Binding Time in Distrubuted Shared Memories

A THESIS
SUBMITTED TO THE FACULTY OF THE GRADUATE SCHOOL
OF THE UNIVERSITY OF MINNESOTA
BY

Jinseok Kong

IN PARTIAL FULFILLMENT OF THE REQUIREMENTS
FOR THE DEGREE OF
DOCTOR OF PHILOSOPHY

Pen-Chung Yew and Gyungho Lee, Adviser(s)

June, 1999

UNIVERSITY OF MINNESOTA

This is to certify that we have examined this copy of a doctoral thesis by

Jinseok Kong

and have found that it is complete and satisfactory in all respects,
and that any and all revisions required by the final
examining committee have been made.

Pen-Chung Yew and Gyungho Lee

Name of Faculty Adviser(s)

Signature of Faculty Adviser(s)

Date

GRADUATE SCHOOL

Abstract

Distributed shared memory (DSM) architectures try to bind recently referenced data to memory local to a node with the assumption that the data will be referenced again in the near future. However, patterns of reference at the local memory may not be as highly localized as at the processor cache. In general, the higher the cache hit rate, the lower the local memory hit rate. Also, unlike in traditional caches, replaced data from the local memory can have the side effect of affecting locality of data in a remote memory. Thus, utilizing locality of reference at the local memory seems to be arduous in DSM architectures due to higher overhead for dynamic binding and weaker locality of reference.

This thesis considers three binding times for new data at the local memory: page fault time, node miss time, and cache miss time. In DSM architectures, different binding times are associated with different ways of arranging data, which cause different overheads under the same patterns of reference. Since a large number of cache misses can be found in a large (relative to the cache size) working set, binding only at page fault time cannot efficiently utilize locality of reference at the local memory. In a small working set, binding missing data to the local memory at node miss time is not effective due to high cache hit rates. This thesis shows that, at cache miss time, binding the data replaced from the cache to the local memory can improve performance under a wide range of reference patterns.

Acknowledgements

First of all, I am very grateful to my adviser Gyungho Lee, for giving me a chance to study memory architectures. Without his insight of memory architectures, I believe, this work would not even be started. Also, I am greatly indebted to my adviser Pen-Chung Yew for his very helpful guidance of researches in this field. Indeed, this dissertation could not have been completed without his assistance. I would like to thank Professor David Lilja, Professor Matthew O'Keefe, Professor Larry Kinney, and Professor Zhiyuan Li for serving on my oral examination committee. I would like to especially express my gratitude to Professor David Lilja for serving on my reading committee and for his valuable comments on my research.

I owe many thanks to the HPCC group, the former DICE members, and office mates: Chris Amlo, Manu Agarwal, Tong Chen, Sangyeun Cho, Youngsoo Choi, Bob Glamm, Junjie Gu, Chris Hescot, Jian Huang, Sujat Jamil, Liang Jun, Gungok Jung, Iffat Kazi, Junseong Kim, Hock-Beng Lim, Jenn-Yuan Tsai, Steve VanderWiel, Qing Zhao, Yan Zhen, and Bixia Zheng, who have made my stay at Minnesota very worthwhile and most enjoyable. Especially thanks to Sangyeun Cho, Steve VanderWiel, and Professor Lynn Choi for their discussions, suggestions, and encouragement. I owe thanks to more of them than I could list here, especially to the anonymous referees for their valuable comments on the early conference papers.

Finally, I dedicate my work to my parents.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

When a processor makes a reference to memory, there are two necessary conditions: data *integrity* and data *consistency*. The data referenced by the processor must exist in the memory and be supplied to the processor to satisfy the first condition, data integrity. Otherwise, the processor will not operate on the referenced data. Also, the supplied data must be valid for data consistency. Otherwise, the result of the processor will be incorrect. The next consideration is to speed up the execution of programs. For general computation, a computer must be able to multitask among a myriad of programs of varying sizes, ranging from rather small to quite large. This suggests a large amount of memory in the computer. However, memory access speed slows down as memory size grows. Thus, we try to exploit some program properties to speed up program execution on general-purpose computers [DCC+87, Jor83, LLG+90, PC90]. A well-known heuristic property is *locality* of reference in a program: Data that has been referenced recently or data that is near recently referenced data will tend to be referenced soon. Another intrinsic program property is *parallelism* in data processing. Parallelism exists at different levels of data processing such as bit level, instruction level, thread level, program level, etc. We will see throughout the thesis how these two program properties can be exploited in the memory system to achieve high system performance.

Data access latency is the time between when a processor issues a read or write request for data in memory and when the request is completed. For completion of a read request, memory must supply the requested data to the processor. For completion of a write request, memory must update the requested data in memory. The processor can proceed without waiting for the completion of a request if the request is a write or if the following instructions do not depend on the requested data. Thus, the data access can be overlapped with the processor's execution. However, most instructions do rely on data supplied from memory, and thus, a memory system has to have a small data access latency to achieve high performance. Since processors issue many memory references during execution of a program, reduction of accumulated data access latency is more important than reduction of specific data access latency. If there are several physical locations of memory, data access latency will vary with the relative distances of the memory locations from the processor. Also, patterns of reference in a program are changed dynamically over time. Therefore, *how to arrange data in the memory locations at a given instant of time* is a critical factor that affects system performance.

Every datum has a name, called its address. When an address is found missing at a memory location, we have two choices; binding the address to the location or not. *Binding time* for the location is the time when the new address is assigned to the location. At the same time, an existing address may be unbound due to limited space. For data integrity, the unbound address may need to be bound to some other memory location. This may in turn induce another displacement, leading to a new arrangement of data in the memory locations. Thus, a binding is the smallest unit of events that causes a new arrangement of data in the memory locations. Generally, a valid datum is saved in the location at the binding time. The datum at the location could be updated or invalidated before its address is unbound from the location. For data consistency, there must exist at least one valid datum for each address in an address space.

For ease of programming, several different names can be used to refer to the same data (i.e., aliasing). For high performance, one or more names for the same data may need to be bound to several memory locations. If necessary, a memory architecture can implicitly create new names for data (data renaming). These new names are not visible to compilers or programmers. Address translation from virtual to physical address space is one example of data renaming. Another example is register renaming in register files.

Figure 1.1 shows an example of how a computer binds data to memory locations to compute a program. The program in Figure 1.1 (a) has a name, x, which refers to value 3. There is another value, 20, whose name is not explicitly specified. The program adds the two values, and the result is named x again. The computer in Figure 1.1 (b) has one processing unit, ALU, and three memory locations, A, B, and C. To execute the program, the value 3 and 20 are assumed to be in locations A and B respectively. That is, name x is bound to location A and has a valid value. At the next time, the sum of the two values is saved in location C. Now, there are two locations for the name x, locations A and C. However, the value in location A is now invalid. Thus, a new value can be overwritten on location A. However, since the value in location C is valid, we have to relocate that value to a different location before writing a new value on location C.

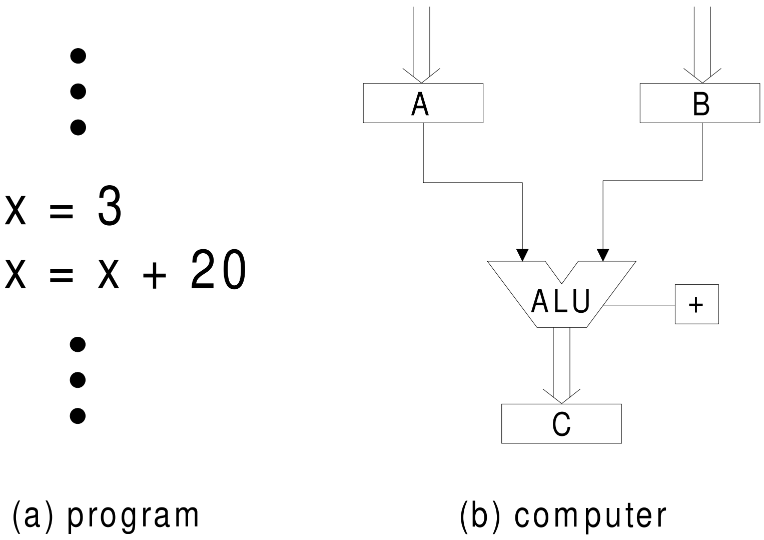


Figure 1.1: Program execution on a computer

A memory system has three types of *overhead*: overhead for accessing memory locations; overhead for moving data through the datapath; and overhead for controlling memory accesses and movements of data on the datapath. Each binding may require at least two memory accesses (copying the data from a source location and saving the data in a target location) and one use of the datapath. To change the state of data in a location (assuming the location also has information about the state of the data), another use of the datapath and access to the location might be required. Also, dynamic binding could increase the complexity of control. Therefore, binding can create extra overhead compared with non-binding at that time. Although the extra overhead can be hidden from the critical path of a data access, it can still make other data accesses longer due to possible conflicts. However, a current binding can reduce future data access latency. Thus, a memory system should commit the binding only when the new data arrangement provides a better chance of reducing the overall data access latency.

To utilize locality of reference, we organize memories hierarchically — recently referenced data and temporary values used in a program are stored in the higher levels of the memory hierarchy [PH96]. In a typical memory hierarchy, three address spaces can be found: virtual address space on disk, physical address space in main memory, and register address space in register files. Generally, the virtual and register address spaces are specified in a program while the physical address space is utilized to map the virtual address space, i.e., to rename the virtual addresses. The separation of the two address spaces used in a program contributes to enormous performance improvements since the registers contain temporarily used data and reduce the number of references to main memory. If a compiler cannot be certain that data is only temporarily used or if the size of the data is too large to be captured by the register files, then virtual addresses are used as the names of the data. Using virtual addresses instead of physical addresses in a program makes it possible for many processes to share the same physical memory. Also, they can use more address space than the existing physical address space because the disk is used for storing data which overflow from the physical memory. Thus, only the working set of a program could be maintained in main memory. Usually, the initial mapping between virtual and physical addresses is done by an operating system, while subsequent address translations from virtual to physical address are done by the hardware.

Caches [PH96] between processors and main memory are another approach to design of the memory hierarchy. To satisfy memory bandwidth requirements between levels of the memory hierarchy, the unit of data communication between the cache and the main memory, a *line*, is much smaller than the unit between the main memory and the disk, a *page*. Most memory architectures bind the recently referenced address to the cache and keep a valid copy in the small and fast cache at the time of a cache miss with the assumption that the newly bound data will be reused soon. Thus, the access latency to the data can be shorter next time. Compared with non-binding, which does not cause new arrangement of the data, binding data to the cache requires extra overhead. Data must be rearranged, referenced data saved into the cache, and replaced data relocated to main memory. For cache coherence, control complexity is increased as well. Since patterns of reference at the cache are highly localized, the extra overhead can be easily amortized by the faster data access latency under the new data arrangement.

Utilizing memory local to a node in a multiprocessor system is as important as utilizing the caches especially when the working set of a program does not fit in the cache. However, the role of local memory and its usage have not been well established yet because the problems are not clearly defined. If we know the reference patterns of a program before execution, we can schedule the binding times so as to speed up data access. A message-passing architecture [ABC$^+$95, DCC$^+$87] is one which can support the scheduled binding times by using special instructions such as send and receive.

Unlike message-passing architectures, a distributed shared memory (DSM) architecture [HHL92, LLG$^+$90] utilizes the event of reference for the binding, i.e., a DSM architecture tries to keep recently referenced data in the local memory at a data miss time on the assumption that the data will be referenced again in the near future. Thus, the DSM architecture provides a simple programming model since communication among memory locations for data replication and migration is implicitly carried out in accordance with the reference events of a program. That is, the arrangement of data in the local memories is transparent to programmers. Also, even when the reference patterns of a program cannot be revealed before the execution of the program, DSM architectures can still provide good system scalability. However, the DSM architectures charge an additional cost for their automatic control of the data arrangement. Locations where requested data exist and where replaced data can be saved must be found, and data states must be updated. Minimizing these costs is a hot issue in DSM architectures.

When we utilize locality of reference at the local memory in a DSM architecture, we need to decide the binding time carefully. References to local memory may not be as highly localized as references to the cache since the reference to the local memory happens only when the referenced data is missing at the cache. In general, the higher the cache hit rate is, the lower the local memory hit rate is. Also, unlike in conventional caches, the replaced data from local memory can cause side effects on locality of data in remote memory. The question we address here is what is the proper binding time for the local memory when a page fault, a node miss, or a cache miss occurs. Using variations in binding time, we will characterize the various DSM architectures, and understand their performance.

1.1 Contributions

- This dissertation proposes a DSM architecture which permits binding at the event of a cache miss.
- This thesis introduces a directory structure which supports an efficient relocation algorithm for the data replaced from the local memory.
- Also, the storage requirements of the directory are minimized without limiting the number of pointers per line address.
- A non-blocking directory protocol for an efficient arrangement of data is described .
- A qualitative comparison among the DSM architectures is performed in detail.
- For the quantitative evaluation, DSM machines are designed and simulated.

