PPT+SPT Cache	3 43 3 43 7 1	 MPT access may not be needed Page owner has changed MPT access may not be needed Copy Directory updated in parallel
Type 5	Requestor → Default Owner → Page Owner → Default Owner → Page Owner → Default Owner → New Page Owner → Requestor	RREQ RREQ RREQ RREQ RREQ RREQ RDCPY	PPT PPT+MPT PPT PPT+MPT PPT PPT+MPT PPT+SPT Cache	3 43 3 43 3 43 7 1	 MPT access may not be needed Page owner has changed MPT access may not be needed Page owner has changed MPT access may not be needed Copy Directory updated in parallel

Table 5.8: Hop-sequences of protocol for read access in one-level scheme.

remote read requests in the one-level scheme. An example *hop-sequence* is listed for each *hop-sequence* type number between 0 and 5. In each example, the table identifies the sequence of nodes involved in the *hop-sequence* (the Requester initiates the access), the messages sent by these nodes in sequence, the mapping tables accessed at each node and the latency incurred in accessing tables at each node.

A *hop-sequence* type 0 is not valid since it implies a read-hit in the requesting node. The example for *hop-sequence* type 1 describes the protocol flow where the read request is received and serviced by the default owner itself. The read access initiates a search of the local Present Pages Table (PPT) (or the Present Block Table). When the block is not found in the PPT, the read request *RREQ* is sent to the default owner by the requesting node. When the default owner receives the remote read request, the PPT, and then the SPT are looked up to determine the owner of the block. When the default owner is also the owner of the block, a read copy is sent back to the requester with the message *RDCPY*. When the read copy is received at the requester, the read-copy of the data is directly passed to the cache at the requester for immediate access. The latency of accessing tables at each node is computed based on the access latency of individual tables specified in Table 5.1.

A protocol flow with *hop-sequence* type 2 is required when the requested block is owned by some node other than the default owner. In this case, the default owner must identify the current owner by looking up its Migrated Page Table (MPT) (or Migrated Block Table). The read request is forwarded to the owner by the default owner. The reply is sent directly by the owner of the block. Protocol flows with *hop-sequence* type greater than 2 are possible when the owner of the block changes while the read request is in progress. In these cases, the owner prior to the change forwards the request back to the default owner.

As we mentioned earlier, alternative examples with a different sequence of nodes and messages are possible for many *hop-sequence* types. In addition, even for the example *hop-sequences* illustrated here, the mapping tables accessed at the nodes can be different. For instance, an MPT access may be needed at the default owner for the read access with *hop-sequence* type 1 if the page is not present at the default owner and the default owner checks the MPT to find that no other node owns the page, i.e., the page is on disk.

Table 5.9 presents the example cases for read accesses in the hierarchical scheme. The example for *hop-sequence* type 1 describes the protocol flow where the read request is received and serviced by the page owner itself. The Present Pages Table (PPT) is first searched at the node issuing the read access. In the example scenario, the page that holds the accessed data is identified to be in the shared state. A lookup of the Shared Page Table (SPT) entry for the page is initiated. When the required block is indicated absent in the SPT, the identity of the page owner is ascertained from the SPT entry. A read request *RREQ* is sent to this page owner. When the page owner receives the remote read request, the PPT, and then the SPT are looked up to determine the owner of the block. When the page owner also possesses ownership of the block, the read copy (*RDCPY*) is sent back to the requester.

The example for *hop-sequence* type 2 deviates from the previous example when the requested block is owned by some node other than the page owner. In this case, the identity of the block owner is determined by the page owner from the copy directory (CoD). The read request (*RREQ*) is forwarded to the block owner. When the request is received by the block owner, the PPT and the SPT are looked up to ascertain ownership of the block, after which the read copy (*RDCPY*) is sent directly to the requester.

Hop-sequences of Protocol for Read Access in Two-level Scheme					
Type of hop-sequence of protocol	Nodes (Sequence)	Messages (Sequence)	Tables Accessed at Node	Table Access Latency (cycles)	Comments
Type 0					Not possible
Type 1	Requestor → Page Owner → Requestor	RREQ RDCPY	PPT+SPT PPT+SPT Cache	7 7 1	Page owner identified by SPT Page owner is also block owner
Type 2	Requestor → Page Owner → Block Owner → Requestor	RREQ RREQ RDCPY	PPT+SPT PPT+SPT+CoD PPT+SPT Cache	7 14 7 1	Page owner identified by SPT Forwarded to block owner Copy Directory updated in parallel
Type 3	Requestor → Default Owner → Requestor → Page Owner → Requestor	RREQ PGOWN RREQ RDCPY	PPT PPT+SPT PPT PPT+SPT Cache	3 7 3 7 1	Page owner not known Bi-lateral response Page owner was identified Page owner is also block owner
Type 4	Requestor → Default Owner → Requestor → Page Owner → Block Owner → Requestor	RREQ PGOWN RREQ RREQ RDCPY	PPT PPT+SPT PPT PPT+SPT+CoD PPT+SPT Cache	3 7 3 14 7 1	Page owner not known Bi-lateral response Page owner was identified Forwarded to block owner Copy Directory updated in parallel
Type 5	Requestor → Page Owner → Block Owner → Default Owner → Requestor → Page Owner → Requestor	RREQ RREQ RREQ PGOWN RREQ RDCPY	PPT+SPT PPT+SPT+CoD PPT+SPT PPT+SPT PPT+SPT PPT+SPT Cache	7 14 7 7 7 7 1	Page owner identified by SPT Forwarded to block owner Block owner has changed Response to requestor Read request re-issued Page owner is also block owner

Table 5.9: Hop-sequences of protocol for read access in hierarchical scheme.

When no blocks from the page that has the required data are present at the node issuing the request (examples for *hop-sequence* types 3 and 4), the node cannot identify the owner of the page (no PPT entry for the page and thus no SPT access). Hence, the read request is sent to the default owner. Since the protocol uses bilateral exchanges, when the default owner is not the owner of the page, a bilateral response identifying the page owner (*PGOWN*) is sent back to the requester. In the example situations for *hop-sequence* types 3 and 4, the default owner possesses copies of the blocks of the page allowing it to determine the page owner by accessing the SPT, without requiring Migrated Page Table (MPT) access. When the *PGOWN* message is received, the request is sent directly to the page owner by the requester. When the page owner is also the owner of the block, the owner of the data is reached with three messages (*hop-sequence* type 3). The subsequent protocol flow is similar to the example for *hop-sequence* type 1. When the required block is owned by some node other than the page owner, an additional message (*hop-sequence* type 4) is needed to reach the owner of the block. Protocol flows with *hop-sequence* type number greater than 4 can occur when block or page ownership moves during the access.

Table 5.10 presents the example cases for write accesses in the one-level scheme. When the node issuing the write access is also the owner of the block, no messages are required to reach the owner of the block (*hop-sequence* type 0). Present Pages Table (PPT), Shared Pages Table (SPT) and copy directory accesses are needed to determine the locations of read copies. The copies are invalidated (*INV*). When all acknowledgments (*ACK*) are received, the write access is allowed to progress.

When the node issuing the write access is not the owner of the block, the write request *WREQ* must be sent to the default owner. If the default owner is also the owner of the block (*hop-sequence* type 1), the default owner performs the invalidations. When all acknowledgments are received, the ownership copy (*WDATA*) is sent to the requester after the identity of the new owner is updated in

Hop-sequences of Protocol for Write Access in One-level Scheme					
Type of hop-sequence of protocol	Nodes (Sequence)	Messages (Sequence)	Tables Accessed at Node	Table Access Latency (cycles)	Comments
Type 0	Requestor → Sharing Node → Requestor	INV ACK	PPT+SPT+CoD PPT+SPT Cache	14 7 1	Access by page owner Copy invalidated When all ACKs received
Type 1	Requestor → Default Owner → Sharing Node → Default Owner → Requestor	WREQ INV ACK WDATA	PPT PPT+SPT+CoD PPT+SPT PPT+SPT+MPT Cache	3 14 7 47 1	 Default owner is also page owner Copy invalidated New page owner updated in MPT
Type 2	Requestor → Default Owner → Page Owner → Sharing Node → Page Owner → Default Owner → Requestor	WREQ WRBCK INV ACK BLKBCK WDATA	PPT PPT+SPT PPT+SPT+CoD PPT+SPT PPT+SPT PPT+SPT+MPT Cache	3 7 14 7 7 47 1	 MPT access may be needed Invalidate for writeback Copy invalidated Write back (after all ACKs) New page owner updated in MPT
Type 3					Not possible
Type 4					Not possible
Type 5					Not possible

Table 5.10: Hop-sequences of protocol for write access in one-level scheme.

the Migrated Page Table (or Migrated Block Table). The example for *hop-sequence* type 2 describes the case when the default owner is not the owner of the block. In this case, a second message (*WRBCK*) is needed to reach the owner of the block. Upon receiving the write-back request, invalidates are sent by the block owner to all read copies. When all acknowledgments are received, the block is sent back (*BLKBCK*) to the default owner. The ownership is forwarded to the requester (*WDATA*) by the default owner after the MPT is updated.

The MPT entry is locked by the default owner when a write request is in progress. If the MPT entry is locked when a new write request is received, the new write request must be retried. A retried write request will follow one of the three *hop-sequences* described earlier. Hence, there are no protocol flows with *hop-sequence* type greater than 2.

The example *hop-sequences* for write accesses in the hierarchical scheme are shown in Table 5.11. The example for *hop-sequence* type 0 is identical to the equivalent example discussed for writes in the one-level scheme. The examples for *hop-sequence* types 1 and 2 are also similar to the equivalent cases discussed for writes in the one-level scheme. The only difference is that ownership transfers at writes are coordinated by the default owner in the one-level scheme, while they are coordinated by the page owner in the hierarchical scheme. In the hierarchical scheme, the page owner commits the transfer of block ownership by updating its SPT, without any changes to the MPT. *Hop-sequence* types 1 and 2 are possible in the hierarchical scheme only when the node issuing the write can identify the page owner from the SPT entry for the page (i.e., the node must possess other blocks of the page).

When the node issuing the write cannot identify the page owner (no SPT access), the write request must be sent to the default owner. This is the case in the examples for *hop-sequence* types 3 and 4. As we discussed, for read requests in the hierarchical scheme, the default owner must be queried for the identity of the page owner using a bilateral exchange. The examples for *hop-sequence* type 3 and type 4 differ from the examples for *hop-sequence* type 1 and type 2 only in the need for two additional messages for the bilateral exchange. In the example for *hop-sequence* type 3, the page ownership changes during the write access.

Hop-sequences of Protocol for Write Access in Two-level Scheme					
Type of hop-sequence of protocol	Nodes (Sequence)	Messages (Sequence)	Tables Accessed at Node	Table Access Latency (cycles)	Comments
Type 0	Requestor →	INV	PPT+SPT+CoD	14	Access by block owner
	Sharing Node →	ACK	PPT+SPT	7	Copy invalidated
	Requestor		Cache	1	When all ACKs received
Type 1	Requestor →	WREQ	PPT+SPT	7	Page owner identified by SPT
	Page Owner →	INV	PPT+SPT+CoD	14	Page owner is also block owner
	Sharing Node →	ACK	PPT+SPT	7	Copy invalidated
	Page Owner →	WDATA	PPT+SPT	7	Copy Directory updated in parallel
	Requestor		Cache	1	
Type 2	Requestor →	WREQ	PPT+SPT	7	Page owner identified by SPT
	Page Owner →	WRBCK	PPT+SPT+CoD	14	Request block write-back
	Block Owner →	INV	PPT+SPT+CoD	14	Invalidate for write-back
	Sharing Node →	ACK	PPT+SPT	7	Copy invalidated
	Block Owner →	BLKBCK	PPT+SPT	7	Write back (after all ACKs)
	Page Owner →	WDATA	PPT+SPT	7	Copy Directory updated in parallel
	Requestor		Cache	1	
Type 3	Requestor →	WREQ	PPT	3	Page owner not known
	Default Owner →	PGOWN	PPT+SPT	7	Bi-lateral response
	Requestor →	WREQ	PPT	3	Page owner was identified
	Page Owner →	INV	PPT+SPT+CoD	14	Page owner is also block owner
	Sharing Node →	ACK	PPT+SPT	7	Copy invalidated
	Page Owner →	WDATA	PPT+SPT	7	Copy Directory updated in parallel
	Requestor		Cache	1	
Type 4	Requestor →	WREQ	PPT	3	Page owner not known
	Default Owner →	PGOWN	PPT+SPT	7	Bi-lateral response
	Requestor →	WREQ	PPT	3	Page owner was identified
	Page Owner →	WRBCK	PPT+SPT+CoD	14	Request block write-back
	Block Owner →	INV	PPT+SPT+CoD	14	Invalidate for write-back
	Sharing Node →	ACK	PPT+SPT	7	Copy invalidated
	Block Owner →	BLKBCK	PPT+SPT	7	Write back (after all ACKs)
	Page Owner →	WDATA	PPT+SPT	7	Copy Directory updated in parallel
	Requestor		Cache	1	
Type 5	Requestor →	WREQ	PPT	3	Page owner not known
	Default Owner →	PGOWN	PPT+SPT	7	Bi-lateral response
	Requestor →	WREQ	PPT	3	Page owner was identified
	Page Owner →	WREQ	PPT	3	Page owner has changed
	Default Owner →	WRBCK	PPT+SPT+CoD	14	Default owner is also page owner
	Block Owner →	INV	PPT+SPT+CoD	14	Invalidate for write-back
	Sharing Node →	ACK	PPT+SPT	7	Copy invalidated
	Block Owner →	BLKBCK	PPT+SPT	7	Write back (after all ACKs)
	Default Owner →	WDATA	PPT+SPT	7	Copy Directory updated in parallel
	Requestor		Cache	1	

Table 5.11: Hop-sequences of protocol for write access in hierarchical scheme.