1.2 Organization

Chapter 2 of our discussion describes the events of reference which could trigger the bindings for a new data arrangement. In Chapter 3, we examine relative performance of DSM architectures under some typical patterns of references. For performance study under realistic reference patterns, DSM machines are designed, analyzed, and compared in Chapter 4. In Chapter 5, the machines are simulated and the simulation results reported. Chapter 6 offers a conclusion.

Chapter 2

Events in Memory Architectures

Before studying memory architectures at the level of memory references (reads and writes), we will take a look at how memory systems could be organized. Based on a memory organization, we can define the event of a memory reference.

2.1 Memory locations

Figure 2.1 shows a possible memory organization in a multiprocessor system. In the figure, we can find three levels of organization: cluster, group, and site. Each level contains several nodes. A node consists of a processing unit and a storage unit. The storage unit for the cluster, group, and site are called caches, memories, and disks respectively. For communication between levels, address mapping is maintained between the processing unit and the storage unit. Shared virtual address space is utilized in the site. Here, note that all virtual addresses used in a program may not always appear on the physical storage (i.e., disks). A part of virtual address space is translated into physical address space, and used in the group. Again, all of the translated (or renamed) physical addresses need not be in the physical storage (i.e., memories). Some parts of the physical address space are more actively bound to the cluster, although virtual caches [PH96] in the cluster eliminate address translation from virtual to physical addresses.

A good memory organization is one that is very scalable to any number of nodes and which achieves low cost without sacrificing programmability. Whenever we increase the number of nodes in a level, we have to consider two system parameters relevant to the datapath — data transmission rate and distance between nodes. The data transmission rate determines the amount of data that can be transferred per unit time. The time to propagate a signal from one node to another depends on the distance between the two

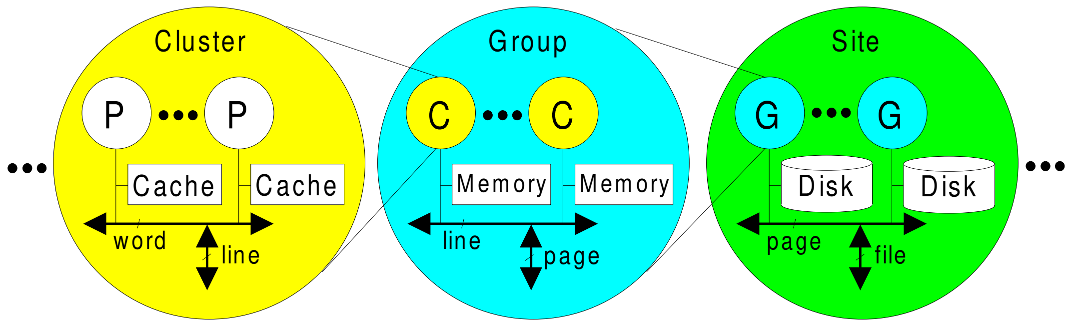


Figure 2.1: Distributed memory systems
In the figure, P represents a processor. The cache, the memory, and the disk are the three typical memory locations in a computer system.

nodes (propagation delay). By increasing the data transmission rate or by reducing the node distance, we could achieve a high datapath bandwidth. For a program to be executed on the system without causing a communication bottleneck, sufficient bandwidth must be available on each datapath. If the bandwidth requirement of a program is larger than the provided bandwidth, substantial performance degradations can result from this communication bottleneck. Maintaining a working set of a program in a level can reduce the bandwidth requirement between levels (vertical arrows in Figure 2.1). For example, by maintaining a working set of a program in main memory, the operating system can reduce the bandwidth requirement between main memory and the disk, and hence avoid thrashing [Den68].

After the bottlenecks between levels are resolved, the next consideration for node scalability is removing the bottleneck within a level. Bandwidth requirements within a level (horizontal arrows in Figure 2.1) can also be reduced by maintaining a working set in each node of the level. The role of the storage unit local to a node, thus, is to reduce the amount of traffic among nodes in each level for large-scale multiprocessors.

The data size for each communication should be determined by access time to the storage unit and by the bandwidth requirement needed in each datapath. The difference between disk access times for small and large data sets is very small compared to the disk access time itself. Since memory access time is much smaller than disk access time, the difference between memory access times for small and large data sets becomes much larger relative to the memory access time itself. To balance the bandwidth requirement between a group and a site, pages used in a site are significantly larger than the lines used in a group. If the time for network communication between nodes is relatively long, the data size for inter-node communication may need to be larger than the data size for intra-node

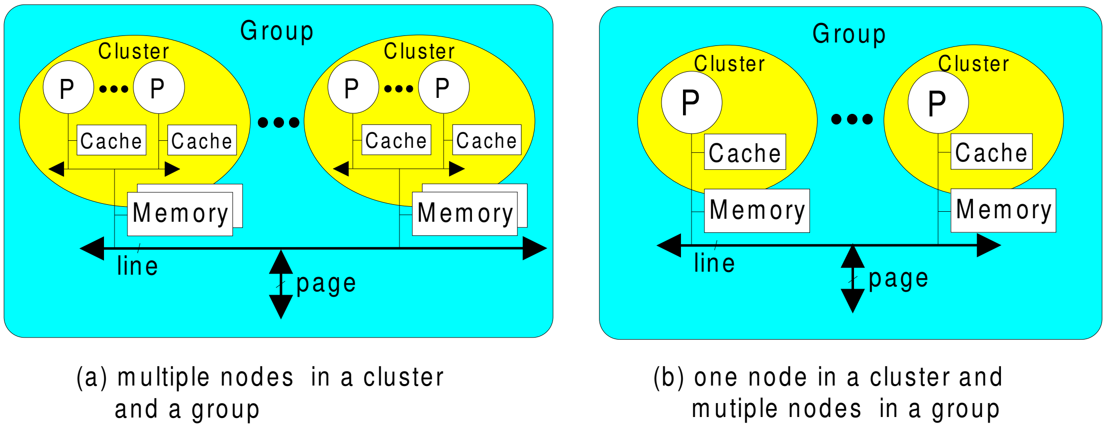


Figure 2.2: Two different memory organizations
In the figure, P represents a processor and both organizations have the same number of processors.

communication. However, since this thesis assumes that the system nodes are located very closely to one another, we do not take into account the different data sizes within a level.

Centralized shared memory has the organization of one node in a group and multiple nodes in a cluster. If a group consists of multiple nodes, the organization is called a DSM (distributed shared memory) organization. A distributed virtual shared memory (DVSM) organization consists of multiple groups in a site. When the total number of nodes is small, the organization of multiple nodes in a cluster could be preferable. However, in a large-scale multiprocessor, storage units should be organized differently because the local cache is too small to prevent a communication bottleneck in the clusters. For a large-scale multiprocessor, we can consider two possible organizations: an organization of multiple nodes in a cluster and in a group (Figure 2.2 (a)); and an organization of one node in a cluster and multiple nodes in a group (Figure 2.2 (b)). Here, we assume the distance between processors is the same in both organizations. If the data size in the cluster is a word and there is a scheme to maintain locality among the caches, then having multiple nodes in a cluster is more attractive. However, few schemes exist to keep locality in the cluster and the data size in the cluster is usually a line, not a word. Therefore, there is almost no difference in the frequency or latency of the data accesses to a remote memory in the two different organizations. That is, the two organizations may have similar performance. This thesis focuses on the organization of one node in a cluster (Figure 2.2 (b)) to reduce implementation costs.

reference seq.	ref_0	ref_1	ref_2	ref_3	ref_4	ref_5	ref_6	ref_7	$\cdots$
cache	Miss	Hit	Miss	Hit	Miss	Hit	Miss	Hit	$\cdots$
node	Miss	Hit	Hit	Hit	Miss	Hit	Hit	Hit	$\cdots$
system	Miss	Hit	Hit	Hit	Hit	Hit	Hit	Hit	$\cdots$
event type	type 0	type 3	type 2	type 3	type 1	type 3	type 2	type 3	$\cdots$

type 0: a page fault, type 1: not a page fault but a node miss,
type 2: a node hit but a cache miss, type 3: a cache hit

Table 2.1: Events of references characterized by the location of data.

2.2 Reference events

For a given memory organization, data might need to be arranged dynamically among its storage units for high performance. For the dynamic data arrangement, we can consider two methods: an *implicit* method and an *explicit* method. The explicit method explicitly specifies the data arrangement in a program using some binding (or communication) instructions such as send, receive, broadcast, and reduce. If we know the reference patterns of a program, we can allow data to be rearranged at an appropriate time by inserting the binding instructions into the program. A message-passing architecture adopts the explicit method. That is, a message-passing architecture brings new data into the local memory by using the binding instructions before the processor generates a reference to the data. Thus, the binding instructions in a program guarantee that all data referenced by the processor can be found in its local memory.

Due to its hardware simplicity, the message-passing architecture has been used in many large-scale multiprocessors. However, the message-passing model restricts programmers to passing data by value between nodes. In other words, programmers are responsible for the dynamic arrangement of data. If a program has a reference whose address is not known before the run time, it may be hard to execute the program efficiently on the message-passing architecture. Further, since programmers usually need to schedule processes, balancing the workload across nodes is another serious problem for this architecture.

The implicit method uses the event of a reference to arrange data. When a memory reference is issued by the processor in a node, the event can be characterized by three aspects; location of data, state of data, and reference type (read or write). Depending on the location of the valid data referenced by the processor, the event is called a cache hit (type 3), a cache miss but a node hit (type 2), a node miss but not a page fault (type 1), or a page fault (type 0). Table 2.1 illustrates the four different types of the events. The locations of data which are not referenced by the processor characterize the event as well

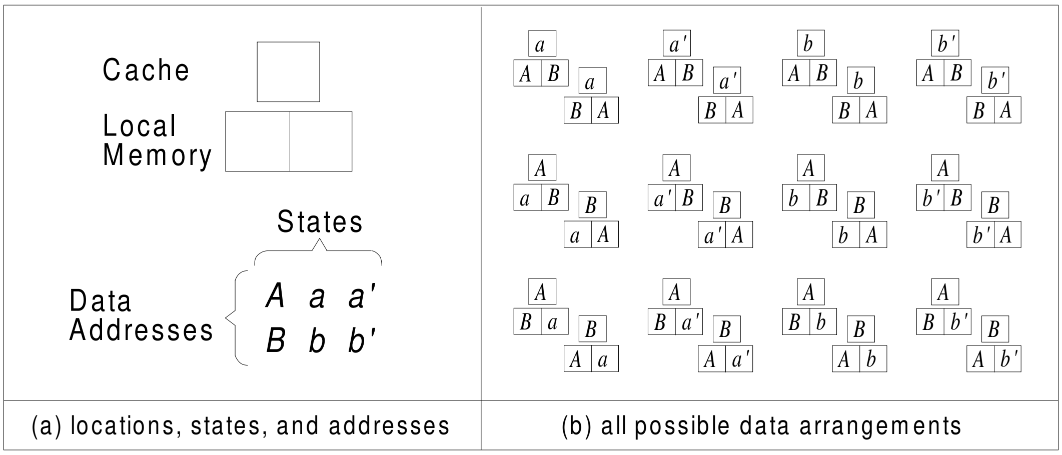


Figure 2.3: All possible data arrangements in a memory system
In the figure, A, a, and a' represent an owner, a non-owner, and an invalid datum with the same address A respectively.

since they could cause different memory behavior. For example, to decide to fetch data in advance, a prefetch algorithm [Smi82] needs to know the locations of the data that are not referenced by the processor. Another important property of an event is the state of the data related to the event. The data can be shared by several nodes. There can exist an ownership copy among the shared copies of the data. The data could be invalid. Also, the data in the cache may or may not be in its local memory. A great number of other states are possible though not mentioned here. Also, for practical reasons, the number of states can be restricted to a small number in a given machine.

Now, let us see how many different events can exist in a memory system by using a simple example. In the example, a memory system is organized with a one-frame cache and a two-frame local memory. This is shown in Figure 2.3 (a). In the memory system, there exist two addresses (A and B) and three possible states for each address. A, a and a' respectively denote owner, non-owner, and invalid states for the same address A. Since there is one and only one ownership copy for each address, the number of possible ways of allocating memory locations for the ownership copies is the number of ways of taking l out of m $(= \frac{m!}{l!(m-l)!})$, where m is the number of memory locations and l is the number of addresses. In our example, m is 3 and l is 2. Given the allocation, the number of possible arrangements for the ownership copies is $l!$. The number of arrangements for the other state copies on the remaining memory locations is $(l(b-1))^{(m-l)}$, where b $(= 3)$ is the number of states. Therefore, the total number of data arrangements on the memory locations is $(\frac{m!}{l!(m-l)!})(l!)((l(b-1))^{(m-l)}) = 24$. These 24 data arrangements are shown in Figure 2.3 (b).

<table>
<tr><td rowspan="2">Memory architecture</td><td rowspan="2">Data size</td><td colspan="3">Fresh data initiating the LM bindings</td><td colspan="2">Implementation method</td></tr>
<tr><td>event type 1</td><td>type 2</td><td>type 3</td><td>requirement</td><td>implement</td></tr>
<tr><td>CC-NUMA</td><td>line</td><td></td><td rowspan="8"></td><td rowspan="7"></td><td rowspan="6">page table (no change of physical addresses)</td><td>hardware</td></tr>
<tr><td>page mig.</td><td rowspan="2">line, page</td><td rowspan="2">if (count = K) referenced data</td><td rowspan="2">hardware, software</td></tr>
<tr><td>R-NUMA</td></tr>
<tr><td>DVSM</td><td>page</td><td rowspan="6">referenced data</td><td>software</td></tr>
<tr><td>S-COMA</td><td rowspan="2">line, page</td><td rowspan="2">hardware, software</td></tr>
<tr><td>multiplexed</td></tr>
<tr><td>COMA</td><td rowspan="5">line</td><td rowspan="5">physical address tag in the local memory</td><td rowspan="5">hardware</td></tr>
<tr><td>loose-incl.</td><td rowspan="2">referenced data</td></tr>
<tr><td>no-inclusion</td><td rowspan="2">referenced data, rep. cache data</td><td rowspan="3">replaced cache data</td></tr>
<tr><td>E-NUMA</td><td rowspan="2"></td></tr>
<tr><td>DYMA</td><td>rep. cache data</td></tr>
</table>

Table 2.2: Comparison of DSM architectures

Since the processor can generate a read or write reference for each address, the number of all possible events in our memory system is $24 \times 2 \times l = 96$.

From the above analysis, it is possible to characterize an event with a myriad of combinations of data locations, data states, and the reference type. One simple reference event is a write hit on shared data in the local memory. Here, note that the history of past events can also characterize the current event, for example, the number of occurrences of a particular event [CDV$^+$94, FW97], and the dynamic detection of migratory data [CF93, SBS93]. In theory, any fresh data can be bound to any memory location in any event of a reference. In the example of Figure 2.3, at each memory reference, the data arrangement on the memory locations could be any of the 24 possible data arrangements.

DSM architectures exploit the implicit method. By inserting special binding instructions, such as the `prefetch` instruction [Smi82], into a program, the explicit method can also be utilized in the DSM architectures. However, unlike in message-passing architectures, the data bound to a memory location by a binding instruction can implicitly be replaced from that memory location next time. The protocol of a DSM architecture describes how to arrange data in memory locations for each event of a reference. In the following section, we will see some DSM architectures which have different arrangements of data in each event of a reference.

2.3 Case studies

Table 2.2 shows some DSM architectures categorized by their data sizes for network communication, binding times to the local memory (LM), and implementation methods. Here, we assume that the procedure for handling page faults (type 0) is the same in all memory architectures. That is, the architectures bring a page into the local memory at a page fault time with the help of the same hardware and software scheme. Also, the line missing in the cache is always allocated in the cache in all architectures. Thus, in the following discussion, we focus just on the data in the local memory. A Cache-Coherent Non-uniform Memory Access architecture (CC-NUMA) does not permit bindings in different events (type 1, 2, and 3). The data unit for communication between nodes is a line. Thus, CC-NUMA is usually implemented with a hardware-oriented protocol.

If a remote node accesses a page much more frequently than the currently owned node, the page may need to be migrated to that remote node. Page migration in CC-NUMA [CDV$^+$94, LL97] supports that. In general, we do not know which node will access the page frequently until the accesses are actually made at run time. If a node makes a certain number (K in Table 2.2) of remote accesses to a page, the page migration is initiated, assuming the node will continue to access the page frequently. When the page is migrated, the original page copy is invalidated. Examples of CC-NUMA machines are Stanford DASH [LLG$^+$90] and FLASH [KOH$^+$94], MIT Alewife [ABC$^+$95], Sequent STiNG [LC96], HP Exemplar [BA97], Data General NUMALiiNE [Cor97], HAL S1 [WGH$^+$97], and SGI Origin 2000 [LL97].

Similarly in a cache, one may want a page to be replicated whenever the node causes a node miss for that page (the cache brings a new copy whenever it has a cache miss). The cache-like function at the local memory can be implemented in software as in distributed virtual shared memory (DVSM) [LH89, KCDZ94]. However, it may be slow due to its software-based control. This is the reason why the unit of data communication is usually a page in software-based schemes although a fine-grain data communication can be accomplished in software [SFL$^+$94, SGT96]. The page-sized granularity of data allocation and communication could take up memory space and network bandwidth [BLA$^+$94, IDFL96]. Also, the page-sized granularity of the sharing unit may increase the amount of false sharing data [LYL87, TLH94]. Simple Cache Only Memory Architecture (S-COMA) [SWCL95], such as SUN S3.mp [NAB$^+$95], tries to overcome the drawbacks of the DVSM by adding hardware coherence control. The unit of data for the coherence control is a line, which is much smaller than a page. Thus, the amount of network traffic caused by false sharing can be reduced. However, it still has limitations in utilizing memory space and network bandwidth because a large page can contain a fairly large amount of redundant data. As a solution to the memory fragmentation problem in S-COMA, a multiplexed S-COMA is proposed [BT98]. By mapping multiple virtual pages to the same physical page working set, the page working set of an application can be compressed in size. However, the software which handles the necessary complex address translation can slow down program execution in the multiplexed S-COMA.

The reactive NUMA (R-NUMA) [FW97] does not replicate a page at each node miss. Instead, it replicates a page when a node makes a certain number (threshold value K in Table 2.2) of remote accesses to the page. Unlike in CC-NUMA with page migration, the original page copy is not invalidated when it is replicated. Although the replication of the page-sized data is carried out by a software-oriented scheme, such as the operating system, the coherence control for line-sized granularity is supported by a hardware scheme as in S-COMA. There are many ways of detecting the need for page replication. The directory can maintain a counter per node for each page [LL97]. Each node can maintain a set of per-page counters [FW97]. Or, a page relocation counter can be attached to each cache set [MD98]. The threshold value (K) which maximizes its performance could be changed depending on programs and memory pressure. Thus, the study in [KCKS98] proposes a scheme which dynamically adjusts the threshold value.

It is also possible to divide the local memory into two parts: space always allocated for memory lines as in CC-NUMA and (unallocated) space utilized for data replication as in DVSM. All of the schemes discussed so far could still apply for the hybrid use of the local memory. For example, the Stache approach [RLW94] implemented in Typhoon-0 [RPW96] uses the unallocated space as the local memory of S-COMA, i.e., it allocates memory at page granularity in software and manages coherence at line granularity in hardware.

The above memory architectures do not allow the physical address of a memory location to be changed. Thus, to rearrange a page at run time, the virtual (or global [SWCL95]) address of the page has to be mapped to the physical address of the new memory location, i.e., the mappings in the page tables must be changed. At the same time, the TLBs of the processors may need to be modified as well. This process could be so expensive that the advantage of the page arrangement could be annulled [CSG97]. In the following hardware-oriented memory architectures, the unit of data communication between nodes is a line, which is much smaller than a page. Arranging lines in the local memory dynamically requires extra hardware such as physical address tags in the local memory.

Hardware-oriented COMA [HHL92, JH94, LQK94], such as the Kendall Square Research KSR-1 [IFKR92], commits the bindings at a node miss (type 1), i.e., the referenced data missing in a node is bound to the local memory when the node retrieves the data. The loose-inclusive COMA proposed in [DL97] allows binding at a cache hit (type 3) as well as at a node miss (type 1). On a read miss in a node, a clean copy from a remote node is allocated in the local memory only if a non-ownership or invalid copy can be replaced. If the reference type is a write at a node miss, the incoming copy which has ownership is always saved in the local memory. When the processor writes on a clean copy in the cache (type 3) and the copy does not exist in the local memory, the space for the copy is allocated in the local memory. Thus, the copy replaced from the cache at a cache miss can always be written back in its local memory if the copy has ownership. That is, ownership copies are guaranteed to inclusively reside in nodes. There is no new binding to the local memory at a node hit but a cache miss (type 2). When an ownership copy is replaced from the local memory at a node miss or at a cache write hit, it is relocated to the local memory of a remote node.

The no-inclusion COMA [DL97] allows new data to be bound to the local memory at a cache hit (type 3) and a cache miss (type 2 and 1). If there is available space in the local memory (non-ownership or invalid copy) on a node miss (type 1), the referenced data is bound to the local memory. When a write hits on a shared copy in the cache (type 3) and there is available space in the local memory, the copy is allocated in the local memory. Binding at a cache miss (type 2 and 1) can happen when an ownership copy is replaced from the cache. At that time, if another ownership copy is replaced from the local memory, it is saved in a remote node.

The binding in Excel-NUMA (E-NUMA) [ZCT99] happens at a cache miss (type 2 and 1). When a clean (non-ownership) copy is brought from a remote node due to a node miss (type 1), the copy is saved in the local memory. When an ownership copy is replaced from the cache at a cache miss (type 2 and 1), the replaced copy is saved in the local memory. In E-NUMA, every line has a home node and a high priority at the home. Thus, the local memory is divided by two parts; memory cells for both high and low priority lines and remote caches for low priority lines. When a low priority copy which has ownership is displaced at the local memory, it is always relocated in the local memory of the home node though this could induce another displacement from the local memory.

Dynamic Memory Architecture (DYMA) [KL96] allows binding when a fresh copy is discarded from the cache due to a cache miss (type 2 and 1). The data coming from a remote node at a node miss (type 1) is inserted only in the cache, not in the local memory.

From the above observations, to exploit program locality, most DSM architectures attempt to bind recently referenced data to a memory location as close to the referencing processor as possible. Generally, no bindings are made on a cache hit, and a cache miss causes the referenced data to be stored in the cache. The missing data is retrieved to the node at a node miss. Also, a page fault initiates the loading of a new page into main memory. It is possible to allow bindings to the local memory at a cache hit (type 3) as in the loose-inclusion or no-inclusion COMA. However, it seems obvious that bindings at cache hits are redundant.

This thesis shows how to arrange data in the local memory to utilize the locality of references effectively. Studying all possible data arrangements is beyond the scope of this thesis. Instead, we focus on the three DSM architectures: CC-NUMA, COMA, and DYMA. As shown in Table 2.2, most of the other DSM architectures have similar features to these three DSM architectures. Therefore, we believe the performance of the other DSM architectures could also be explained by comparison with the performance of CC-NUMA, COMA, and DYMA. Since CC-NUMA allows data arrangement only at a page fault (type 0), there might be severe restrictions on utilizing locality of references in the local memories. COMA can utilize the locality of references in the local memories. However, the binding of referenced data to the local memory at a node miss (type 1) is so early that the data in the local memory could be out-of-date even before being discarded from the cache. Since DYMA can permit the bindings only when a valid copy is discarded from the cache on a cache miss, DYMA can effectively utilize the locality of references in the local memory.

Chapter 3

Three DSM Architectures

To arrange data in memory locations and to change the state of data at a reference event, each memory architecture charges different overhead. In the following section, we analyze the relative overhead for the three DSM architectures under some patterns of reference.

To see how different binding times can help to predict reference patterns and reduce the memory access time with different overheads, consider the simple configuration shown in Figure 3.1. In the figure, the cache and the local memory (LM) have two and four frames respectively, and they are direct-mapped. A, a and a' in Figure 3.1 respectively represent an owner, a non-owner, and an invalid data line with the same address A. Initially, A, B, C and D are in Node_i's LM, and a and d are in Node_i's cache. E, F and G are in Node_j's LM, and e and f are in Node_j's cache. Here, we assume a write-invalidation control mechanism to support the following discussion.

When processor P_i in Node_i reads line E at time t_1, it will trigger a node miss, and e is brought to Node_i. To utilize locality in the cache, every architecture, including CC-NUMA, COMA and DYMA, binds e to Node_i's cache. For this binding, access to the cache to store line e and to replace line a is additionally required in all three DSM architectures. The differences among these architectures happens in the LM. Since the replaced a from Node_i's cache is neither updated nor new to the LM, no more overhead is required in both CC-NUMA and DYMA. Since e is new to Node_i's LM, COMA binds e to the LM. That is, COMA assumes that some overhead can be reduced when e is later replaced from the cache, and assumes that e is more likely to be referenced again than A is when a cache miss happens. To bind e to Node_i's LM, an access to the LM is needed. Also, note that if A is the last copy in the system, it should be relocated to a remote memory. In this case, two more LM accesses (copy line A from Node_i's LM and save it at a remote LM), some network bandwidth, and some directory processing (for a directory-based system) are

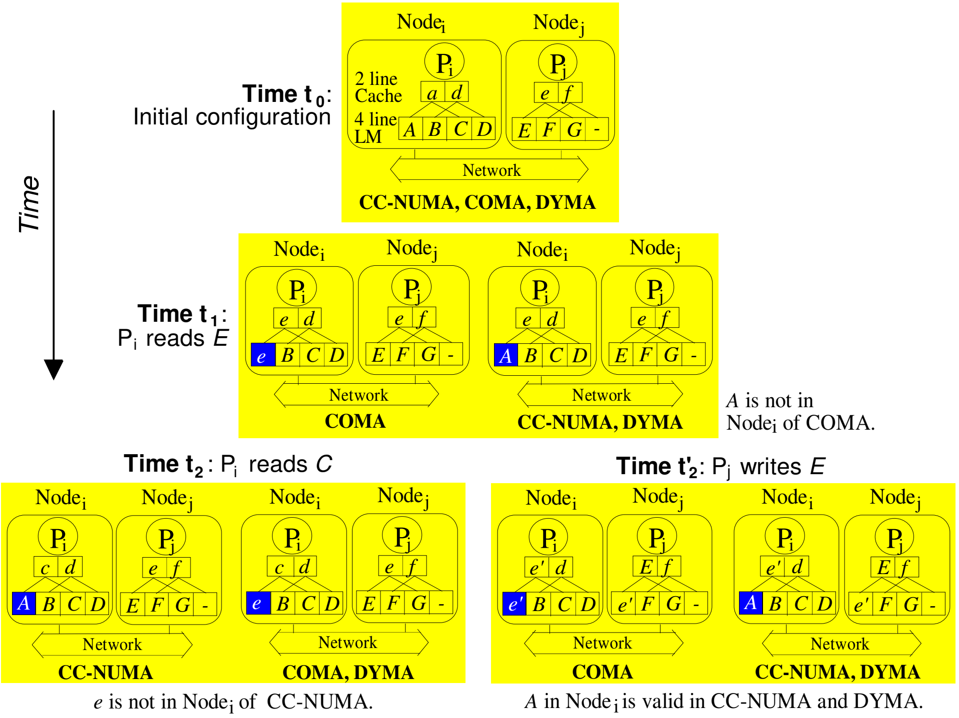


Figure 3.1: Arranging data in DSM
In the figure, A, a, and a' represent an owner, a non-owner, and an invalid line with the same address A respectively.

necessary. Furthermore, if A is relocated to Node$_j$, the copies of E are removed from Node$_j$ since the inclusion property should be enforced to prevent a new line from being bound to a local memory at a different time and to keep coherence control from becoming overly intricate [KL96]. This can reduce P$_j$'s cache and LM hit rates. Since the replaced a from Node$_i$'s cache is not new to the LM, the extra overhead incurred in COMA does not occur in DYMA. DYMA has the same data arrangement as CC-NUMA, which can cause a higher node hit rate than that of COMA since A is not in Node$_i$ of COMA.