5.1.6.3. Breakdown of Non-Local Requests by Hop-Sequence Type of Protocol Flow

During simulation, the *hop-sequence* type number of the protocol flow used to satisfy each remote request was monitored. To eliminate cold-start accesses from this measurement, non-local accesses which fetch the data from disk are not included in the measures. The number of such accesses which fetch the data from disk is tabulated in Tables 5.6 and 5.7 in Section 5.1.6.1. In this subsection, these measurements are used to present a breakdown of the frequencies of occurrence of different classes (*hop-sequence* types) of protocol flows. Results are presented for a range of block sizes, for reads and writes, with the one-level and hierarchical schemes.

Figure 5.15 shows the breakdown of remote read accesses in the one-level scheme simulation, based on their *hop-sequence* type. For each application, we measured the breakdown in several simulation runs, each at a different block size. The breakdown is plotted at several block sizes from 16 bytes through 1 KB in Figure 5.15. At any block size, the total height of the bar is the total number of remote read accesses during the entire execution of the application. The number of remote read accesses that were satisfied using protocol flows of a specific *hop-sequence* type number is indicated by the height of the segment of the bar that corresponds to that type.

In most cases, the total number of non-local read accesses decreases with increasing block size. This is because larger block sizes capture more spatial locality. When large blocks are transfered to service a read miss, more subsequent accesses are likely to hit on the same block without requiring remote access. In the

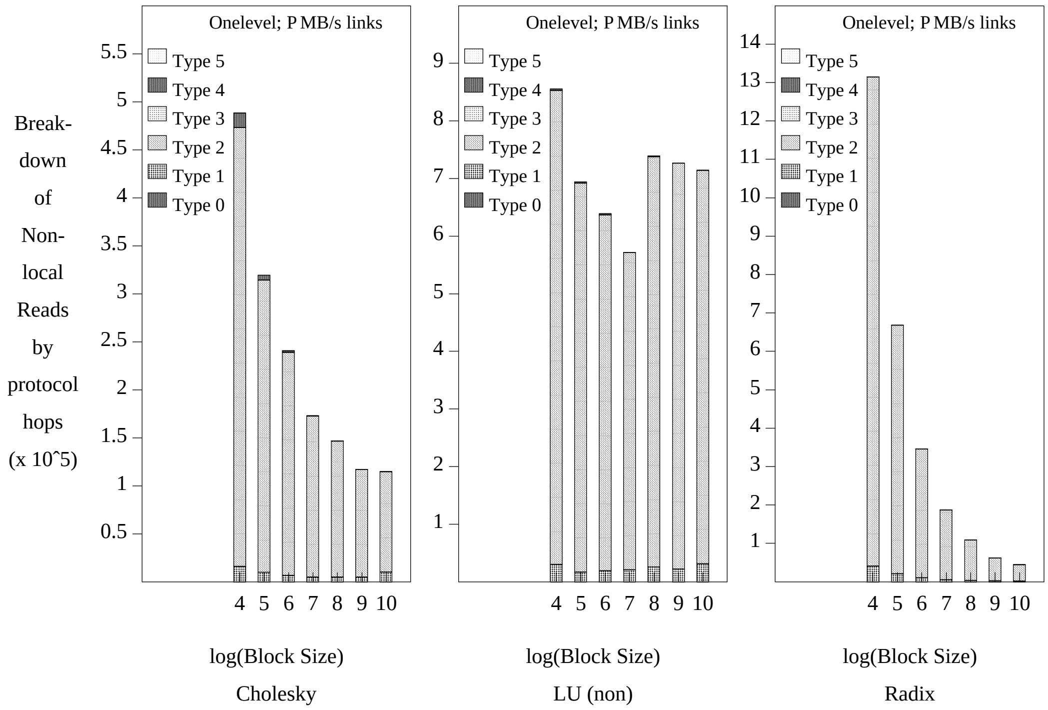

Figure 5.15: Break down of read accesses by hop-sequence type in the one-level scheme.

case of LU, when the block size is increased beyond 128 bytes, data accessed by different nodes get mapped to the same block. As a result of this false sharing, a larger number of read access require remote access.

From Figure 5.15, it is also clear that most read accesses have *hop-sequence* type 2 in the one-level scheme. This *hop-sequence* type corresponds to the scenario where the read request is sent to the default owner and then forwarded by the default owner to the current page (block) owner (see Table 5.8). The hop to the default owner is expected since a node requesting a read copy typically does not know the owner of the page (block) in the one-level scheme. The node knows the identity of the page owner only when it is the default owner. The additional hop to forward the request from the default owner to the current owner is also common because the default owner is rarely also the owner. This is due to two reasons. First, the ownership of a block keeps changing because different nodes write to the same block. Second, the assignment of blocks to default owner nodes and processes to nodes are done independently without regard to the behavior of the application. When the block is not written by multiple processes in the application, this points to an opportunity for improving performance. Application performance can be improved by assigning processes to nodes which are the default owner of blocks which are written to mostly by these processes. Figure 5.15 also shows that accesses are rarely satisfied by *hop-sequence* type 1 protocol flows. This occurs when the node requesting read access is also the default owner or when the default owner is also the page owner.

Figure 5.16 shows the breakdown for read accesses in the hierarchical scheme. The figure shows that a greater proportion of accesses are satisfied by protocol flows with *hop-sequence* type 1 in the hierarchical scheme than in the one-level scheme. This indicates that the node issuing the request frequently sends the request to the owner of the block directly. Since only the page owner knows the block owner, we can draw two conclusions from this observation. First, the requesting node frequently has other blocks of the page and, hence, knows the identity of the page owner. Second, it indicates that the page owner owns the required block in the hierarchical scheme more frequently than the default owner owns the block in the one-level scheme. This is to be expected since the page owner is determined dynamically based on the application's access pattern to the page, while the default owner is determined statically, independent of the

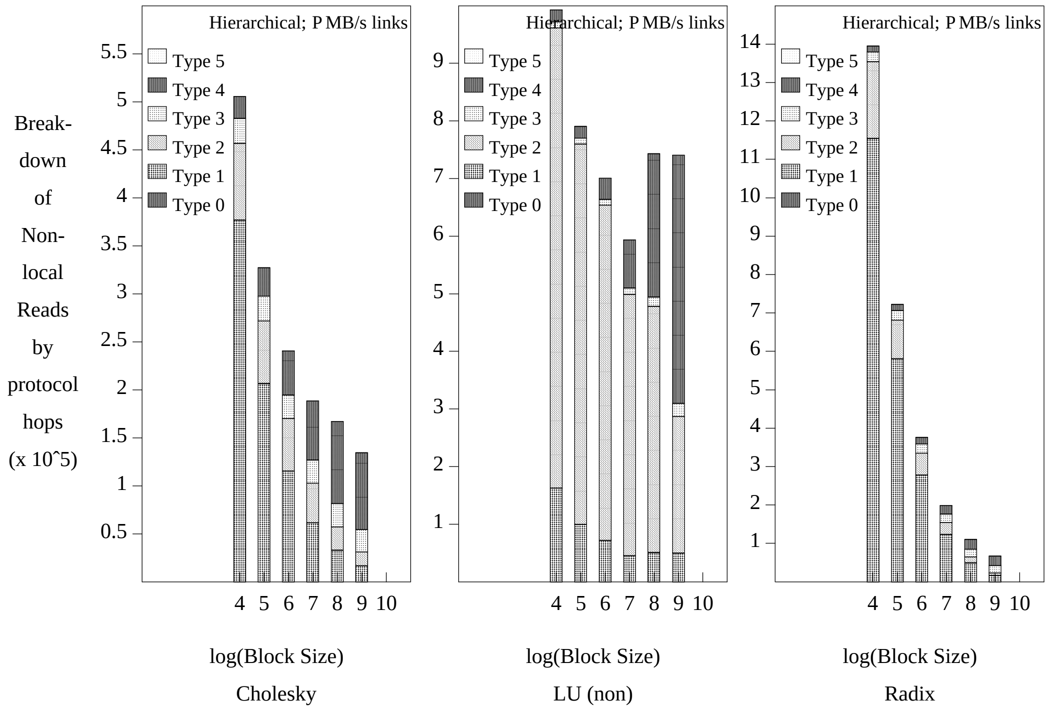

Figure 5.16: Break down of read accesses by hop-sequence type in the hierarchical scheme.

application.

With Cholesky and Radix, for small blocks, the predominance of requests serviced by protocol flow with *hop-sequence* type 1 shows that it is common that the requesting node knows the page owner and the page owner is also the block owner. As block size increases, the first access to a page brings more of the data needed from the page, allowing more subsequent accesses to be satisfied locally. Also, since there are fewer blocks per page at larger block sizes, there are fewer additional blocks to be requested after the first access to the page. For both these reasons, it is less likely to have many accesses which will reach the block owner in one hop at larger block sizes. This leads to the interesting result where, with increasing block size, the total number of remote reads in Cholesky and Radix decreases mostly by the reduction in the number of reads with *hop-sequence* type 1 protocol flow. In the application kernel LU, forwarding from the page owner to the block owner is more likely since data modified by multiple processors are mapped to the same 1 KB page (recall from Section 5.1.2 that the LU kernel is a less-optimized version where data modified by a process is <u>not</u> allocated contiguously). This results in a larger proportion of accesses with *hop-sequence* type 2.

Protocol flows with *hop-sequence* type higher than 2 are needed in the hierarchical scheme when the default owner must also be contacted (see Table 5.9). This occurs more frequently with large blocks. With larger blocks, there are fewer blocks per page and there is greater false-sharing. Hence, when an access is issued, it is more often the case that other blocks of the page are not resident locally. This increases the proportion of accesses requiring default owner lookup (since page owner is not known locally). The higher false sharing associated with the larger block size also increases the proportion of accesses where the request must be forwarded to the block owner by the page owner. Both these factors contribute to a higher count of of accesses that are satisfied by protocol flows with *hop-sequence* types 3 and 4 at higher block sizes. (Recall that default owner lookup requires <u>two</u> additional messages due to the bilateral exchange.)

Figure 5.16 also shows that with increasing block size, the number of reads satisfied by protocol flows with *hop-sequence* type 4 increases more sharply than the number of reads satisfied by protocol flows with *hop-sequence* type 3. This is most likely due to the fact that larger blocks have one effect when most of a page is modified and a different effect when most of a page is unmodified. When most of a page is unmodified, most of the page resides with the page owner and an access to the page from a node that has no other blocks of the page is more likely to be satisfied by a protocol flow with *hop-sequence* type 3. When the block size increases, the first access to a page brings more of the data. But, it does not affect the number of accesses that are issued when a node has no other blocks of the page. Therefore, the number of accesses with *hop-sequence* type 3 is largely unchanged when the block size is increased. When most of a page is modified, data of the page is likely to be owned by nodes other than the page owner (nodes which modify the data). In this case, an access to the page from a node that has no other blocks of the page is more likely to be satisfied with a protocol flow of *hop-sequence* type 4. As before, with larger blocks, more of the data is brought by the first access to the page. However, with larger blocks, the number of writes to a block is also higher due to the higher false sharing. Data brought by the first access to the page is, hence, more frequently invalidated. Hence, it is more likely (with pages where most of the page is modified) that the next read access is also issued when the node has no other blocks of the page, requiring default owner lookup. Thus, the larger block size increases the number of reads which must be satisfied using protocol flow with *hop-sequence* type 4. As mentioned earlier, with LU, data modified by multiple processors get mapped to the same 1 KB page. Hence, the block is frequently owned by a node other than the page owner. This results in a larger number of protocol flows with *hop-sequence* type 4 than in other applications.

We can compare the behavior of remote read accesses in the one-level and the hierarchical schemes (Figures 5.15 and 5.16). Most remote read accesses reach the owner of the block within two protocol hops in both schemes. However, with the hierarchical scheme, it is more likely for a remote read access to require more than two protocol hops. This is because the hierarchical scheme uses bilateral exchanges to query the default owner and supports an additional level of ownership.