If processor P$_i$ reads address C at time t$_2$ (a cache miss), e will be discarded from Node$_i$ in CC-NUMA even though address E is more recently referenced than address A. Thus, CC-NUMA can increase the node miss rate. Since e already exists in the LM of COMA, no write-back occurs. DYMA binds e to the LM at this time since e is fresh to the LM. The configuration of DYMA becomes that of COMA. Unlike COMA, DYMA needs an extra access to Node$_i$'s cache to have a copy of e, but can relocate A to Node$_j$'s LM without removing e from Node$_j$'s cache. Under the new data arrangement, a higher LM hit rate may compensate COMA and DYMA for the extra overhead at time t$_1$ and t$_2$.

Memory architecture	Overhead at time t_1			Overhead at time t_2			Overhead at time t'$_2$		
	cache access	LM access	network use	cache access	LM access	network use	cache access	LM access	network use
CC-NUMA	1	1	1	1	1	0	2	1	1
COMA	≥ 1	≥ 4	≥ 2	1	1	0	2	2	1
DYMA	1	1	1	2	≥ 4	≥ 1	2	1	1

Table 3.1: Overhead for data arrangements.

Before e is removed from the cache of Node$_i$, if processor P$_j$ writes to address E at time t'_2 (a cache write hit on shared data), every shared copy whose address is E is invalidated for data consistency under write-invalidation coherence control. For this coherence action, sending invalidation signals through network, invalidating the shared copies, and returning acknowledgments through the network are needed. Although no new bindings are necessary in this event, the states of data are changed. In COMA, the access to Node$_i$'s LM tag to invalidate e is extra, and nothing can compensate for the extra overhead incurred at time t_1 and t'_2. A higher node miss rate can be observed in COMA than in CC-NUMA and DYMA.

Another possible event of reference can occur when e in Node$_i$'s cache is modified and selected for replacement at time t''_2 (a cache miss on an owner line). This is not shown in the figure. In this case, writing E back into Node$_i$'s LM is again required in COMA because e' in the LM was invalidated. The write-back in COMA does not cause any effects on the locality of data in a remote memory. DYMA has the same overhead incurred at time t_2. CC-NUMA should write E back to Node$_j$'s LM through the network although CC-NUMA does not generate any further bindings. In CC-NUMA, the write-back from Node$_i$ increases the number of valid copies in Node$_j$.

In this section, we examined the relative overhead and way of arranging data in the three DSM architectures for some events of references. Table 3.1 summarizes the system overhead incurred by the data arrangements in Figure 3.1. From the table, we find that no one architecture can outperform the others across all possible reference patterns. The relative performance of the architectures varies based on the reference patterns of a program. In the following sections, we will discuss each architecture in depth with emphasis on the general reference patterns and their strengths or weaknesses.

The reference patterns to the local memory are very sensitive to the size of a working set relative to the size of the cache. If the working set is small enough to fit in the cache, few cache lines will be replaced, and seldom referenced again after being replaced from the cache. However, if the cache does not match well with the working set of a program, a large number of cache copies will be replaced, and most of the replaced copies are likely to be referenced again. Also, in some patterns, some shared addresses are extensively referenced by several processors. In some other patterns, most cache copies are not modified when they are replaced.

3.1 Page-fault time: CC-NUMA

When a new page is brought into main memory from disk due to a page fault, in CC-NUMA, all of the lines in the page are bound to some local memories, and the memory locations for the lines are not changed until the lines are relocated or replaced from main memory at another page fault time. Thus, the address set of the lines bound to the cache is a subset of the address set of the lines in main memory. This *inclusion property* enables CC-NUMA to avoid replacements of a line from the local memory in different reference events. The reference patterns would change dynamically during the sojourn of the page in main memory. Therefore, cache missing data is open in a remote memory location. A replaced line from the cache at a cache miss time is also open written back into a remote memory. Even though such a write-back does not cause any further displacements, the replaced line does not improve locality of reference in the affected remote node either. If the working set of an application is large, a large number of data accesses and replacements have to go to a remote memory through the underlying network. Thus, CC-NUMA may suffer from seriously long memory reference time, as did centralized shared memory architectures, if the network service time is long and the network bandwidth is too narrow to remove communication bottlenecks. If the working set is small, CC-NUMA may perform well since less network traffic is generated and there are no extra bindings to the local memory for the temporarily used data in the cache. If we can predict the reference patterns of a program before binding to the local memory at a page fault, determining how to partition and distribute data among local memories to reduce network traffic is the main issue in CC-NUMA.

3.2 Node-miss time: COMA

For more precise locality of reference in the local memory, COMA binds a new line to the local memory, called attraction memory (AM), when the line is found missing in the node. The newly bound address will be unbound from the AM at another node miss. Thus, COMA rearranges data in the AMs at a node miss. Since a node miss happens only when there is a cache miss, the missing lines are loaded into the cache inclusively. Also, in order to prevent a line from being bound to the local memory at a different time and to keep the coherence control from becoming overly intricate, the inclusion property between the cache and the AM is enforced even when replaced lines from a node are relocated to some other node.

Using the AM in this way, the overhead at a cache miss with a node hit (type 2) can be reduced, compared with CC-NUMA, since no replaced lines from the cache consume network bandwidth in COMA. The data arrangement in COMA at this time can also reduce future node misses than in CC-NUMA because the replaced lines from the cache are always kept in the AM. However, the reference patterns generated by cache misses to the AM may not be as highly localized as in the cache. The AM binding at a node miss (type 1) can create extra overhead which does not occur in CC-NUMA. At a node miss, CC-NUMA requires at most two accesses to the local memory — copying the missing line from main memory (assuming the copy in main memory is valid) and, if necessary, writing a replaced cache copy back into main memory. Even if we assume that the write-back from the cache is not necessary at a node miss, COMA requires at least two accesses to the AM: copying the missing line from a remote AM and saving the line in its AM. Here note that a scheme to check whether the missing line is in the AM or not is required as well. If a replaced AM line is the last valid copy in the system, at least two more accesses to the AM are required to copy the replaced line from its AM and to save the copy in a remote AM.

The replacements frequently happen especially when *memory pressure*, $r = \frac{l}{f}$, is high, where l is the number of line addresses in the physical address space, and f is the number of locations (or frames) in all AMs. The possibility of selecting the last valid copy per node miss is approximately r^a [JH94], where a is the AM's set associativity. Thus, the required network bandwidth in COMA can be larger than in CC-NUMA. Another additional overhead can be observed in COMA when coherence enforcement is needed. All of the nodes which share a line can have two copies of the line, while in CC-NUMA only the home node can have two copies. Thus, the number of accesses to the AMs for invalidation in COMA can be larger than in CC-NUMA. Even if there is an algorithm which can hide the extra overhead (especially for replacements) from the critical path of a data access, the overhead can still make other data accesses longer due to possible contention for system resources.

The new data arrangement at a node miss does not always have the positive aspects of reference locality. The address newly bound to its local AM would more likely be referenced again by its cache misses than the other existing addresses in the AM. However, the address bound to a remote AM may not be recently referenced by the remote node. Thus, the new data arrangement can increase the miss rate of the remote AM. Furthermore, the inclusion property enforced between the cache and the AM can cause negative effects on the locality

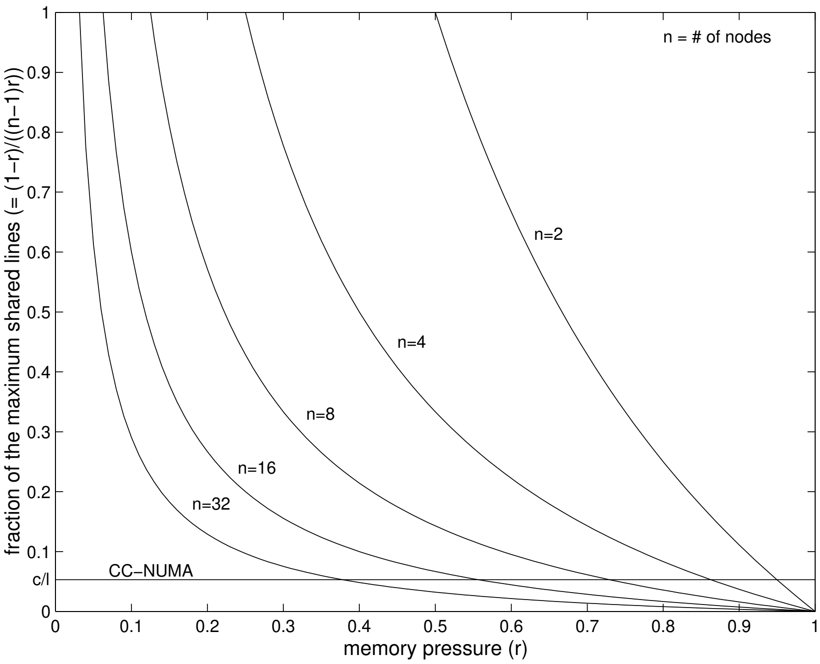


Figure 3.2: Limited data sharing in COMA

The figure shows that the maximum number of lines shared by n nodes drops dramatically as memory pressure r and n increase.

of the remote cache [KL96]. Therefore, if the reference patterns to the AM are not highly localized, it may be hard for COMA to offer performance advantages over CC-NUMA primarily due to the extra overhead and the side effects of affecting locality of data in a remote node at binding time. The reference patterns of a small working set may not be strongly localized at the AM since cache misses rarely happen. When the working set is too large to be captured by the cache, the replaced lines from the cache would frequently be referenced again in the near future. Since a very large AM could keep the replaced lines, a large working set can improve the locality of reference at the AM and make the side effects trivial. Here, we can expect a synergy of the AM miss rate between the actions of saving the newly referenced line in an AM and relocating the replaced line into an AM. A higher AM miss rate increases the number of replacements, and hence further increases the AM miss rate.

Another drawback of COMA is its limitation on the number of addresses simultaneously shared by system nodes. For example, let us consider the configuration in Figure 3.1 again. We assume that the system has only two nodes, Node_i and Node_j, and both nodes want to have the shared line e at the same time. Since line A is the last valid copy, it should be either in Node$_i$ or Node$_j$. The line e cannot reside in both nodes simultaneously. To see how the number of shared addresses is limited by memory pressure ($r = \frac{l}{f}$) and the number of system nodes (n), let us assume a fully associative AM. Then, the maximum number of addresses that can be shared by n nodes is $\frac{f-l}{n-1}$. Therefore, the ratio of shared line addresses to total line addresses is:

$$(\frac{f-l}{n-1})(\frac{1}{l}) = \frac{1-r}{(n-1)r}. \tag{3.1}$$

For instance, if $r = 0.5$ and $n = 2$, then $\frac{1-r}{(n-1)r} = 1$, i.e., both nodes can hold all of the line addresses at the same time. The relationship among the number of nodes (n), memory pressure (r), and the ratio of shared lines ($\frac{1-r}{(n-1)r}$) is drawn in Figure 3.2. As can be seen in the figure, the maximum number of lines that can be shared by n nodes is extremely small when memory pressure is high and the number of nodes is large. Generally speaking, if some shared lines are actively referenced by several processors, COMA under high memory pressure can suffer from high node miss rates because the shared data cannot be located at the requesting nodes simultaneously. Also, this situation increases the number of replacements. The maximum number of addresses that can be shared by n nodes at the same time in CC-NUMA is proportional to the cache size (c), i.e., the ratio is $o(\frac{c}{l})$, which is invariant to the system size and the memory pressure. To minimize the extra overhead and negative effects, one might want low memory pressure in COMA. However, low memory pressure makes caching space in main memory small for the data in disk. This contradicts the philosophy of COMA, accommodating a large working set with the local memories. From the above analysis, it is questionable how much COMA can improve performance over CC-NUMA, even with programs which have a large working set.