Figures 5.15 and 5.16 also show that at small block sizes, the total number of non-local read accesses is marginally higher in the hierarchical scheme than in the one-level scheme using the same block size. As we discussed in Section 5.1.6.1, this is most likely due to the fact that SPT entry contention can be higher in the hierarchical scheme than the one-level scheme at small block sizes. Since each SPT entry corresponds to a page of shared data in the hierarchical scheme, the hierarchical scheme has fewer SPT entries (e.g., 128 entries in the hierarchical scheme and 2048 entries in the one-level scheme with 16-byte blocks in our simulations, as listed in Table 5.2 in Section 5.1.2) than the one-level scheme. Hence, it is more likely that shared data that is mapped to local memory is evicted to free an SPT entry in the hierarchical scheme. Subsequent reads that would have been satisfied locally by the evicted page can only be satisfied with non-local access. This can lead to a small increase in the number of non-local read accesses in the hierarchical scheme.

Figure 5.17 shows the breakdown of remote write accesses based on their *hop-sequence* type in the one-level scheme. Protocol flows with *hop-sequence* type 2 are predominant. This is expected since it corresponds to the most common case when the default owner transfers the write ownership to the requester from another node. Cholesky and LU display a significant number of cases when a

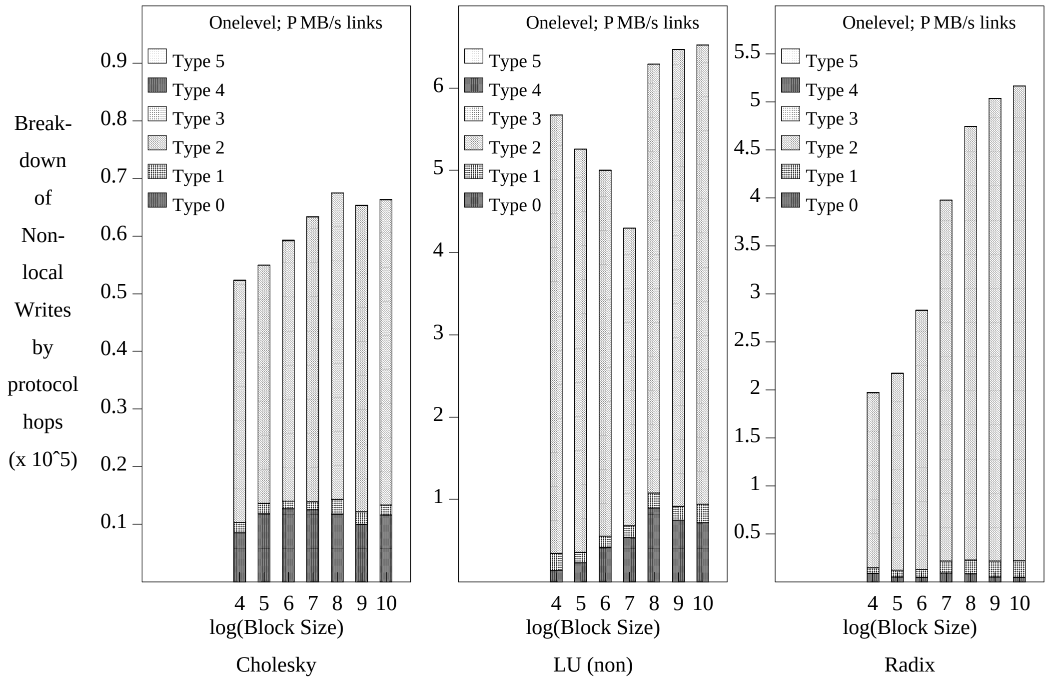

Figure 5.17: Break down of write accesses by hop-sequence type in the one-level scheme.

shared block is written by the owner itself (*hop-sequence* type 0), requiring invalidation only.

The breakdown of remote write accesses based on their *hop-sequence* type in the hierarchical scheme is shown in Figure 5.18. More write accesses are satisfied with *hop-sequence* type 1 protocol flow in the hierarchical scheme than in the one-level scheme. As we described in the discussion of read access behavior, this indicates that frequently the requesting node knows the identity of the page owner, and frequently the page owner is also the block owner. In the application kernel LU, the page owner frequently does not own the block, accounting for a higher proportion of accesses serviced by *hop-sequence* type 2 protocol flows (similar to the read access behavior discussed earlier).

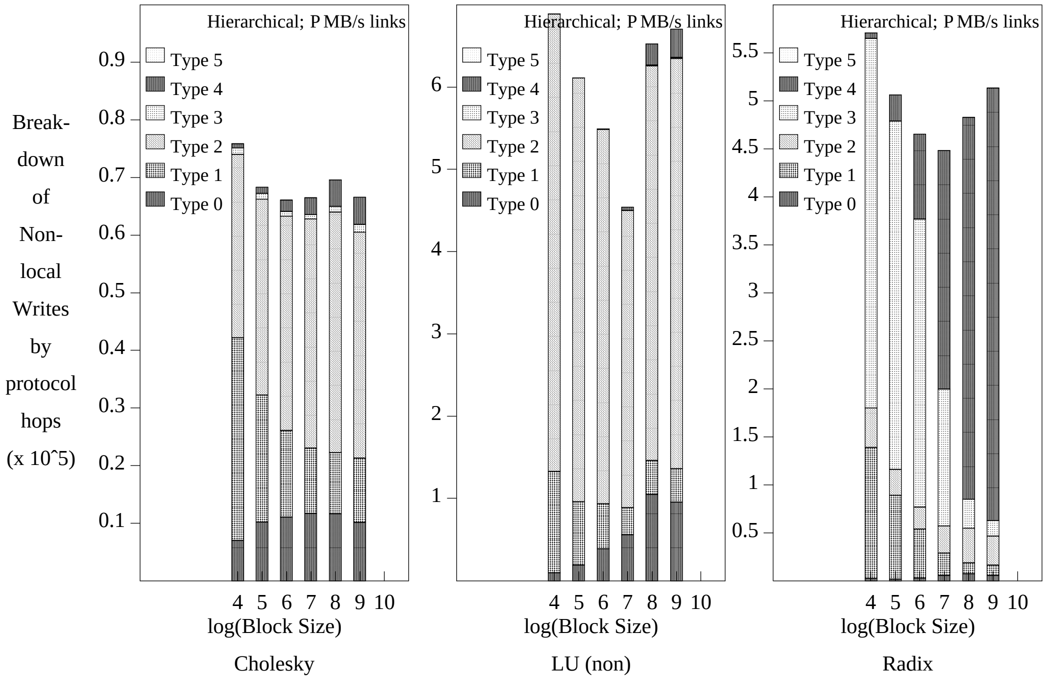

Figure 5.18: Break down of write accesses by hop-sequence type in the hierarchical scheme.

The behavior of write accesses in the Radix kernel is significantly different on the one-level and the hierarchical systems (Figures 5.17 and 5.18). We describe the reasons for the differences below. In the Radix kernel, processes initialize (write) the keys to be sorted at the start. In addition, after each phase of the sort, the keys being sorted are written to their new positions in the array. These array locations may not be mapped to the local memory of the processor writing the location. Therefore, many write accesses are first accesses to a page. Since these pages are not mapped to the physical memory of the requesting node at the start of execution, these write accesses require a default owner lookup. This results in a higher proportion of protocol flows with *hop-sequence* types 3 and 4. At higher block sizes, there are fewer blocks per page and higher false sharing. As previously described in the discussion of read access behavior in the hierarchical scheme, these lead to a high incidence of protocol flows where the default owner, the page owner and the block owner are all accessed (*hop-sequence* type 4).

Another difference in the behavior of Radix on the two schemes is that with small blocks the overall number of non-local write accesses is higher with the hierarchical scheme. This difference is partly an artifact of the fact that the disk is accessed in block-sized chunks in the one-level scheme, while it is accessed in page-sized chunks in the hierarchical scheme. In the one-level scheme, these cold-start writes fetch the required block from disk and are, hence, not counted in our non-local write access counts (e.g., about 1.6×10^5 accesses with 16-byte blocks, as listed in Table 5.7 in Section 5.1.6.1). In the hierarchical scheme, the first such write access brings the entire page from disk. All subsequent writes to blocks of the page must be satisfied by the page owner. These are not disk accesses and hence, are included in the count of non-local write accesses in the hierarchical scheme. The higher number of non-local accesses in the hierarchical scheme at small blocks can also be attributed to the higher contention for SPT entries in the hierarchical scheme.

It is also apparent from Figures 5.17 and 5.18 that the overall number of non-local write accesses do not decrease with increasing block size, unlike the number of non-local read accesses. This is partly due to the fact that these figures do not include the counts of non-local writes that were satisfied from disk. We have previously discussed the effect of block size on the total number of non-local writes (including those that are satisfied by disk accesses) in Section 5.1.6.1. As we pointed out in Section 5.1.6.1, the total number of non-local write accesses does decrease slightly at small block sizes. With large blocks, it is more likely that the block is shared by other nodes. Since writes require exclusive access to the block, more writes require remote access to invalidate these copies at larger block sizes.

5.1.6.4. Breakdown of Non-Local Request Latency

The latency of a non-local access depends on the network latency of the messages exchanged for the access and the access latency of coherency tables accessed at nodes involved in the access. We have investigated the contributions of message latency and table access latency to the latency of non-local accesses. We describe our investigation and its results below.

The most direct way to analyze the latency of non-local accesses is to trace the progress of each non-local access. To accomplish this trace, the simulator must be instrumented to follow the protocol flow's progress through different nodes. At each stage of the protocol flow, the latency of the messages exchanged and tables accessed must be measured and accumulated by the simulator. For these capabilities, the messages must be tagged with data structures that identify the source non-local access which caused the generation of the message and track the latencies accumulated until that stage of the protocol flow. The data collection mechanisms must also address the issue that asynchronous replacement of data can prematurely satisfy the non-local access before the completion of the protocol flow. To illustrate this issue, consider the case of an asynchronous writeback of data when a write request is in progress for the same block. Suppose that the protocol flow for the write request is at the stage when the page owner sends a write back request to the current block owner. If the block is written back by the block owner to the page owner <u>before</u> the write-back request is received (due to asynchronous replacement of the data), the message carrying the block back (*BLKBCK*) will not contain any information regarding the non-local write access. Additional data structures must be maintained in the page owner to handle these situations.

Our simulator does not support the functionality to trace individual non-local accesses. However, the simulator traces the progress of individual <u>messages</u> through the network to measure the transmission latency of messages through the network links and the network queuing delay encountered by messages. We use these measurements and the measurements we have taken to characterize the occurrence of protocol flows with different *hop-sequence* types (Section 5.1.6.3) to study the breakdown of remote access latency. This allows us to indirectly (and less precisely) break down remote access latency into message latency and table access latency components. We describe our methodology and results below.

During application execution, the simulator measures the link latency and queuing delay of each message that is exchanged. The link latency of a message measures the latency of the message from source to destination in the absence of any contention in the network. The link latency of a message is determined by the bandwidth of the links, the size of the message and the number of links traversed to reach the destination from the source. The queueing delay of a message measures the additional delay caused by buffering within the network during network contention. The overall latency incurred by a message is the sum of its link latency and queuing delay components. The simulator computes the average link latency and average queuing delay independently for control messages and block data messages. These averages are computed as the arithmetic mean of the measures collected for each individual message. Table 5.12 lists the average link latency and average queuing delay of control messages and block data messages measured by the simulator in simulation runs with network link bandwidth of 8P MB/s. The averages are listed for simulation runs with block sizes of 64 bytes and 256 bytes both for the one-level scheme and the hierarchical scheme.

In Tables 5.8 through 5.11 (Section 5.1.6.2), we have previously presented nominal cases of protocol flows for each of *hop-sequence* types 0 through 5. In each case, the tables identify the latency of tables accessed during the protocol flow. The tables also identify the messages that are exchanged during the protocol flow. We calculate the average link latency and average queuing delay of these messages using the average end-to-end link latency per message and the average end-to-end queuing delay per message measured (independently for control and block data messages) during the simulation. This allows us to identify the table access, message link latency and message queueing delay components of each *hop-sequence* type of protocol flow listed in Tables 5.8 through 5.11. As previously presented (Section 5.1.6.3), our simulator measures the number of occurrences of each *hop-sequence* type of protocol flow during the entire execution. The average cost breakdown of non-local accesses is calculated by computing the average of the costs of individual *hop-sequence* types weighted by their frequency of occurrence.

It should be noted that the breakdown of remote access latency into table access and network latencies is based on the tables accessed and messages exchanged in the specific protocol flows listed in Tables 5.8 through 5.11

Application	Scheme	Block Size (bytes)	Network Measures for Control Packets (in processor cycles)		Network Measures for Block Data Packets (in processor cycles)	
			Link Latency	Queuing Delay	Link Latency	Queuing Delay
Cholesky	Onelevel	64	8.561	2.429	16.923	7.539
		256	8.841	19.688	40.835	49.946
	Hierarchical	64	9.139	3.013	17.119	4.096
		256	9.056	17.659	41.102	44.590
LU	Onelevel	64	8.538	2.227	16.482	1.694
		256	8.584	16.344	40.528	9.633
	Hierarchical	64	7.939	1.321	15.837	0.812
		256	8.801	11.789	40.784	6.588
Radix	Onelevel	64	9.074	2.480	17.087	2.819
		256	9.066	17.416	41.105	17.290
	Hierarchical	64	9.063	7.892	17.062	6.627
		256	9.080	15.316	41.068	12.417

Table 5.12: Average Link Latency and Queuing Delay of Network Messages. The measures are listed from simulation runs with 8P MB/s links. The average link latency is the average latency of the message through the links in the absence of network contention. The average queuing delay is the average additional delay due to buffering in the network when there is network contention.

(Section 5.1.6.2). As we have mentioned earlier, alternative protocol flows are possible during application execution. These protocol flows can have a different *hop-sequence* type number than our example protocol flows. Even when protocol flows have the same *hop-sequence* type numbers as one of our example protocol flows, they can access different tables incurring a different total table access latency. Our analysis does not account for these alternative protocol flows. We report the measured average latency of remote accesses along with the results of our analysis. The inaccuracy in the analysis can be estimated by comparing the measured average remote access latency with the average remote access latency computed by our analysis.

Comparison of Cost of Hop-sequences of Protocol for Read Accesses				
Scheme	Type of hop-sequence of protocol	Total Table Access Latency (cycles)	Number of Control Messages	Number of Block Data Messages
One-level	Type 1	11	1	1
One-level	Type 2	54	2	1
One-level	Type 3	57	3	1
One-level	Type 4	100	4	1
One-level	Type 5	146	6	1
Two-level	Type 1	15	1	1
Two-level	Type 2	29	2	1
Two-level	Type 3	21	3	1
Two-level	Type 4	35	4	1
Two-level	Type 5	50	5	1

Table 5.13: Summary of cost of hop-sequences of protocol for read accesses.

For clarity, the cost components (table access latency, number of control messages and number of data messages) of each *hop-sequence* type of protocol flow are summarized in Table 5.13 and Table 5.14 (from Tables 5.8 through 5.11). Table 5.13 summarizes the costs of protocol flows for read accesses both on the one-level and on the hierarchical scheme. The message link latency for a protocol flow is the sum of the message link latencies for the control messages and the message link latencies for the block data messages exchanged in the *hop-sequence* type of the protocol flow. The message queuing delay in the protocol flow can be calculated similarly.

Comparison of Cost of Hop-sequences of Protocol for Write Accesses				
Scheme	Type of hop-sequence of protocol	Total Table Access Latency (cycles)	Number of Control Messages (c is number of copies)	Number of Block Data Messages
One-level	Type 0	22	2+2*(c-1)	0
One-level	Type 1	72	3+2*(c-1)	1
One-level	Type 2	86	4+2*(c-1)	2
Two-level	Type 0	22	2+2*(c-1)	0
Two-level	Type 1	36	3+2*(c-1)	1
Two-level	Type 2	57	4+2*(c-1)	2
Two-level	Type 3	42	5+2*(c-1)	1
Two-level	Type 4	63	6+2*(c-1)	2
Two-level	Type 5	66	7+2*(c-1)	2

Table 5.14: Summary of cost of hop-sequences of protocol for write accesses.

Table 5.14 summarizes the costs of protocol flows for write accesses on the one-level and hierarchical schemes. The average number of copies invalidated during a write access — ‘c’ in Table 5.14 — is measured during simulation. We estimate the cost associated with invalidating multiple copies ($c > 1$) based on a model in which the node sending the invalidate messages sends them in a pipelined manner and the nodes receiving invalidate requests process their invalidate concurrently. With this model, the total latency of invalidating multiple copies is the latency of invalidating the last copy. Also, the additional latency incurred by the write access due to the invalidation of multiple copies is the pipeline delay encountered by the last invalidate message at the node issuing the invalidations. We account for this additional delay by including a pipeline delay equal to the latency of one control message through one link for each extra (greater than one) copy that must be invalidated. We report this additional delay as part of the table access latency component of the write access protocol flow. The method of calculating the message link latency and message queuing delay for write protocol flows is the same as the method described for read protocol flows.

The methodology that we described above allows us to identify the contributions of coherency table access, network link latency and network queuing to the overall latency of remote accesses. As we have shown earlier, due to the multiple levels of ownership, protocol flows with higher *hop-sequence* type numbers are more likely in the hierarchically managed system. As described below, in order to account for the overhead associated with complicated protocol flows, we report separately the latency for the simplest remote accesses and the added latency for protocol flows with higher hop counts.

Using simple protocol flow for remote accesses, we define the base *hop-sequence* for read accesses and the base *hop-sequence* for write accesses. The cost overhead (in table access, in message link latency and message queuing) associated with complicated protocol flows can be estimated by comparison with the cost of this base *hop-sequence*. Our base *hop-sequences* are shown in Table 5.15. The base *hop-sequence* for read accesses is the same as the example *hop-sequence* of type 1 for reads in the one-level scheme (example in Table 5.8). The base *hop-*

sequence for write accesses is a *hop-sequence* of type 1 in the hierarchical scheme. In the protocol flow of the base *hop-sequence*, the default owner is also the page owner and the block owner. The default owner services the write request after invalidating all copies of the block. We use the same base *hop-sequences* in our analysis of protocol flows in both the one-level and hierarchical schemes to provide a common basis to compare the protocol flows in the two schemes.

Hop-sequence used as Base type in Access Latency Breakdown Analysis					
Type of Access	Nodes (Sequence)	Messages (Sequence)	Tables Accessed at Node	Table Access Latency (cycles)	Comments
Read	Requestor → Default Owner → Requestor	RREQ RDCPY	PPT PPT+SPT Cache	3 7 1	Same as One-level Type 1 hop-sequence 11 cycles table access; 1 control message, 1 block data message
Write	Requestor → Default Owner → Sharing Node → Default Owner → Requestor	WREQ INV ACK WDATA	PPT PPT+SPT+CoD PPT+SPT PPT+SPT Cache	3 14 7 7 1	Modified Two-level Type 1 hop-sequence 32 cycles table access; 3+2*(num of copies - 1) control messages, 1 block data message

Table 5.15: Hop-sequence used as base type in access latency breakdown analysis.

The base *hop-sequence* for read accesses incurs 11 cycles of table access latency and uses one control message and one data message. The base *hop-sequence* for write accesses incurs 32 cycles of table access latency. It uses one control message for the initial request and two control messages for each shared copy (invalidate and an acknowledgement) that is present during the write. One data message is used to return the data to the requester.

The following access scenarios illustrate the additional cost incurred by protocol flows with higher *hop-sequence* type numbers than the base *hop-sequences*. Consider the *hop-sequence* type 4 read access protocol on the hierarchical scheme shown in Table 5.9 (Section 5.1.6.2). The cost for this protocol flow is summarized in Table 5.13: 35 cycles of table access, 4 control messages and 1 data message. This protocol flow has an additional cost of 24 cycles of table access and 3 control messages over the cost of the base *hop-sequence* for read accesses. Similarly, consider the *hop-sequence* type 2 write access protocol flow on the one-level scheme illustrated in Table 5.10. This protocol flow has an additional cost of 54 cycles of table access, one control message and one data message over the cost of the base *hop-sequence* for write accesses.

Note that the protocol flow with *hop-sequence* type 0 for write accesses (Table 5.14) incurs lower table access latency and requires fewer messages than the base *hop-sequence* for write accesses. We account for the negative added overhead of this write protocol flow relative to the base write protocol flow by reducing the additional overhead due to protocol flows more complicated than the base protocol flow. The total negative overhead due to write protocol flows with *hop-sequence* type 0 is calculated from the number of occurrences of such flows during execution. The net overhead due to write protocol flows different than the base *hop-sequence* type is calculated by subtracting this total negative overhead from the additional overhead due to write protocol flows more complicated than the base protocol flow.

We have analyzed the breakdown of remote read and write access latency in all three applications for both the one-level and the hierarchical schemes. We

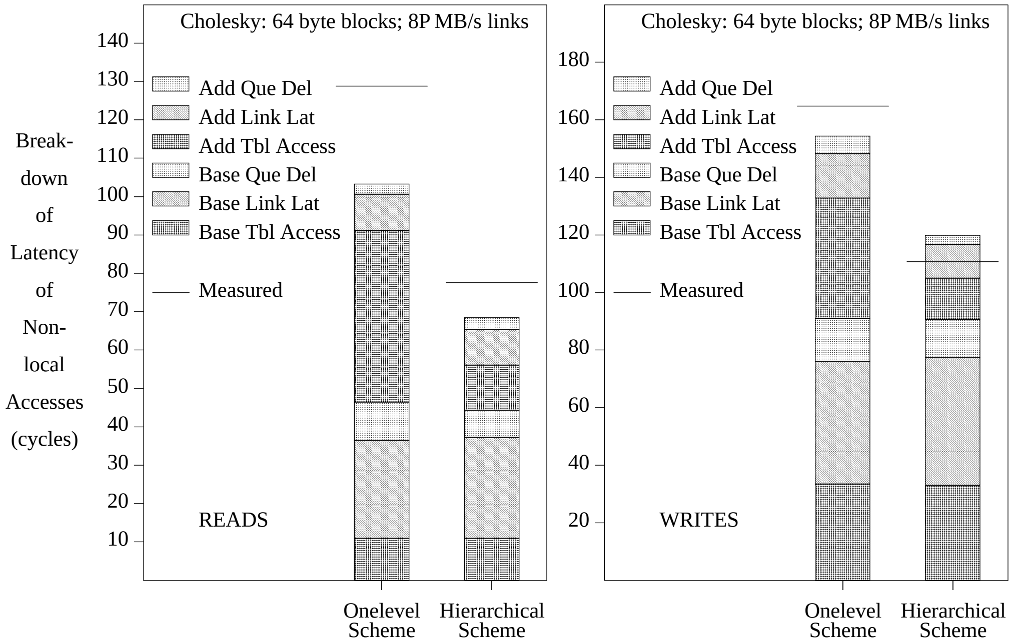

Figure 5.19: Access Latency Breakdown (Cholesky).

present and discuss the breakdown for simulation runs with a block size of 64 bytes. This block size represents a reasonable design point where our method of analysis is most meaningful. Our analysis is based on arithmetic means of the various latency components. As such, its accuracy is likely to be poor when the system behavior is bursty. With blocks significantly larger than 64 bytes, bursty system behavior is expected due to network traffic and hot spot contention for shared resources. Specifically, large blocks require large messages for moving blocks between nodes. Our simulations assume that messages cannot be partitioned into multiple packets. Hence, large blocks lead to large packets being transmitted across the network. Large packets occupy network links for long periods of time, blocking other packets which need the link. For small messages (e.g. control messages), the total latency through the network may be increased dramatically by the time spent waiting in queues for large packets to clear required links. As discussed earlier, larger blocks lead to higher false sharing. Thus, it is more likely that different processors contend for access to the same block (to access different shared variables) when large blocks are used. At various intervals during execution the owners of such blocks must service a large number of requests and can become hot spots [Pfis85] in the system. During those intervals, the latency of messages to the hot spots can be much higher than the average message latency. Furthermore, many nodes in the system may be blocked due to the limited rate at which requests are handled by the shared block owners.

Figure 5.19 shows the breakdown of remote read and write access latency in the Cholesky application. The breakdown is shown for both the one-level and hierarchical schemes (block size of 64 bytes). The height of the segments of the bar corresponding to the base table access, link latency and queuing delay (labeled ‘‘Base ..’’) show the latencies if all remote accesses in the execution were satisfied by protocol flows of the base *hop-sequence* type. The height of the other segments of the bar (labeled ‘‘Add ..’’) show the additional latencies due to protocol flows with *hop-sequence* different than the base. This is easily computed since the difference in cost of each *hop-sequence* from the cost of the base *hop-sequence* is known. The overall height of the bar shows the overall average remote access latency computed using our analysis. The horizontal line marks the average remote access latency that was directly measured in the simulation.

The discrepancy between the computed and measured average access latency is expected due to our indirect analysis. We have used the average message link latency and queuing delays measured by the simulator. Any burstiness in the network traffic during the simulation is not captured by these numbers. In addition, we have based our analysis on specific *hop-sequences*. As we have mentioned earlier, other *hop-sequences* as well as different table lookups for the *hop-sequences* are possible. The difference between the computed average access latency and the measured average access latency can be attributed to the differences in the protocol flows and to the dynamic variations in the network behavior.