To maintain the inclusion property between the cache and the AM, several techniques can be used. Write-through introduces long memory access latency for write accesses even though the coherence control becomes simple. Another way to enforce the inclusion property is to probe the cache whenever there is a remote access request and broadcast invalidation signals to the cache when a line in the AM is replaced [Wil87]. By attaching an *inclusion bit*, which indicates whether a copy of the data is in the cache or not, to each AM frame [BW88], excessive probing of the cache can be diminished while maintaining the inclusion property. However, additional intra-node communication is still generated.

Some studies [Joe95, SWCL95] on COMA consider relaxing the inclusion property somewhat to improve performance. To relax the inclusion property in COMA, a prediction is required when a line is brought to the node (at node miss time). If a line that will be updated or invalidated is bound to an AM frame at the node miss time, the overhead for the binding is created unnecessarily. On the other hand, if a line that will not be invalidated is not bound to an AM frame, the line can easily be replaced from the node due to the small size of the cache. In this case, locality of reference cannot be exploited at the AM. Thus, either way limits the effective utilization of the AM, thereby diminishing the advantage of COMA. For example, if the inclusion property is simply relaxed for read-only (shared) data [Joe95], there is a limitation in taking advantage of the relaxed inclusion property fully since writable (shared) data to be invalidated in the cache still has access to the AM. Also, the read-only data may soon be replaced from the node since the cache is small. This may decrease the node hit rate if the read-only data is referenced again in the near future. If the writable shared data is allowed to reside in the cache without saving a copy in the AM, coherence control may become complicated because more state information needs to be controlled at the cache. Also, if the writable shared data is replaced from the cache without being modified, we will not be able to exploit locality for the writable shared data.

3.3 Cache-miss time: DYMA

Like CC-NUMA, on a node miss, DYMA saves the line coming from a remote node only in the cache and not in the local memory. The cache keeps the line as long as the line is persistently referenced. The longer the sojourn time of the line in the cache, the higher the possibility of the line being invalidated or updated by a write reference. Like COMA, on a cache miss, DYMA keeps the valid line replaced from the cache in the local memory, called dynamic memory (DM). Thus, in DYMA, the binding of a new address to the DM could happen at a cache miss as well as at a node miss and at a page fault. No inclusion property is enforced between the cache and the DM.

(1) If the line loaded from a remote node is invalidated by a remote write reference before being replaced from the cache, the overhead of binding to the DM does not occur. (2) If the line loaded from a remote node is updated by a local write reference before being replaced from the cache, copying the line from the cache and saving it into the DM are not additional overhead since the other architectures also have to write the updated line back into a local memory. (3) If the line loaded from a remote node is neither invalidated nor updated by a write reference when being replaced from the cache, copying the line from the cache is extra overhead since the other architectures can simply write over the clean line. The clean copy replaced from the cache is saved in the local DM with the assumption that the copy is more likely to be referenced again than is the DM line removed from the node. The replaced DM line can in turn induce another displacement at a remote node, leading to a new data arrangement as in COMA. Since no inclusion property is enforced between any two memory locations in DYMA, the replaced DM line can be saved in the DM of a remote

node without purging a cached copy from the remote node. Thus, no side effects are found in remote caches, and the overhead for binding to the DM may be less than in COMA. By delaying the binding to the DM, the DM line, which would have been discarded from the node in COMA, can be referenced again in DYMA. This increases the DM hit rate.

The copy missing in the cache can be from its local DM at a node hit. (1) If the cached line whose copy is in its DM is either updated or invalidated by a write reference, the same address copy in the DM will be updated or invalidated when the cached line is selected for replacement. (2) If the cached line whose copy is in the DM is clean when selected for replacement, the line may simply be discarded. However, if, for each cache replacement, DYMA tries to access DM tags to see whether the DM has the same address copy or not, another overhead may be incurred. By adding more state bits in each cache frame, we can develop a scheme where only an indispensable line, when discarded from the cache, is delivered to the DM to access the DM tags [KL96].

If most of the lines in a small working set are seldom replaced from the cache, the data configuration of DYMA is similar to that of CC-NUMA, which is desirable since the ineffective bindings to the local memory rarely happen. If most of the lines replaced from the cache are referenced again, as is most likely in a large working set, and if the effort of binding a new line to the DM for a higher local memory hit rate is not easily nullified by the side effects of replacements, DYMA can have the advantage of reducing data access latency over CC-NUMA. Unlike COMA, DYMA allows a line to be replicated to other nodes even under 100% of memory pressure. As a matter of fact, the maximum number of addresses that can be shared by n nodes in DYMA is the sum of those for CC-NUMA and COMA, i.e., $o(c) + \frac{f-l}{n-1}$. Thus, DYMA is robust under the patterns which actively reference some shared lines. However, if the cached lines are neither updated nor invalidated when they are unbound and if few of the replaced lines are referenced again, DYMA can cause an unwanted arrangement of data with higher overhead than the others.

The relationship between the AM of COMA and the DM of DYMA is similar to the relationship between the miss cache and the victim cache [Jou90]; the address of the miss cache is bound at the line-incoming time to the lower level cache as in COMA while the address of the victim cache is bound at the line-discarding time from the lower level cache as in DYMA. Due to the full associativity of the victim cache, all the victim cache lines are different from the lines of the lower level cache. Victim caching is always an improvement over miss caching [Jou90]. The victim caching is extended to the two-level exclusive caching [JW94] where the large second-level cache is used as a victim cache. The protocols for the miss, victim, and exclusive caches assume that the lower level cache is direct-mapped. Also, the miss, victim, and exclusive caches are designed to decrease average data access latency by reducing cache misses in uniprocessors, while the AM and the DM are designed to decrease average data access latency by reducing node misses in multiprocessors.

Until now, we assumed a conceptual datapath and its control mechanism to argue pro and con about the three DSM architectures. In the following section, we will design a realistic machine for the DSM architectures. Since it is possible that a practical control mechanism which supports a binding time is too complicated to have the overhead described in this section, we must examine the datapath and the control mechanism intensively. Minimizing the cost of implementing a machine is another goal of the machine design.

Chapter 4

Machine Design

In the previous chapters, we studied ways of arranging data in memory locations at an event of reference for a scalable system. We have learned that there are a huge number of possible data arrangements at a reference event. In fact, there are also large numbers of ways to move data for one data arrangement. This chapter discusses some automatic control protocols to support the data arrangements of CC-NUMA, COMA, and DYMA based on a hardware organization.

We can categorize the functions of DSM protocols into three: finding missing lines, arranging data in memory locations, and maintaining shared lines coherently. (1) Automatic protocols in DSM architectures provide schemes to find requested data and to supply the data to the processor. (2) For high performance, the DSM protocols additionally support dynamic data arrangements in memory locations. Since a binding to a memory location can create a replacement, the protocols must have schemes to find backup space for the replaced data. (3) If we allow data to be replicated in more than one memory location, the protocol must keep the data coherent. That is, if the processor wants to update the shared data by a write reference, all of the shared copies must be either updated or all but one must be invalidated.

4.1 Existing protocols

If the addresses for memory locations are assigned statically as is the local memory in CC-NUMA, we can significantly simplify the DSM protocol. However, many memory locations, such as those in the caches, allow many addresses to be mapped to each memory location dynamically. Figure 4.1 shows some possible DSM protocols which support dynamic data arrangements. Most existing small-scale multiprocessors use some kind of broadcasting

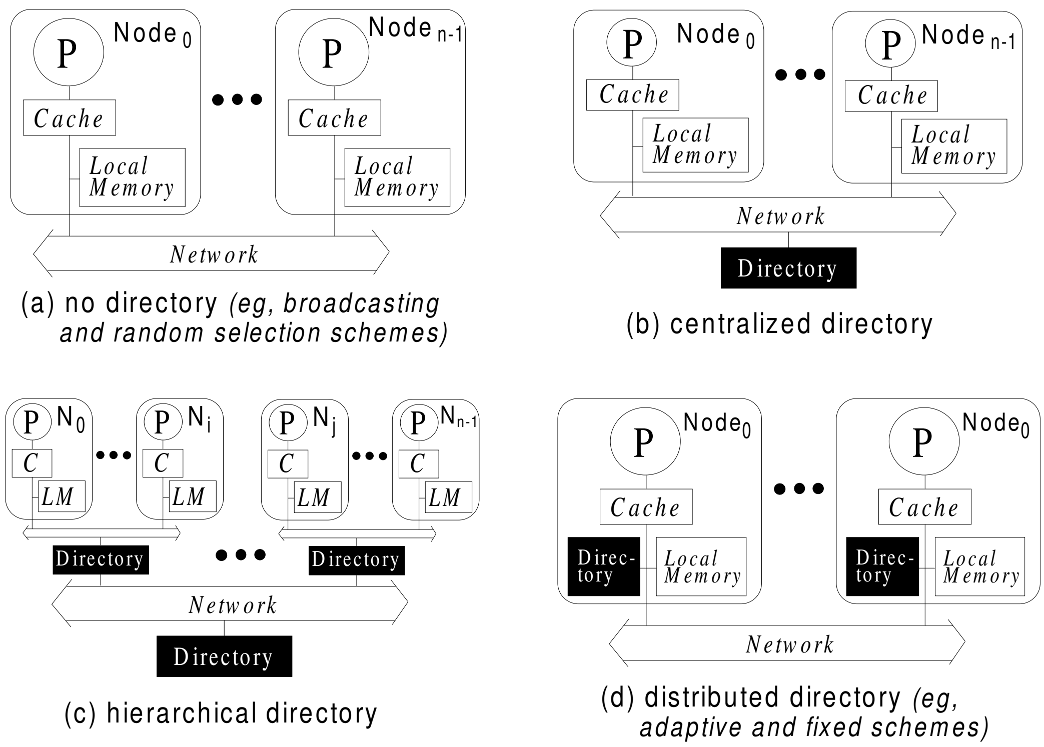


Figure 4.1: Directory-based DSM protocols

scheme to support the above three functions. The broadcasting schemes do not maintain information about data arrangement in the memory locations (see Figure 4.1 (a)). Instead, they require every system node to snoop on a broadcasting medium, such as a bus, to check the network transactions transmitted on the medium [AB84, Goo83, SS86]. An advantage of the broadcasting schemes is the short length of the critical path of a network request, i.e., a network request can be completed in two network traversals. However, contention in the broadcast medium can become very severe when the system size increases because each network request must be serviced sequentially. Thus, snooping protocols are not very scalable to a system that has a large number of nodes.

To overcome these limitations in broadcasting schemes, networks which support point-to-point communications between nodes have been considered desirable in large-scale DSM multiprocessors. However, if a DSM protocol randomly selects a node to carry out the three functions, the DSM multiprocessors may perform even worse than the broadcast-based multiprocessors. Thus, most protocols which use point-to-point communications have storage, called a *directory*, to maintain some information about the data arrangement. If only the links needed for communication between nodes are utilized with the directory, the requirement for network bandwidth can be minimized. Thus, directory-based protocols can support large-scale multiprocessors because of the distributed nature of the protocols.

For efficient point-to-point communication, a number of directory protocols [CF78, CKA91, JLGS90, Sim95, Tan76] have been proposed since late 1970's. Information about the nodes sharing a line is kept in a directory entry. Thus, unlike in the broadcasting and random selection protocols, only the links needed to invalidate or update the shared copies are used for network communications. The first directory protocol proposed in [Tan76] centralizes all directory information in a single source (see Figure 4.1 (b)). This protocol has the same problem, contention for a single resource. There are many ways of distributing the directory information. In a hierarchical directory structure, the processing nodes which contain data are the leaves of the hierarchical structure (see Figure 4.1 (c)). Each directory node in the hierarchical tree structure maintains state information about all data in the processing nodes below it. Upon a node miss, a request propagates up the hierarchy until it arrives at a directory node whose information indicates that a processing node below it has a valid copy. At that point, the request propagates down the hierarchy to the node that has the requested copy [SJG92].

Another distributed scheme is keeping "probable" owner node information for all system data in each node [LH89] (see Figure 4.1 (d)). Thus, when the requested data is missing in a node, a request is sent to the "probable" owner node pointed to by the node's directory. If the "probable" owner node does not have the ownership copy, the request is forwarded to the next "probable" owner node pointed by that node's directory. When the request reaches the node that has the ownership copy, it will returned to the requesting node with the valid copy. Directory information is updated appropriately whenever a node receives a request. For example, the "probable" owner node information about the requested data is changed to the requesting node in each node visited if the request type is a write. Although this adaptive scheme shows the possibility of distributing directory information dynamically, it requires large storage and increases protocol complexity.