Figure 5.19 shows that table access latency and the message link latency are the main contributors to the access latency (with 64 byte blocks). Although the hierarchical scheme uses protocol flows with higher *hop-sequence* type numbers (Section 5.1.6.3), the one-level scheme incurs a much higher cost in table access than the hierarchical scheme. This is primarily due to frequent access to the Migrated Block Table (MBT) in the one-level scheme. In the one-level scheme, the MBT must be updated on every write access and must be looked up at the default owner on many read accesses. The effect of MPT access on the latency of remote accesses in the one-level scheme may be largely artificial since the latency of accessing the MBT in our simulations (40 cycles) may be excessive. A realistic implementation of the one-level scheme must minimize access to the MBT as well as speed up accesses to the MBT. To perform a fair comparison, the remote access latencies on the one-level and the hierarchical schemes must be compared with an optimized MBT.

We have run additional simulations to study the one-level and hierarchical schemes when the latency of accessing the Migrated Block (Page) Table is only 4 cycles. Figure 5.20 shows the breakdown of remote access latency in the Cholesky

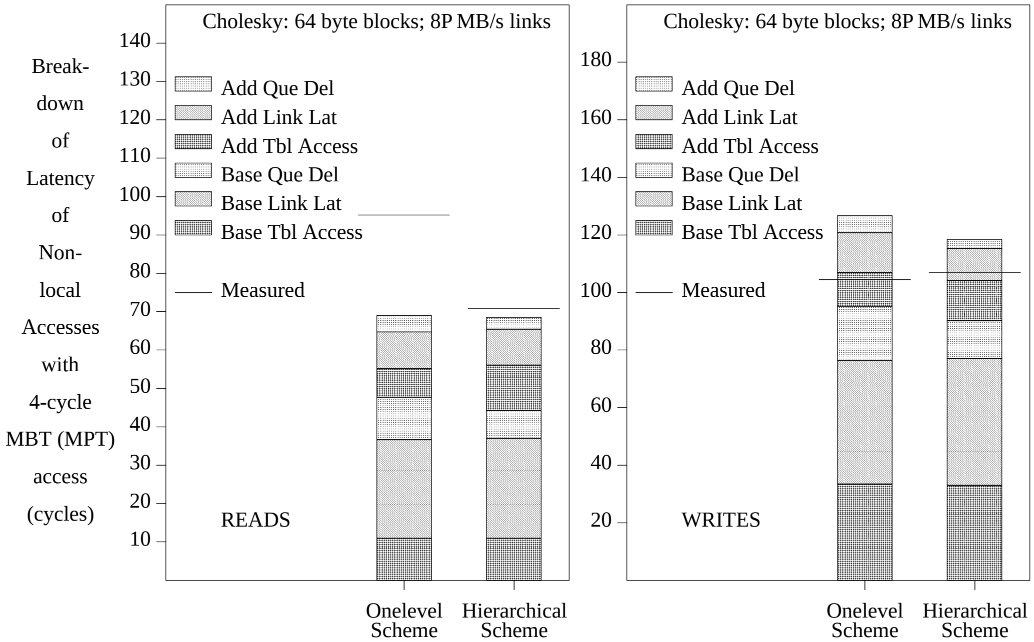

Figure 5.20: Access Latency Breakdown (Cholesky) with MBT access latency of 4 cycles.

application kernel with the optimized MBT (MPT) access for a block size of 64 bytes. As expected, the latency incurred in mapping table lookup in the one-level scheme is smaller when the MBT access latency is only 4 cycles. By comparison with the access latency breakdown shown in Figure 5.19 for an MBT access latency of 40 cycles, it can be seen that the measured and the computed overall latency of remote accesses decrease when the MBT access latency is reduced, both with the one-level scheme and with the hierarchical scheme. The benefit from an optimized MBT is more significant for the one-level scheme because the MBT is more frequently accessed in the one-level scheme. The figures also show that write accesses appear to benefit more from the optimized MBT than read accesses (the measured latency of remote write accesses drops by about 36% while the measured latency of remote read accesses drops by about 27%). This is expected since write accesses must always update the information in the MBT while read accesses may not always need to access the MBT.

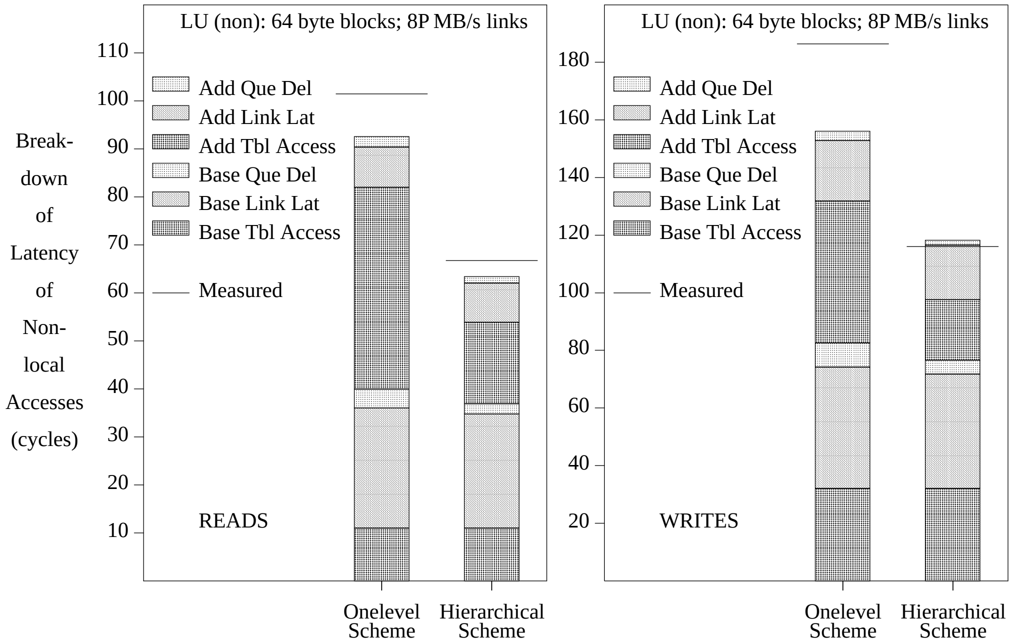

Figure 5.21: Access Latency Breakdown (LU).

Figure 5.21 shows the breakdown of the latency of read and write accesses in LU with the unoptimized MBT (MPT) access (40 cycles). The breakdown of remote access latency in LU has the same character as the breakdown of remote access latency in Cholesky. The latency incurred in accessing mapping tables and the message link latency constitute the major fraction of the remote access latency on both the one-level and hierarchical schemes. Also, the one-level scheme incurs significantly higher mapping table overhead due to the high cost of MBT access.

Figure 5.22 shows the breakdown of remote access latency in LU when the

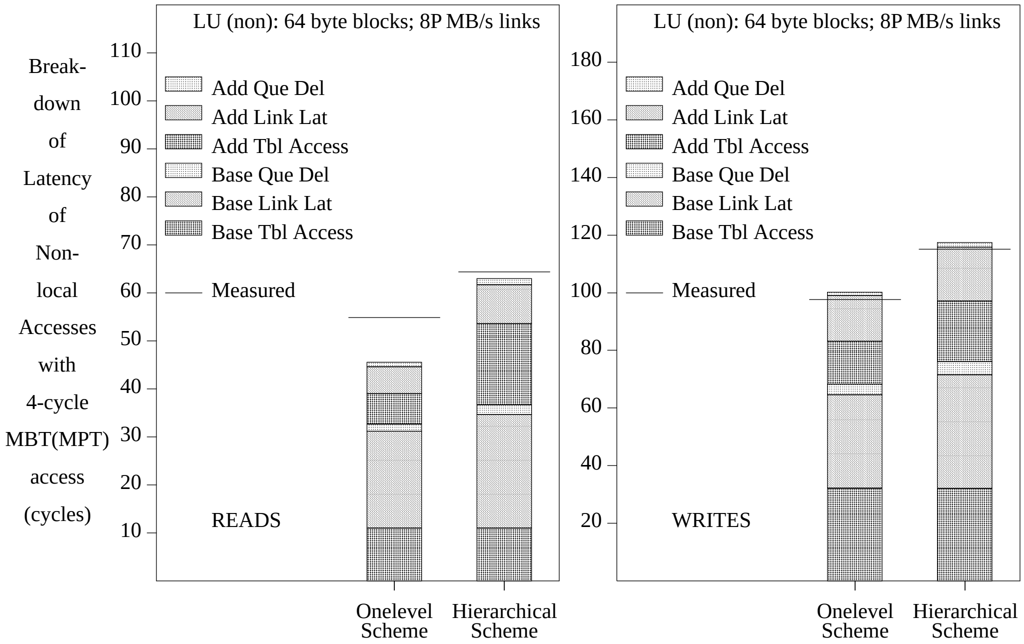

Figure 5.22: Access Latency Breakdown (LU) with MBT access latency of 4 cycles.

MBT (MPT) access has a latency of 4 cycles. Reducing the latency of MBT accesses significantly improves the performance of remote accesses on the one-level managed DSM system. The latency incurred in mapping table lookup in the one-level scheme is much smaller. As we mentioned earlier, the optimized MBT access benefits the one-level scheme much more than it benefits the hierarchical scheme. As a result, the average latency of remote accesses is lower in the one-level scheme than in the hierarchical scheme. The optimized MBT access benefits write and read accesses nearly equally (the measured latencies remote writes and remote reads drop by about 47%).

Figure 5.23 shows the breakdown of the latency of remote accesses in the Radix application when the MBT access latency is 40 cycles. In the Radix

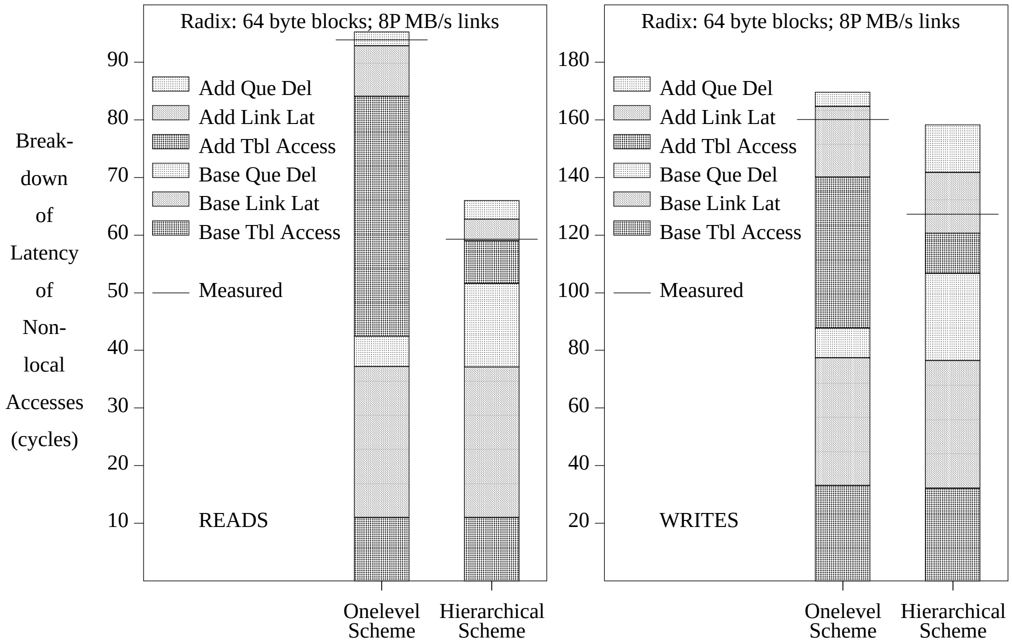

Figure 5.23: Access Latency Breakdown (Radix).

application, mapping table access latency and message link latency are the major components of remote access latency in the one-level scheme. However, the message queueing delay appears to be a significant part of the access latency in the hierarchical scheme. As we have mentioned earlier (in Section 5.1.6.1), in the Radix sorting application, processes may need to access data from disk when the keys being sorted are written to their new locations after each phase of sorting. With the hierarchical scheme, the data is accessed from disk in large pages. Since all processes concurrently attempt to update the array locations and also contend for synchronization locks, the large page-size messages can increase contention and queuing delay in the network. The latency of accessing mapping tables is higher in the one-level scheme than on the hierarchical scheme due to the long latency of MBT accesses, as was the case with the other two applications.