A full-map directory [CF78] distributes the directory information along with distributed main memory (see Figure 4.1 (d)) to alleviate contention at a centralized directory. Directory information for each physical address is maintained in a fixed node. When a request wants to know where the requested data is, the request is sent to the predetermined (or home) node in which the necessary information for the request is provided by the directory. This scheme is simple compared to the other distributed schemes. Thus, many researchers have delved into the this type of distributed directory schemes for scalable multiprocessors.

As mentioned in the previous chapter, in COMA, an overhead results from memory copy replacements. When a miss happens at an AM, a line may need to be replaced to make space for the line coming from a remote node. Since the replaced line may be the last valid copy in the system, to avoid loss of data, the line must be relocated to some other AM that has space for it. Bus-based COMA [LQK94, LK95, LD96] can solve the problem by using the broadcasting feature of the bus [CKL99]. However, use of such bus is expensive and not scalable. In a directory-based COMA multiprocessor, finding a node whose AM has space for the replaced block can create a serious problem [Joe95]. Although a hierarchical directory structure in COMA can provide a deterministic searching mechanism [HHL92], higher network latency is observed [SJG92]. The hierarchical directory structure increases the memory requirement overhead. Also, since network requests are relegated to fewer high-level directories, data access latency can further be increased due to contention.

4.2 System organization

Although there are several ways of designing a machine for CC-NUMA, COMA, or DYMA, we will choose one hardware organization among them to compare performance. Figure 4.2 shows a datapath block diagram for a DYMA and a CC-NUMA machines. The hardware organization assumes a directory-based write-invalidation protocol. The main system resources in the organization are the processor, the cache, the local memory, the directory, and the interconnection network. The control mechanism designed in this section makes each system resource function individually, i.e., the control for the state of a resource does not depend on the states of the other resources but depends on the current state of the resource and the incoming request. Also, the control for the datapath depends only on the request on the datapath.

4.2.1 Datapath requests

In this discussion of requests on the datapath, we use the following naming conventions for the system components involved in network communication. A *requester* node is the node that contains the processor originating a given request, while a *home* node is the node containing the directory information of the requested line. A *remote* node is any node that is not the requester. An *owner* node is the node which has the copy designated as the ownership copy among the multiple copies of the requested line. If the requester node does not have a valid copy of the line requested by the processor, the (remote) owner node is responsible for replying to the request. Thus, the directory entry for a line should have information about the owner node as well as the shared nodes. The ownership also insures that at least one copy of a memory line is maintained in the system at all times.

If the data requested by the processor is missing in the requester node, the data request

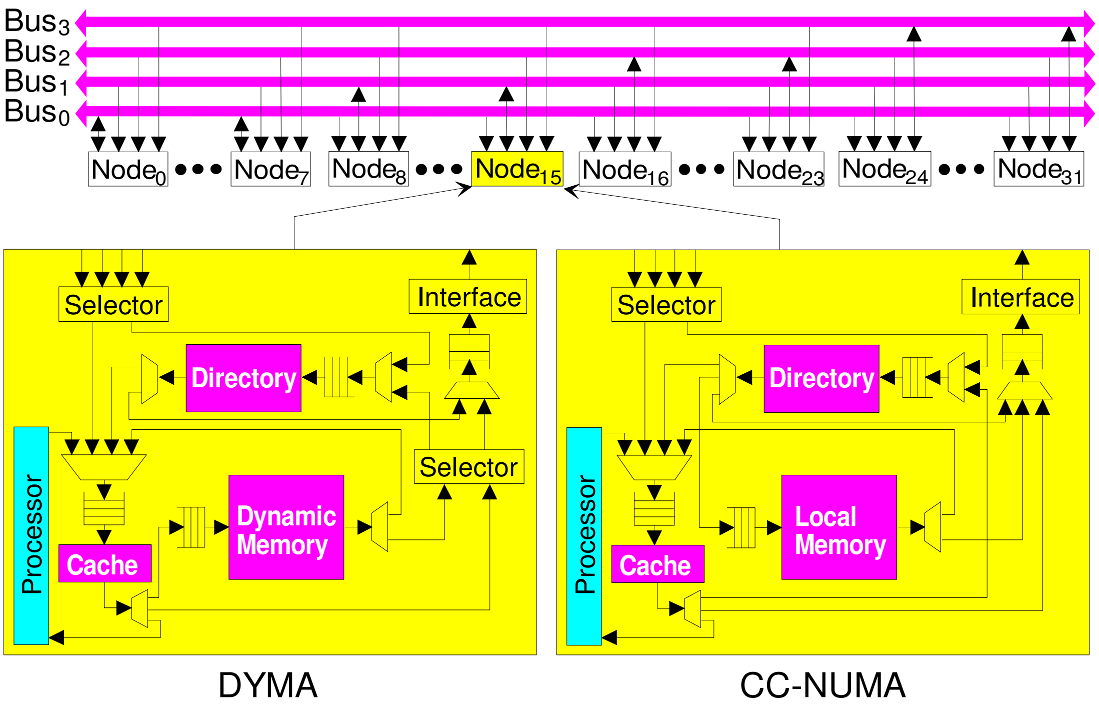


Figure 4.2: Datapath block diagrams for DYMA and CC-NUMA
In the figure, 32 nodes are connected with 4 buses. Thus, a bus serves the network requests generated from 8 system nodes.

is delivered to the home node to determine the current owner node. If the request type is a read, let us call the data request a *read request*. If it is a write, the data request is delivered to the home as a *read-exclusive request* to indicate that the processor wishes to modify the line. The home node forwards the data request to the owner node, and the owner node returns the requested data to the requester node.

When a valid copy needs to be replaced from the requester node, a *relocation* request is delivered to the home node if the replaced copy has ownership. The relocation request is further transmitted to a node where the replaced copy can be saved. Now, the node becomes the owner of the line. If the replaced copy does not have ownership, a *replacement-hint* request is initiated to update the information of the directory at the home node. Unlike the relocation request, the replacement-hint request is not forwarded to a node since the request is just needed to maintain correct directory information in the home node. In fact, some machines such as DASH do not generate the replacement-hint request since data consistency and integrity can be guaranteed without the request. When a processor writes to a shared copy, all shared copies except one have to be invalidated under write-invalidation coherence control. This coherence action is accomplished by launching *invalidation requests* on the datapath.

In CC-NUMA, the home node of a line address has dedicated space for the line at its local memory. Thus, the state information (validity) of the memory copy can be maintained in the directory. Then, all cache misses must go to the home directory first even if its local memory has valid copies. The home node is usually assumed to be the owner node if the copy at the local memory is valid. Also, all of the copies replaced from the cache are sent to the home node, where the replaced copies can be written back into the local memory. These schemes in CC-NUMA can reduce data access latency and the amount of network traffic. In COMA and DYMA, the home node of a line address does not have dedicated space for the line at its local memory. Thus, all cache misses first go to the local memory of the requester node. Also, all of the copies replaced from the cache can be kept in the local memory of the requester node.

4.2.2 Data states

To reduce the number of requests on the datapath, several states of data are essential. In order to prevent a request from being generated at every write and to keep data from becoming inconsistent, the state of a line should have *privateness* and *validity*. Also, to avoid loss of a memory line and to minimize the attempts to relocate a replaced copy, *ownership* should be included in the state of a line. If all of the lines discarded from the cache attempt to be relocated in the DM of DYMA, extra overhead can occur. Thus, the state of a cache copy has *inclusiveness* which indicates whether the same address copy is in its DM or not.

We tried to optimize the number of states of data for each architecture without increasing the number of requests for inter-/intra-node communication. This is possible because one state can represent more than one property under a memory organization. Four states for the cache and two states for the local memory are assumed in CC-NUMA. Since a write access to a clean cache copy generates an invalidation request in CC-NUMA, we do not have to differentiate the two clean states, Exclusive (E) and Shared (S), in the MOESI model [SS86]. Generally, a valid bit in each local memory frame is used for both indicating the validity of the copy in the frame and representing ownership of the copy. In COMA, since under our assumption a remote access can get a copy of a modified line from the remote cache without writing the copy back to the AM, all five states in the MOESI model are required for each cache line. To the best of our knowledge, no open architectures including the two-level cache architectures has ever been designed using this type of the coherence control. The AM in COMA has four states since the private state data in the AM has ownership as well. DYMA has 12 and 4 states for the cache and the DM respectively [KL96]. The data in the cache has validity, ownership, privateness, and inclusiveness. Also, to make the cache and the DM function individually, more states are needed. This is the reason why DYMA has a large number of states for cache copies. Data state transitions for DYMA will be described in more detail in a later section.

4.2.3 Interconnection network and traffic

The interconnection network is designed with a multibus. One bus serves the network requests generated from a part of system nodes (eight nodes in Figure 4.2), while all of the nodes can get requests directly from the bus. Unlike the traditional snooping bus where all of the nodes in the system snoop their states and tags for every bus request, here each node selects only the requests which are sent to its node, and the network can serve multiple requests (up to the number of buses) simultaneously like general pointer-to-pointer communication networks such as the mesh, hyper-cube, crossbar switch or multi-stage switch networks.

To extend this structure to a very large-scale machine, increasing the number of buses is one approach. The other approach is connecting several multibuses with a bridge. The reason we chose this kind of network is to take advantage of the simplicity and efficiency of the bus even though arbitration among the nodes generating network requests in the bus is required. If the system nodes are placed within a very short distance, the multiaccess bidirectional broadcast bus system seems to be optimal in terms of node-to-node transmission time and network utilization. It is beyond the scope of this thesis to evaluate whether the multibus network is favorable to a certain memory architecture or not. Indeed, the protocol used in our organization works correctly for any type of network as long as network routing is static, which means the FIFO order must be guaranteed between the same source and destination nodes.

Figure 4.3 depicts a view of network traffic at a reference event in a DSM architecture. The three axes in the figure represent the characteristics of an event. The axis of the data location tells us the location of the line requested by the processor. The axis of the data state has two aspects: (private and shared) states of the requested line and (invalid, non-owner, and owner) states of the replaced copy from a node. The three boxes in the figure represent the types of network traffic: data traffic, replacement traffic, and coherence traffic. Thus, network traffic types can be decided based on event characteristics. Data traffic is needed when a processor has access to a line but a copy of the line exists neither in the cache nor in the local memory (node miss). The line must be copied from a remote node by launching a data request to the network. Since in most cases the home node of a line does not have

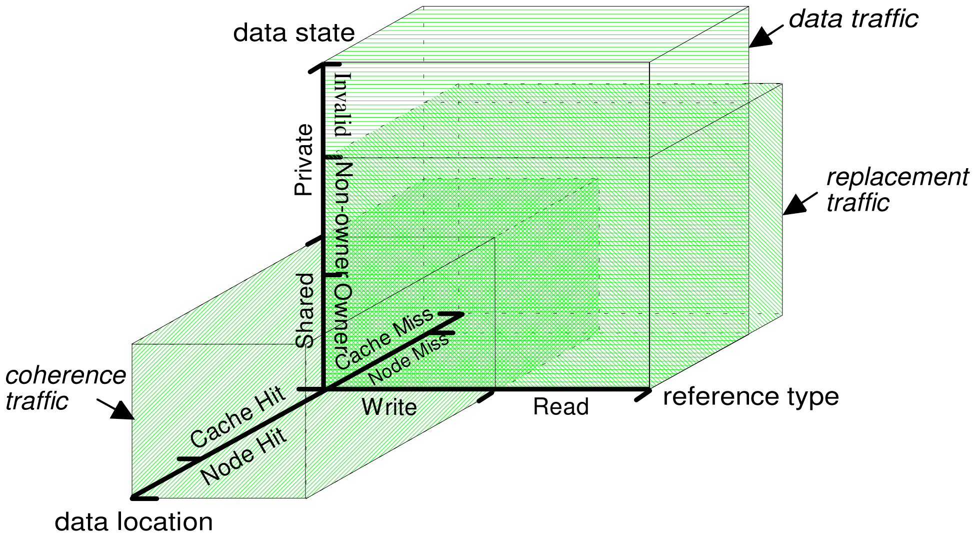


Figure 4.3: A view of network traffic at a reference event

Three axes in the figure represent event characteristics which influence network traffic.

a valid copy of the line in COMA and DYMA, data traffic usually requires three network traversals: a traversal from the requester to the home node; a traversal from the home node to the owner node; and a traversal from the owner node to the requester node. While the data requests for the first two traversals have only head information without data, the reply for the last traversal carries data as well as head information. In CC-NUMA, the chance of having only two network traversals is high since the home node of a line always has a copy of the line.

Replacement traffic is divided into two cases (relocation request and replacement-hint request) depending on the ownership of the replaced line. If the selected line has ownership, it is written back to the pre-allocated location of the memory in CC-NUMA while in COMA and DYMA it needs to be relocated to a remote node. Thus, a relocation request generally requires just one network traversal in CC-NUMA while it usually takes more then one network traversal in COMA and DYMA. Since the replacement-hint request's sole purpose is to maintain correct information in the directory about nodes sharing a memory line, it usually takes one network traversal. If a cache miss hits at the local memory, no line needs to be selected for replacement in COMA since COMA has the inclusion property while cache misses can generate replacement traffic in both CC-NUMA and DYMA.