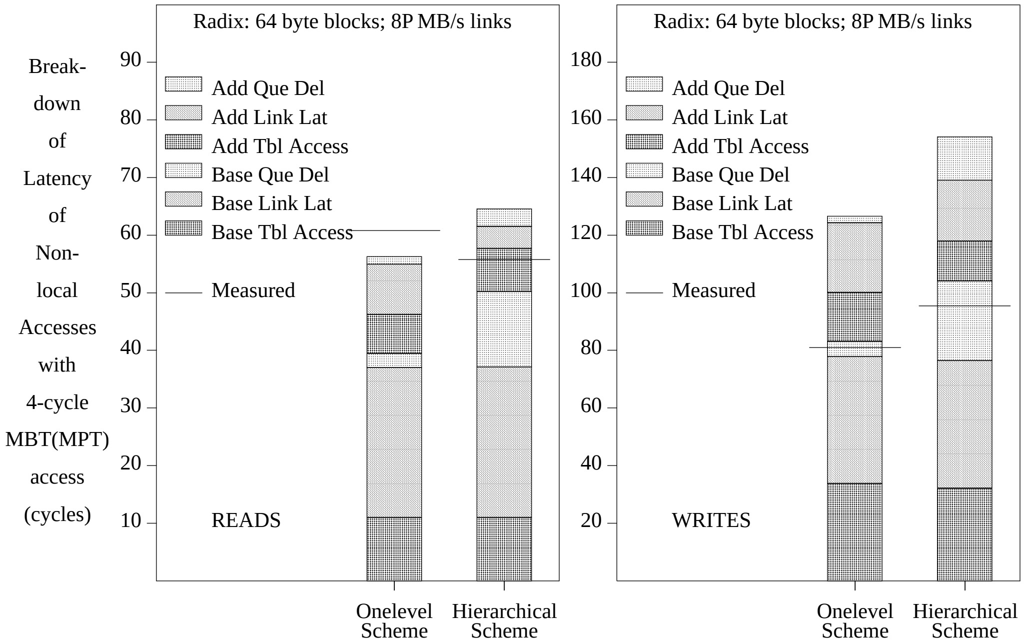

Figure 5.24: Access Latency Breakdown (Radix) with MBT access latency of 4 cycles.

When the MBT (MPT) access latency is reduced to 4 cycles, the latency incurred in mapping table accesses and the overall remote access latency (shown in Figure 5.24) improve significantly. The impact of the optimized MBT access is the same in Radix as in the other two applications: the one-level scheme benefits more than the hierarchical scheme and write accesses benefit more than read accesses. The measured latency of remote writes drops by about 50% while the measured latency of remote writes drops by about 37%.

In Figure 5.24, it can be seen that our analysis overestimated the average latency of a write access (difference between the measured and computed latencies) by over 40 cycles, for an error of over 50%. In Radix, processes concurrently attempt to update the array locations and acquire synchronization locks after each phase of sorting. Hence, network traffic is likely to be bursty and some shared blocks (blocks with locks and blocks which map array locations modified by different processors) are likely to be hot spots. These problems are likely to be aggravated in the case of the hierarchical scheme due to page-sized transfers from disk. As we discussed earlier, such burstiness and hot spots can affect the accuracy of our breakdown analysis.

5.1.7. Profile of Execution Time

During application execution, a processor can be in one of three states. The processor can be executing instructions, can be performing an access to the shared address space, or can be idle waiting for some synchronization event. A processor's access to the shared address space can be satisfied in several ways: locally (hits), by accessing the data from disk, or by obtaining the data from other nodes in the system. In order to understand the behavior of the system and the applications, it is useful to measure the fraction of application execution time processors spend in each of the three states described above. It is also important to identify the contributions to the average shared access time from hits, disk accesses and remote accesses.

In the following discussion, we analyze application execution by characterizing the behavior of an ‘‘average’’ processor, by assuming that all processors behave uniformly. However, it must be noted that, as we discussed previously in Section 5.1.2, the behavior of processors is not uniform in all applications (Figures 5.4 through 5.6). With the Cholesky application, all processors in the system execute nearly the same number of instructions and total accesses to shared data. The distribution of instructions and shared read accesses amongst the processors is less uniform in LU. In the case of Radix, instructions and shared accesses are unevenly distributed amongst the processors due to the uneven distribution of workload in the application.

The simulator measures the fraction of cycles that each processor spends executing instructions and the fraction of cycles during which each processor is idle. The behavior of the ‘‘average’’ processor is characterized by computing the arithmetic mean of these quantities over all the processors in the system (assuming the behavior of processors is uniform). The total number of cycles that the ‘‘average’’ processor spends executing instructions is calculated as the product of the fraction of cycles the ‘‘average’’ processor spends executing instructions and the measured application execution time (in cycles). The total number of cycles during which the ‘‘average’’ processor is idle is calculated similarly.

The time spent by the processor accessing shared space is characterized using several other measures taken during simulation: the total number of shared address space accesses that were hits, the average latency of such hits, and the total number of accesses to disk. The total number of cycles spent by all processors in shared access hits is the product of the total number of shared access hits and the measured average latency of a shared access hit. The total number of cycles spent by all processors in disk accesses is computed as the product of the total number of disk accesses and the disk access latency (1000 cycles). The total number of cycles spent by the ‘‘average’’ processor in shared access hits is calculated (assuming all processors behave uniformly) as the ratio of the total number of cycles spent by all processors in shared access hits to the total number of processors in the system

(64). Similarly, the number of cycles spent by the ‘‘average’’ processor in disk accesses is calculated (assuming all processors behave uniformly) as the ratio of the total number of cycles spent by all processors in disk accesses to the total number of processors in the system (64).

We use the measurements taken to profile the latency of non-local accesses (Section 5.1.6.4) to estimate the number of cycles spent performing non-local shared address space accesses. The total number of cycles spent in non-local shared accesses is calculated from the occurrence counts of individual *hop-sequence* types and their individual component costs (calculated using the network performance measures as discussed in Section 5.1.6.4). We again assume that the total number of cycles spent in non-local shared address space accesses is uniformly distributed amongst all processors.

In Figure 5.25, we show the breakdown of execution cycles in the application kernel LU. The figure shows the profile of execution time for two different block sizes (64 bytes and 256 bytes) for both the one-level and hierarchical schemes. The time spent in non-local shared accesses is broken down into its table access, message link latency and message queuing components along the lines discussed earlier in Section 5.1.6.4. The horizontal line marks the actual number of application execution cycles measured by the simulation. The measured and computed execution times differ due to the indirect computation of the time spent in non-local shared accesses (see Section 5.1.6.4).

With both the one-level and hierarchical schemes, at the higher block size (256 bytes), more time is spent in sending messages through the network (message link latency and message queuing delay components are higher). With larger

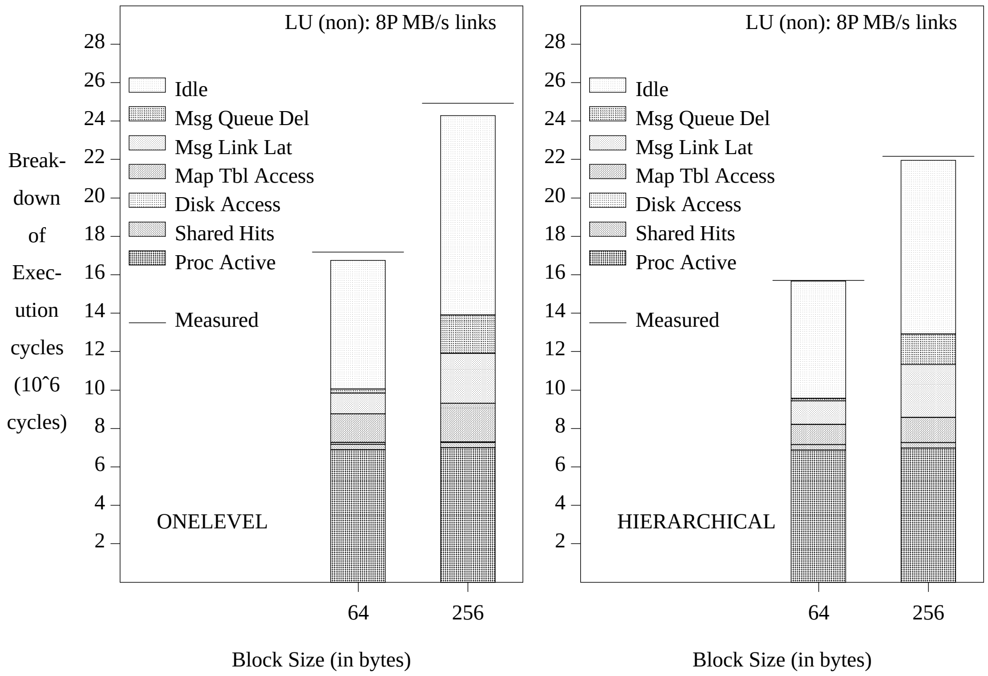

Figure 5.25: Execution cycles breakdown (LU).

blocks, block data messages are larger. For a given link bandwidth, larger block data messages incur higher link latency. Hence, the average link latency of messages is higher at larger block sizes. Larger block data messages also occupy network links for longer duration which increases queuing delays in the network. Due to these effects, the total time spent in remote accesses (sum of the ‘‘Map Tbl Access’’, ‘‘Msg Link Lat’’ and ‘‘Msg Queue Del’’ components) is higher at larger block sizes.

Larger blocks can also increase contention for shared data due to higher false sharing. Since the contention for network bandwidth and shared data is higher with larger blocks, processes that acquire these resources first can complete their task much earlier than processes that acquire these resources later. As a result, with larger blocks, processes that complete their task early may need to remain idle for longer length of time waiting for the later processes to complete their task (at a barrier synchronization, for example). This can increase the processor idle time when blocks are larger (Figure 5.25). The increase in idle time and the time spent in remote accesses lead to higher overall application execution time and lower processor utilization at larger block sizes.

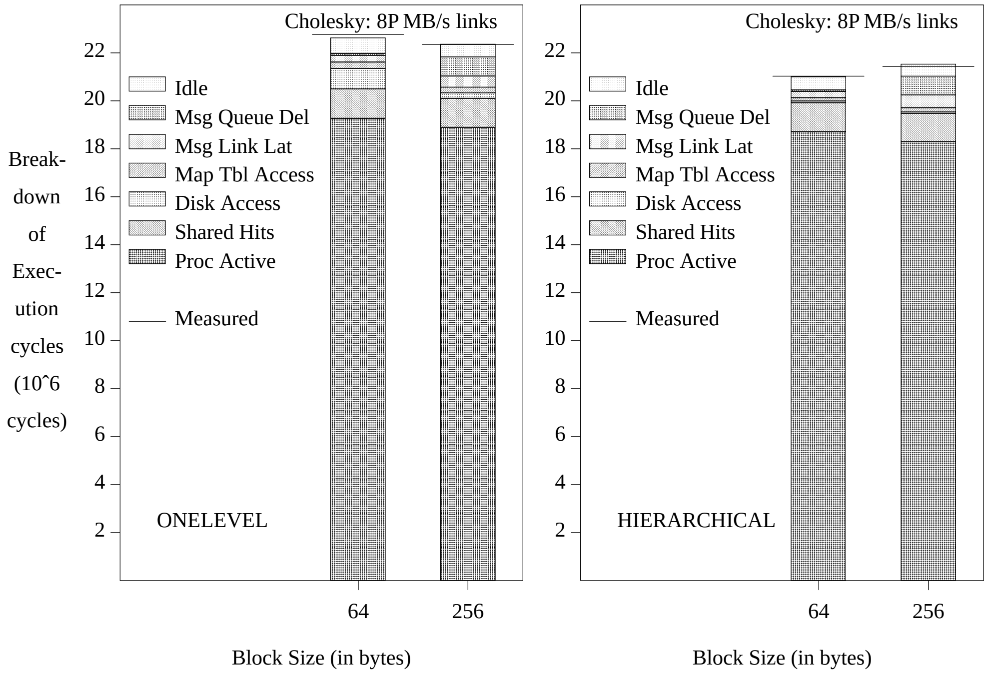

Figure 5.26: Execution cycles breakdown (Cholesky).

Figure 5.26 shows the breakdown of execution cycles in the Cholesky application. As discussed earlier, the message link latency and message queueing delay components are higher at the larger block size (256 bytes). However, the figure shows that the overall execution time of the Cholesky application was higher at the smaller block size (64 bytes) in the one-level scheme. It can be noted from the figure that this is because more time is spent in disk accesses (the segment of the bar labeled ‘‘Disk Access’’) at the small block size. As we have discussed earlier (Section 5.1), this is an artifact in our implementation where disk accesses in the one-level scheme are at the granularity of blocks. With smaller blocks, more disk accesses are needed to fetch the data from disk increasing the overall time spent in disk accesses.

It is interesting to compare the behavior of Cholesky (Figure 5.26) with the behavior of LU (Figure 5.25). The time spent in shared hits is significantly higher in Cholesky than in LU. This matches our earlier results on the miss rate of shared accesses (see Figure 5.14 in Section 5.1.6.1) where Cholesky demonstrated a higher hit rate than LU. Since the Cholesky application has a higher hit rate for shared accesses than LU, the time spent in remote accesses is lower. As a result, processor utilization is higher in Cholesky.

The breakdown of execution cycles in Radix is shown in Figure 5.27. Radix demonstrates behavior similar to that shown by LU and Cholesky. The message link latency and message queuing delay are higher at the larger block size (256 bytes) increasing the time spent in remote accesses. As we discussed with Cholesky, more time is spent in accessing disks with 64-byte blocks than with 256-byte blocks in the one-level scheme due to the fact that we implement disk accesses at the granularity of blocks in the one-level scheme. With a block size of 64 bytes, the message link latency and message queuing delay components of execution time are higher in the hierarchical scheme than in the one-level scheme. As we explained earlier in Section 5.1.6.4, this is most likely due to the large page-sized transfers from disk in the hierarchical scheme.

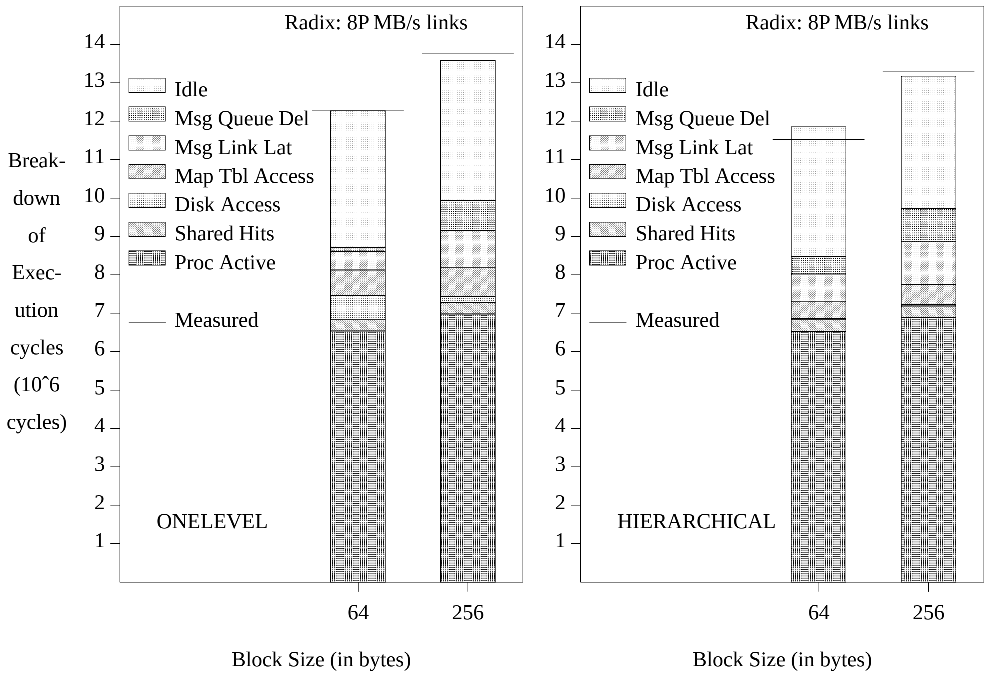

Figure 5.27: Execution cycles breakdown (Radix).

5.2. Evaluation based on Trace-driven Simulation

Our initial evaluations of the hierarchical scheme were based on our trace-driven simulator. As we described in Section 4.4.1, our simple trace-driven simulator does not model the latencies associated with the network or table lookups. Hence, all coherence transactions are completed atomically.

The address traces used to drive our trace-driven simulations were obtained from the simulation of another 64 node multiprocessor architecture on a single processor system. The address traces represent possible executions of three parallel applications: Weather, Speech, and FFT. The Weather application partitions the atmosphere into a three dimensional grid and uses finite difference methods to solve a set of partial differential equations describing the state of the system. The Speech application comprises the lexical decoding stage of a phonetically-based spoken language understanding system. The application uses a modified Viterbi search algorithm to find the best match between paths through a directed graph representing a dictionary and another directed graph representing the input. FFT is a radix-2 fast Fourier transform.

The basic characteristics of the traces are summarized in Table 5.16. The number of instruction references, data references and shared data references in the trace are listed in the table. We performed a static analysis of the trace (without executing the simulator) to identify the size of the shared address space. The total number of 1 KB pages that were shared in the trace was measured and is reported in Table 5.16. More details about the applications and the traces can be found in[Chai90].

Application	Trace length (10^6 refs)	Number of Instruction refs (10^6 refs)	Number of Data refs (10^6 refs)	Number of Shared data refs (10^6 refs)	Max data foot print (KB) per processor	Shared foot print (KB)	
						Average per processor	of system
Weather	31.76	13.64	18.12	5.02	5346	5192	5330
Speech	11.77	10.78	11.77	11.77	510	434	596
FFT	7.44	3.11	4.33	1.05	164	130	130

Table 5.16: Trace characteristics. The system shared footprint is the number of 1 KB pages accessed by more than one processor during the execution of the application. The per-processor shared footprint is the average number of pages, out of the system shared footprint, that were accessed by individual processors.

In our simulations the size of the Present Pages Table (PPT) is chosen to accommodate most of the process footprint to ensure that the number of misses due to PPT overflows will be negligible. Specifically, the PPT size for Weather, Speech, and FFT, was 2048, 1024, and 256 pages, respectively. When there are no free entries in the PPT, LRU replacement is used to evict pages. The number of entries in the Shared Pages Table (SPT) and the number of entries in the copy directory were varied in the simulations. The replacement mechanism for the SPT is biased to evict pages that are not owned locally (to minimize traffic). Pseudo-random replacement is used to pick a victim copy directory entry when eviction is necessary.

The simulator determines the state of the PPT, SPT, and the copy directory after each memory reference and builds statistics on miss ratios, network traffic, and the number of SPT and PPT replacements. Simulations were performed for both 1 KB and 4 KB pages for a range of block sizes, SPT sizes and copy directory sizes.

The miss ratio on accesses to the data address space and the total network traffic are good indicators of the potential system performance and are used to present the results of the simulation. The network traffic is calculated assuming that each control message is 16 bytes long, while the size of data messages (Block or Page) is 16 bytes plus the data being transmitted.

Figures 5.28 and 5.29 show the data access miss ratios, for all three applications with various SPT and block sizes. The size of a page is 4 KB in Figure 5.28 and the page size is 1 KB in Figure 5.29. The corresponding network traffic is shown in Figures 5.30 and 5.31. The advantage of a small block size is demonstrated by the decrease in network traffic with decreasing block size. For Speech and FFT, decreasing block size also results in a decrease in false sharing,

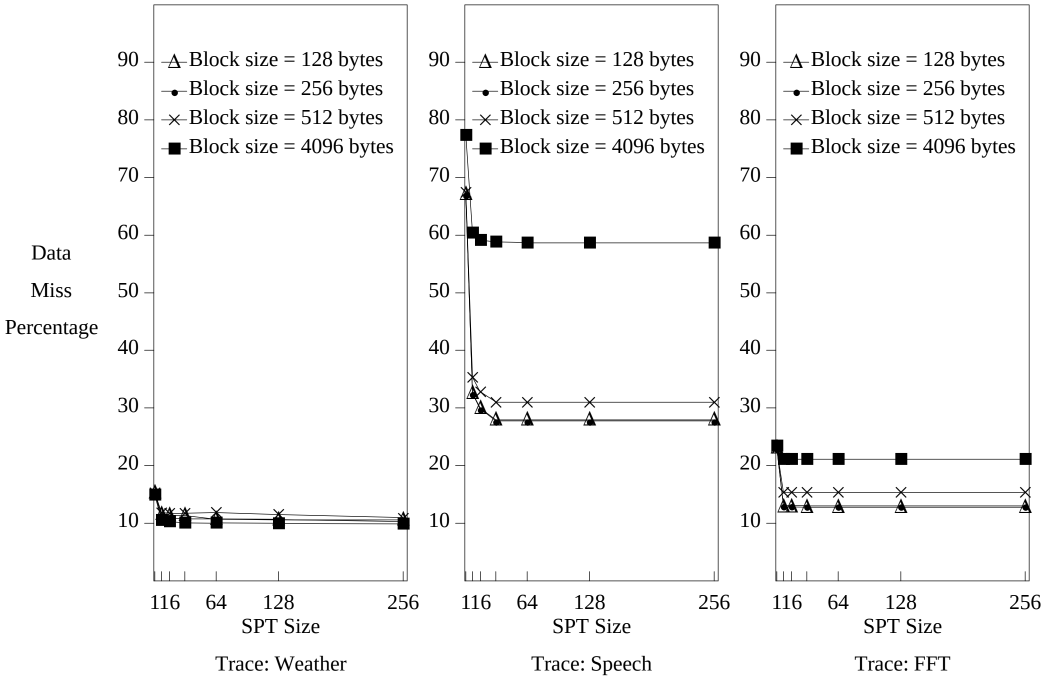

Figure 5.28: Data miss percentage versus SPT size. 4KB pages. 32 BMpP copy directory entries.

leading to a decrease in miss ratio.

An important result of the simulations is that, with the hierarchical scheme, performance is dependent on block size rather than page size. Specifically, for a page-level scheme (block size equal page size), 4 KB pages lead to significantly higher miss ratios and network traffic than 1 KB pages (Figures 5.30 and 5.31). However, for equal block size (128 bytes), performance of the system with 4 KB pages is almost identical to the system with 1 KB pages. Since larger pages result in smaller tables, this is a clear demonstration of the advantage of our multi-level coherency scheme.

For very small SPTs, significant reductions in miss ratio and network traffic

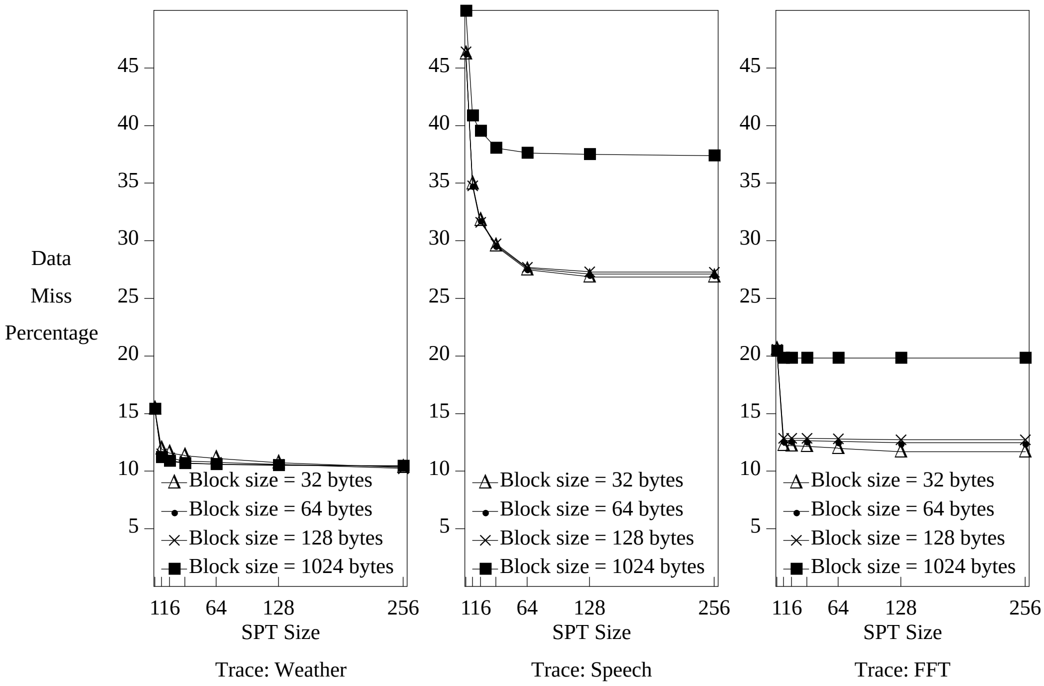

Figure 5.29: Data miss percentage versus SPT size. 1KB pages. 32 BMpP copy directory entries.

are obtained by increasing the SPT size. However, once the knee of the curves is reached (in this case, at around 32 entries), further increases in the SPT size do not provide significant performance improvement. These results confirm that a small SPT (about 10% of the total working set) would be capable of accommodating the ‘‘shared working set’’ and provide good performance. The knee occurs at a higher SPT size for 1 KB pages than for 4 KB pages. This indicates that, for these applications, in our two-level scheme, for a fixed block size, a smaller page size will actually *reduce* performance unless a larger SPT is used.