Invalidation requests on the network are considered as coherence traffic. If copies of the shared line exist only in the requester node and the (remote) home node, two network traversals are required. Otherwise, at least three network traversals are required. The first case can happen frequently in CC-NUMA since the local memory of the home node always has a copy even when the copy is referenced by just one processor. Invalidation requests have head information only. The number of invalidation requests created for a coherence action depends on the number of nodes sharing the line at that time. Here, note that if a line missing at a node is shared by several nodes and the type of reference to the line is a write, invalidation requests are sent to the non-owner nodes to invalidate the shared copies.

4.3 CC-NUMA protocols

Since memory space for data is determined at a page fault, the relocation problem for the data replaced from the cache can be solved by simply copying the up-to-date data into the local memory. Thus, most studies have focused only on the coherence problem for shared copies. Before studying the protocol for a CC-NUMA machine, let us look at the directory organization which can minimize directory size.

4.3.1 Directory size

The storage requirement of a basic full-map directory [CF78] in *each node* is proportional to $O(mn)$ for a system with n nodes, where m is the number of frames (or locations) allocated in *each local memory.* This growth makes the full-map directory very expensive in a large DSM multiprocessor. Since the cache size is much smaller than the local memory size, and data sharing is often limited to a small number of processors, most of the bits in the full-map directory are wasted. One key observation here is that the number of directory bits set at any time is bounded by the total number of lines in the caches, regardless of how the data is shared.

To take advantage of this observation and to improve the scalability of the directory size, some pointer-based directory schemes have been proposed. For example, the size of the chained directory in each node as proposed in [JLGS90] grows as $O(c \log_2 n)$, where c is the number of frames in each cache. This is much smaller than the $O(mn)$ in the full-map directory. However, the chained directory significantly increases protocol complexity and data access latency because directory information about a line is distributed over the cache memories in the form of linked lists. Figure 4.4 shows some chained directory protocols. In the figure, address x is shared by Node_i, Node_j, and Node_k, and the nodes are connected either by a singly linked list (Figure 4.4 (a)) or by a doubly linked list (Figure 4.4 (b)). In the singly linked list, when the cache copy replaced from Node_j arrives at the directory of the home node, the directory should send a network request to update the node_i's link. The

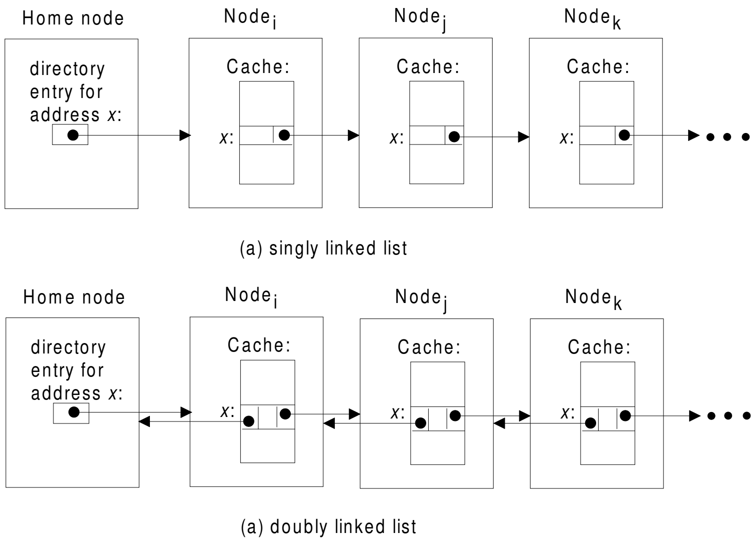


Figure 4.4: Examples of chained directories

doubly linked list can remove the directory overhead by directly delivering network requests to Node$_i$ and Node$_k$ from Node$_j$. However, it is obvious that the chained directory protocols increase the number of network requests. Further, the distributed directory information for one line causes high protocol complexity. For example, there is no simple way to handle the situation when Node$_i$ generates an invalidation request before a replacement-hint from Node$_j$ arrives at Node$_i$.

To solve the problems caused by the complexity of distributed directory information about a line, some other pointer-based directories have been proposed [ASHH88, CKA91, Sim95]. The limited pointer directory [ASHH88] tries to reduce the directory size by limiting the number of pointers allowed for each line in the main memory. An expensive broadcasting scheme is used if sharing a line exceeds that limit. The LimitLESS scheme [CKA91] reduces the broadcasting overhead by overflowing the pointers to full-map bit vectors. The dynamic pointer allocation scheme [Sim95] maintains a shared pointer pool of a fixed size. A linked list of pointers is maintained for each line which has a copy in the caches. Each of the pointers in the linked list points to a node containing a copy of the same line.

Figure 4.5 shows some linked lists of the lines in a node. The linked lists show that nodes 110_2 and 101_2 have a copy of the line 1100101_2, and node 110_2 also has a copy of the line 1100110_2. When the number of pointers needed exceeds the total number of pointers in the shared pointer pool, some pointers will have to be selected and “victimized” by invalidating their corresponding cache copies, hence, freeing the pointers for new assignment. This is unavoidable because it is not guaranteed that a pointer is eventually freed by another

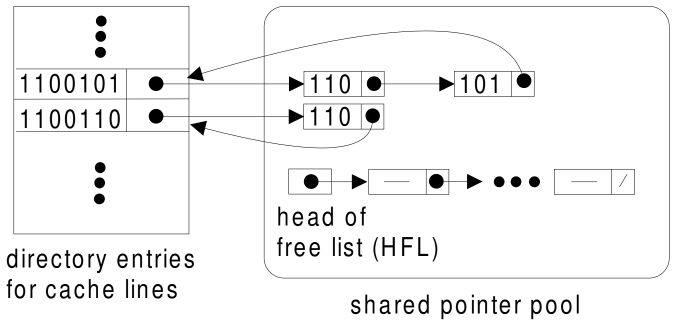


Figure 4.5: Some linked lists in a directory

request. Selecting a victim pointer increases protocol complexity. For instance, the protocol has to learn the address for the victim pointer to send an invalidation signal. Also, different coherence actions are required depending on the ownership of data at the cache pointed to by the victim pointer. Invalidating a valid copy in the cache drops the cache hit rate as well. The protocol should cope with several exceptional cases to maintain the structure of the pointer pool correctly. Therefore, performance could suffer significantly if the pointer pool is too small, and the pointer overflow occurs too often. Heuristics shows that the pointer overflow could be rare with a pointer pool whose size is less than 7% of the memory size. However, the number of pointers still needs to be much (8 to 16 times) larger than the number of frames in the cache [SH91]. Further, if the possibility of overflowing pointers is not zero, the directory protocols must be able to deal with the issue of the pointer overflow, which could be critical for system stability. Our proposed scheme tries to avoid most of those problems.

The DASH [LLG+90] and the Origin 2000 [LL97] are based on a full-map directory design. The commercial machine NUMA-Q [LCS98] uses the chained directory. The LimitLESS scheme is implemented in the Alewife machine [ABC+95]. Since the existing processor is responsible for handling the pointer overflow in Alewife, the data processing speed at the processor could be slow if the limit for sharing a line is frequently exceeded. The dynamic pointer allocation scheme is implemented in the FLASH machine [KOH+94] where a special piece of hardware is dedicated to the management of the pointer pool.

Before proceeding to the proposed scheme, we will see how the line address is interpreted for the cache. Figure 4.6 shows the two examples of indexing cache sets. In the figure, the address field I_l specifies the line address, I_p specifies the page number, and I_b specifies the byte offset in a line. To select a cache set for a line address, the address field I_s is used. The I_s field is highlighted as the shaded address field in the figure. The remaining address bits in the non-shaded address field, whose length is $(I_l - I_s)$ bits, are used as an address

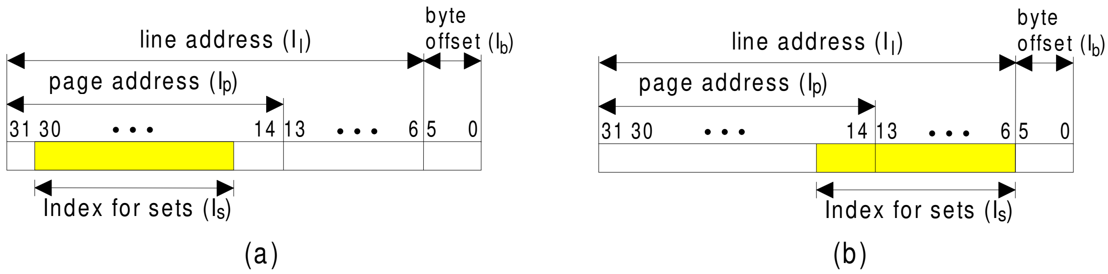


Figure 4.6: Relationship between page addresses and cache sets

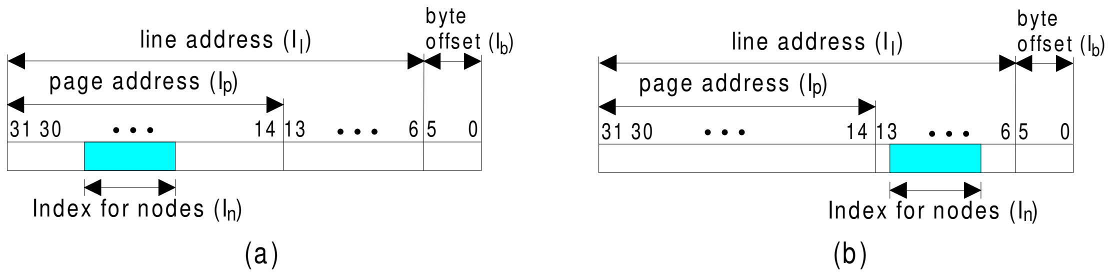


Figure 4.7: Relationship between page addresses and home nodes

tag in the cache. For a direct-map cache, each cache set has at most one valid line. It can be up to a lines in the a-way set associative cache. For fully associative caches, there is no I_s field. The address mapping to the cache depends on the I_s field. If the I_s field is fully overlapped with the I_p field (see Figure 4.6 (a)), then the whole lines in a page are mapped to the same cache set. This can cause frequent conflict misses in the cache set since the spatial locality of the page is not utilized well in the cache. Therefore, as shown in Figure 4.6 (b), I_s should be located far away from I_p for good performance.

Another interesting address field is the I_n field for the node number as shown in Figure 4.7. The I_n field is used to find the home node of a line address. Since each home node has a dedicated space for a line address in CC-NUMA, the data layout in the local memory depends on the I_n field. If I_n is located in the field I_p (see Figure 4.7 (a)), all of the lines in a page are allocated in the same local memory. This is the traditional data interleaving scheme. With this data interleaving scheme, bulk mode access to the local memory can be utilized since the unit of memory access is a page. Another advantage of this is in complexity of the I/O protocol. Since a whole page is the unit of a memory operation, the initiation and the termination of the operation can be easily detected. Also, if all of the lines in a page are frequently referenced by a single node, local memory hit rates could increase by locating the whole page in the node. However, this scheme has some weaknesses

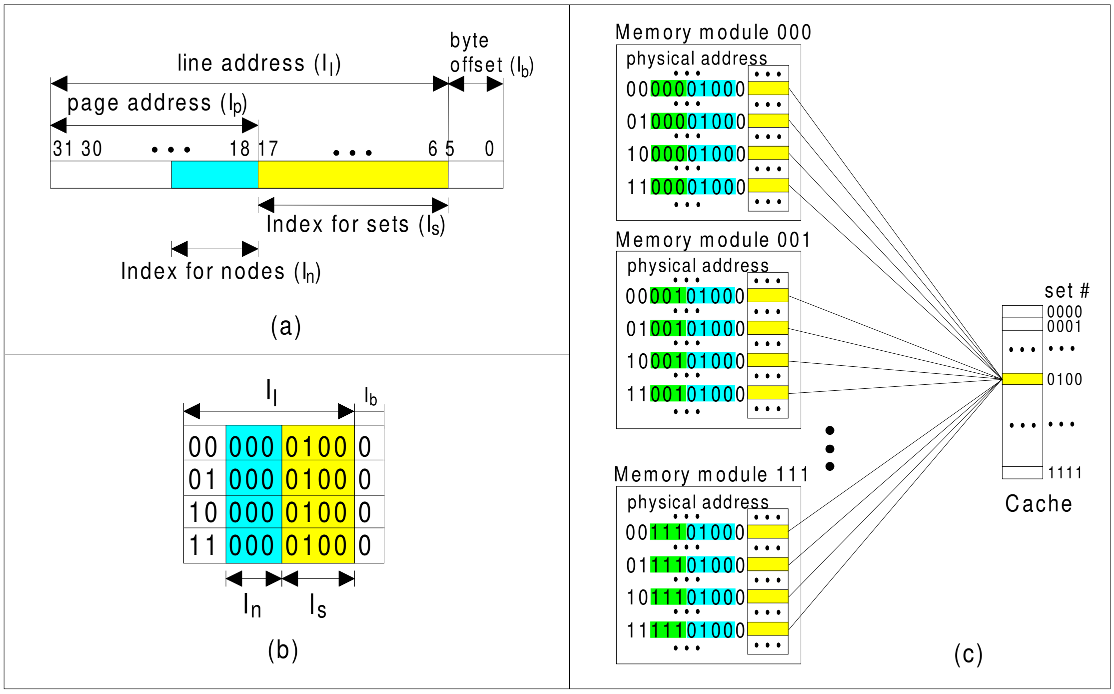


Figure 4.8: An interleaving scheme on local memories with large-size data
(a) A general data interleaving scheme on memory modules; (b) An example to show all of the lines in the local memory 000_2 which are mapped to the cache set 0100_2 in the cache; (c) Every local memory has 4 lines mapped to the cache set 0100_2.

not shared by the other interleaving scheme, which puts I_n outside of I_p (see Figure 4.7 (b)).