Figure 5.32 shows the miss ratio and network traffic for various sizes of the Block Map per Processor (BMpP) copy directory. As expected, performance is

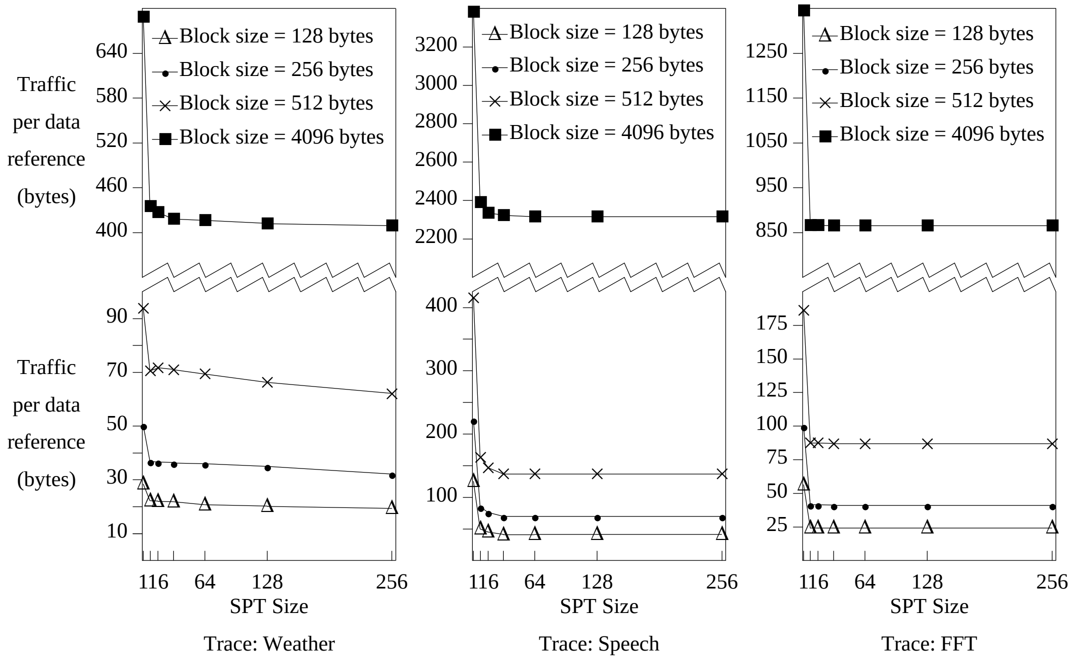

Figure 5.30: Network traffic versus SPT size. 4KB pages. 32 BMpP copy directory entries.

reduced if the directory is too small. For Weather and FFT, the reduction in miss ratio and network traffic with increasing directory size levels off for a BMpP copy directory with 32 entries or more. This agrees with the expectation, based on the access burst model (Section 3.1.2 in Chapter 3), that there is little performance advantage in maintaining full copy directories for shared blocks. The different behavior for Speech is due to the high read-only sharing of a dictionary. A policy of eviction, such as we have employed, will fair poorly in the presence of significant read sharing by a large number of processors. Better support for read-shared pages can be provided with a broadcast policy [Agar88], with a policy that allows a small, dynamically determined, set of pages to have larger copy tables [O'Kr90], or with support for maintaining full copy directories in

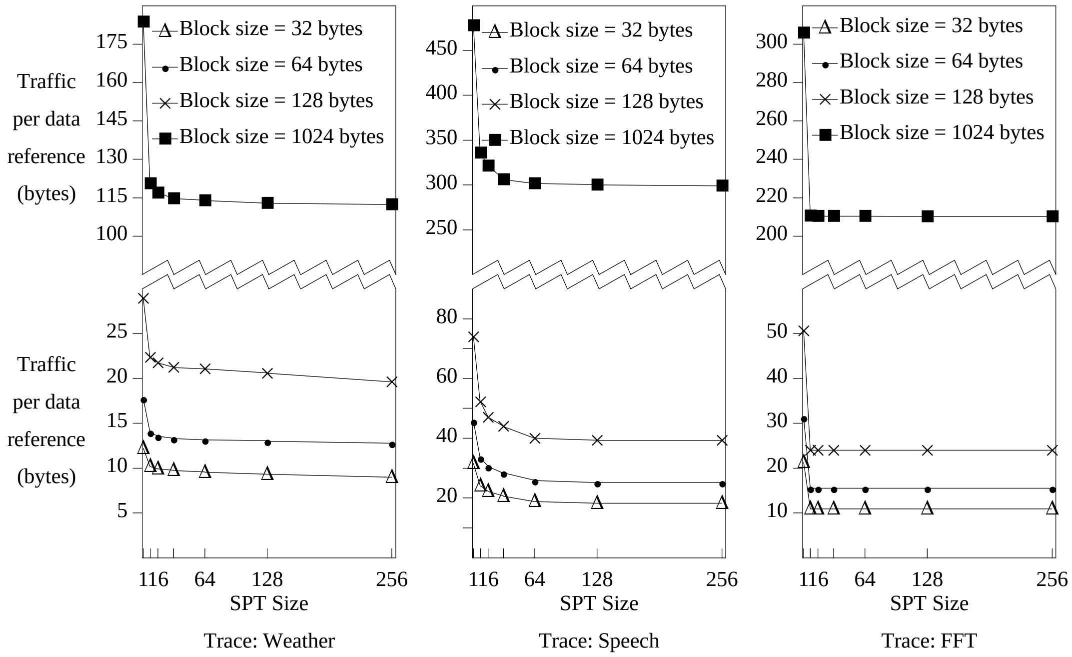

Figure 5.31: Network traffic versus SPT size. 1KB pages. 32 BMpP copy directory entries.

software [Chai91].

Figure 5.33 compares the performance of the Block Map per Processor (BMpP) and the Processor List per Block (PLpB) copy directory organizations. The graphs show the data miss percentage and the network traffic as a function of the size of the copy directory per shared page, for both organizations. The data points for the BMpP organization correspond to 8, 16, 32 and 64 entries per page. The data points for the PLpB organization correspond to 4, 8, 16 and 32 entries per block. As mentioned earlier, the storage overhead of the BMpP organization is smaller than the overhead of the PLpB organization. Assuming that table entries are byte aligned in memory, a BMpP copy directory with 32 entries requires 192 bytes per page. With the PLpB directory organization, 256 bytes per page support

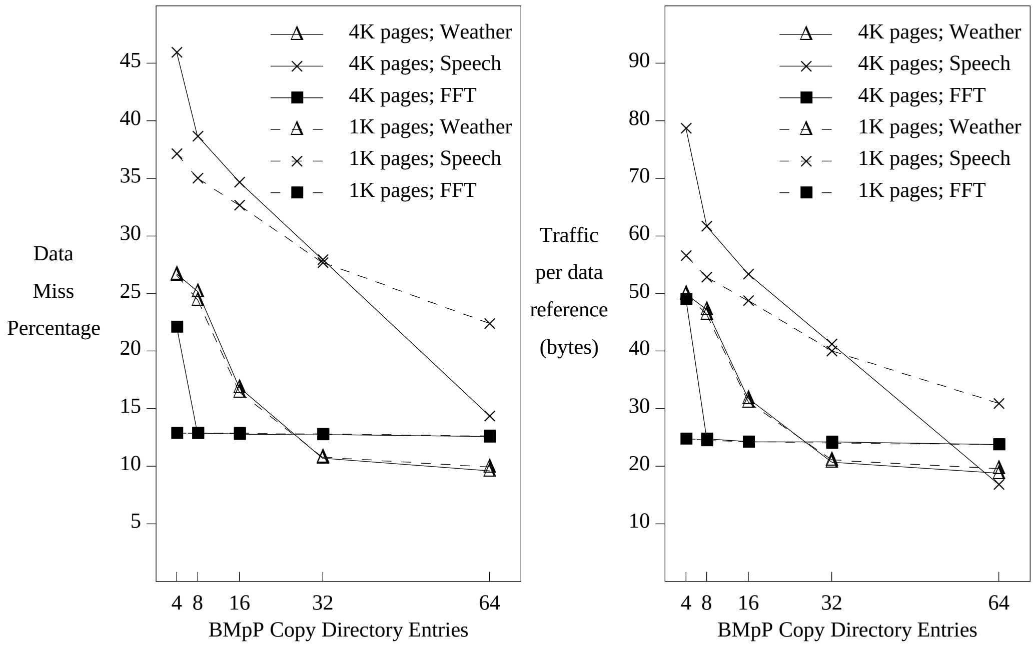

Figure 5.32: Data miss percentage and network traffic versus number of entries in BMpP copy directory for page sizes 1KB and 4KB. SPT size is 64. Block size is 128 bytes.

directories with four entries per block. In our simulations, for comparable performance, the copy directory in the PLpB organization had to be several times larger than in the BMpP organization.

Since our trace-driven simulator models accesses to the shared address space atomically and is based on address traces that provide no timing information, the results from our trace-driven simulations do not provide any estimate of application execution time. The improvement in data miss percentage and network traffic per data reference with the hierarchical scheme may not directly translate into improvement in overall application performance. However, they provide sufficient validation of the basic ideas of the hierarchical scheme to justify detailed evaluation of the hierarchical scheme. As we have described earlier (in

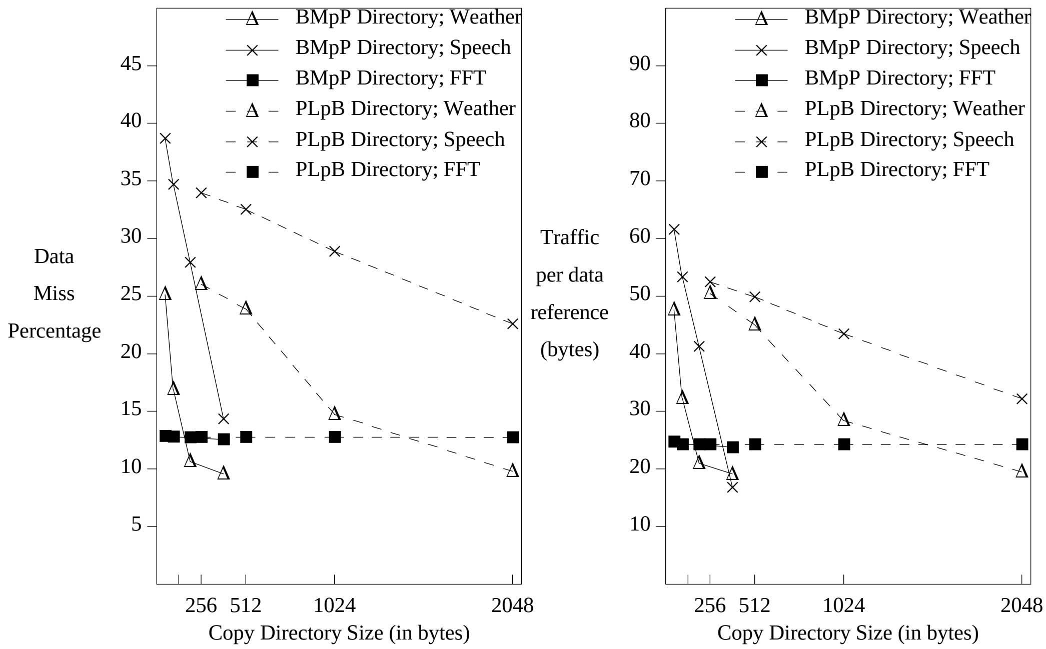

Figure 5.33: Data miss percentage and network traffic versus copy directory size per page for BMpP and PLpB organizations. Page size is 4KB. SPT size is 64. Block size is 128 bytes.

Section 5.1), we have evaluated the hierarchical scheme in more detail using execution-driven simulations.

Our measurement of the shared working set size of applications with the execution-driven simulation (Section 5.1.3) confirm the results about shared working set size from our trace-driven simulations. The results from both our evaluations (Figure 5.7 from execution-driven simulations and Figure 5.31 from trace-driven simulations) indicate that a Shared Pages Table (SPT) which has much fewer entries than the PPT can be used with negligible impact on application performance.

The evaluation based on trace-driven simulation has demonstrated that network traffic is very high with page granularity one-level schemes and decreases with decreasing block size (Figure 5.31). Our execution-driven simulations verify that remote access latency (Figure 5.12) and application execution time (Figure 5.10) are both very high with page granularity one-level schemes. The figures also show that the decrease in network traffic with decreasing block size leads to lower latency for remote accesses and lower application execution time at smaller block sizes. At small block sizes, the overall time spent by the application in performing remote accesses is smaller (Section 5.1.7) leading to higher processor utilization.

We have studied the size of the copy directory and the effectiveness of our Block Map per Processor (BMpP) copy directory organization using both our trace-driven and execution-driven simulations. Our study based on trace-driven simulations was conducted by varying the size of the copy directory. In our study using execution-driven simulations, we have measured the average size of the copy directory used by the applications. Both studies indicate (Figure 5.9 and Figure 5.32) that copy directories with few entries are typically sufficient. However, in our study using trace-driven simulations, we have observed (Figure 5.32) that small copy directories may be inadequate when there is high read-only sharing of data. The applications we have used in our execution-driven simulations do not demonstrate significant read-only sharing. Figure 5.8 (from execution-driven simulation) and Figure 5.33 (from trace-driven simulation) both agree that the BMpP copy directory organization maintains the copy directory information more efficiently than the Processor List per Block (PLpB) organization).