The latter scheme does not allocate a whole page in a single local memory. The page is divided into small parts of lines and interleaved across the nodes. Thus, the time for handling a page fault can be short since the memory operations for a small part of a page happen in multiple nodes concurrently. This can reduce frequency of contention at the local memory if several remote nodes want to reference different lines in a page at the same time. However, this scheme requires additional synchronization overhead since a master controller has to know the termination of each memory operation. Also, since the size of data for the memory operation is smaller than a page, bulk mode access to the local memory can be less effective.

Now, let us look at the relationship between the I_n and I_s fields. To simplify our presentation, first assume the unit of our data interleaving is one page, i.e., we allocate one page to each local memory in a round-robin fashion across all nodes. Figure 4.8 (a) shows the assignment of address fields in a physical byte address using such an interleaving scheme. Let us further assume that the cache is direct-mapped and that its size is the same as the page size. Using such an interleaving scheme, an entire page (or any portion of a page) can reside in any cache.

If we study this data interleaving scheme more carefully, we can observe that, $2^{I_l - I_s - I_n}$ (or $2^{I_p - I_n}$) of the lines in each local memory are mapped to the same cache set in each cache. To see this, in Figure 4.8 (a), the node number of a particular local memory and the set number of a particular cache set are highlighted as the shaded address fields. For a *selected node* and a *particular* cache set, the address bits in those shaded fields are fixed. Only the remaining address bits in the non-shaded address field, whose length is $(I_l - I_s - I_n)$ bits, can be varied. Each of those addresses corresponds to a line in the selected local memory which is mapped to the same cache set in the cache.

Figure 4.8 (b) shows an example with a 10-bit byte address and a 2-byte line. In the figure, the selected local memory (I_n) is 000_2, and the selected cache set (I_s) is 0100_2. The table lists all of the lines in the local memory 000_2 that are mapped to the same cache set 0100_2 in the cache. Since $(I_l - I_s - I_n)$ has two bits, we have four different lines mapped to the same cache set 0100_2. Figure 4.8 (c) shows those four lines in the local memory 000_2. Figure 4.8 (c) also shows that all the other local memories will have four lines mapped to the same cache set 0100_2. There are a total of 32 such lines since there are 8 local memories in our example. It is important to note that, at any instance of time, only one of these 4 lines in each local memory (32 lines in total) can reside in a *particular* cache. Bringing in any of the other lines in this group will create a conflict miss in that cache. With this important observation, we can calculate the maximum number of pointers needed in the pointer pool for each local memory using the dynamic pointer scheme [SH91] without causing pointer overflow.

In each local memory, the I_n field is fixed, and only one line in the group specified in the $(I_l - I_s - I_n)$ field can reside in any cache, and there are 2^{I_s} groups. Therefore, we need a maximum of $(2^{I_n} \times 2^{I_s})$ pointers in each local memory [1]. This worst case happens when all of the caches have all of their lines from the same local memory. Notice that $2^{I_s} = c$ is the total number of frames in each cache, and $2^{I_n} = n$ is the total number of system nodes. The maximum number of pointers required in the pointer pool without "victimizing" any existing pointer is $O(cn)$ (or $O(cn \log_2 n)$ in bits) for each local memory, which grows proportionally to the cache size. In contrast, the basic full-map directory scheme grows proportionally to the local memory size $O(mn)$.

[1] If the cache set associativity is a, at most a lines in the group can reside in any cache. Thus, the maximum number of pointers is $a \times 2^{I_n} \times 2^{I_s}$.

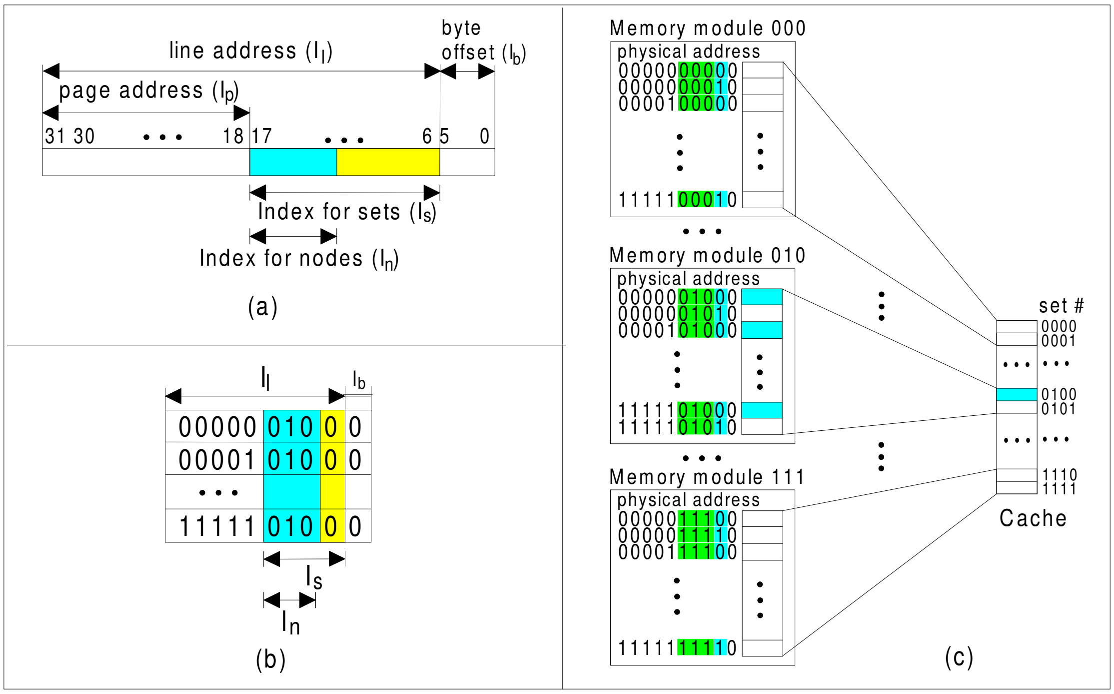


Figure 4.9: An interleaving scheme on local memories with small-size data
(a) An improved data interleaving scheme on local memories; (b) An example to show all of the lines in local memories (32 in each local memory) which are mapped to the cache set 0100_2 in the cache; (c) Our proposed data interleaving scheme will place all 32 lines in the same local memory 010_2.

To see if we could reduce the number of needed pointers even further, let us change the interleaving scheme as follows. As shown in Figure 4.9 (a), we now interleave only a portion of a page specified in the address field $(I_s - I_n)$ across all of the local memories. The node number field I_n occupies the most significant portion of the I_s field (as opposed to being on the outside of the I_s field as in Figure 4.8 (a)). In this interleaving scheme, a page is now spread across all local memories instead of being placed entirely in one local memory. Figure 4.9 (b) shows the same example as in Figure 4.8 (b), but now we have 32 lines in local memory 010_2 that are mapped to the same cache set 0100_2 in the cache. In a sense, the new interleaving scheme places all 32 lines, which used to be spread across all 8 local memories, as shown in Figure 4.8 (c), in the same local memory 010_2, as shown in Figure 4.9 (c). Similar to the earlier argument, only one of these 32 lines can reside in cache set 0100_2 at any instance of time. Bringing in any of the other lines in this group will create a cache conflict miss.

Based on this observation, we can again estimate the maximum number of pointers needed in each local memory as follows. The first interesting observation is that, since the field for node number I_n is part of the address field for the cache set number I_s, all of the lines in a particular local memory (in which the field I_n has a fixed value, i.e., its node number) can only occupy a specific portion of the cache (the part where the I_n field has the value of its node number). In other words, the addresses in each local memory can no longer be mapped to the entire cache as in the previous interleaving scheme, but only to a portion of the cache. The cache is, in effect, being divided into equal portions: one for each local memory, as shown in Figure 4.9 (c). Because of such a reduction in the mappable cache space for each local memory, more lines from a local memory are being mapped to the same cache set. Again, from Figure 4.9 (a), for a selected local memory and a selected cache set (in which the value of the I_s field is fixed and I_n is a part of I_s), we have $2^{I_l - I_s}$ (or 2^{I_p}) lines in that local memory being mapped to the same cache set. It is a much larger group than the group in Figure 4.8 (a), but the number of groups in each local memory decreases from 2^{I_s} to $2^{I_s - I_n}$. The maximum number of pointers needed without having to "victimize" any pointer in each memory module is now $2^{I_n} \times 2^{I_s - I_n} = 2^{I_s} = c$, i.e., it is proportional only to the cache size c (as opposed to cn in the previous case)[2].

It is very important to note that this reduction in the mappable space in each cache for each local memory will not alter the cache miss ratio in the cache. This is because, even though we use different data interleaving schemes for the local memories, we did not change at all the original data layout within the physical address space, which the cache data mapping is based on. That is, the position of the I_s field is unchanged. The only thing that has changed is the location of the data in the local memories, but the cache behavior is not changed. However, different interleaving schemes will incur differing degrees of memory conflict in the local memories. Extra network traffic to the remote memory, which might have been local in the original interleaving scheme, may also result. In fact, our simulations show very little difference between the two address interleaving schemes, since neither scheme has anything to do with locality of reference. We will discuss these issues using simulation results in Chapter 5. Nevertheless, cache miss ratios should remain unchanged in each cache.

Using a similar strategy, there could be many ways to interleave the data among the local memories, each with different consequences to the total number of required pointers and to memory performance. One possible interleaving scheme is to straddle the I_n field between the I_p and I_s fields as shown in Figure 4.10 (a). These schemes tend to increase the required pointers, but our simulations show that these increases have very little impact on overall performance (see Chapter 5).

[2] If the set associativity is a, the maximum number of pointers is $a \times 2^{I_n} \times 2^{I_s - I_n} = a2^{I_s} = c$.

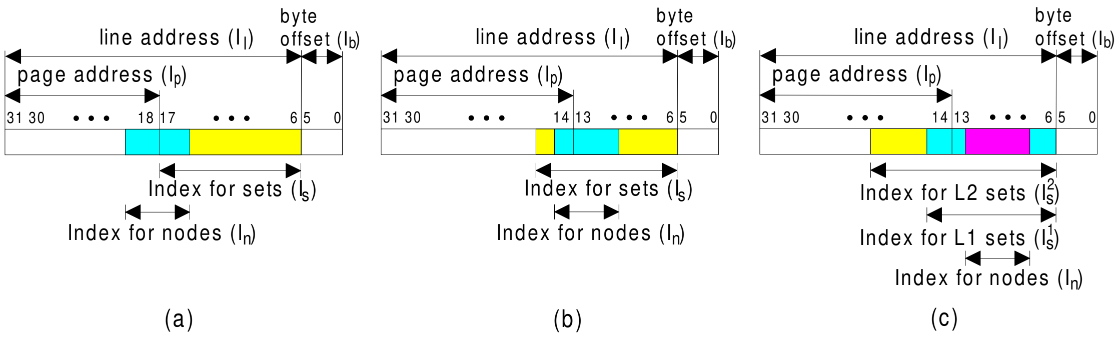


Figure 4.10: Other data interleaving schemes on local memories

Another possible design consideration is the size of a page, which determines the length of the I_p field and its relationship to the cache size. The page size is usually smaller than the size of the L2 or L3 cache, and caches may have a set associativity larger than one. However, whatever the layout of I_p and I_s is, the size of the pointer pool depends only on the location of the I_n field relative to the I_s field. Figure 4.10 (b) shows the case in which the I_n field is a part of the I_s field, but is straddled over part of the I_p field.

For systems with multi-level caches, if all of the caches at different levels satisfy the inclusion property, then we only need to consider the cache level which is farthest away from the processor, and which is usually the largest in size. All of the discussions so far should still apply. However, if the inclusion property is relaxed [Jou90, KL96], then the maximum number of pointers required in each pointer pool will be the sum of the total lines at all levels if the I_n field is included in the I_s field of the L1 cache (I_s^1). That is, if the total numbers of the lines in the L1 and L2 caches are c_1 and c_2, respectively and the I_n field stays within the I_s^1 field as shown in Figure 4.10 (c), the maximum number of pointers needed in each pointer pool will be $c_1 + c_2$. Again, different interleaved data sizes will have different consequences. However, studying all possible variations is beyond the scope of this paper.

4.3.2 Directory organization

The management of the pointer pool used in our proposed scheme is very similar to the one proposed in [Sim95]. The basic organization is shown in Figure 4.11. Each pointer will have two major fields: One stores the node number that has a cached copy of the line, and the other keeps a link which points to the next pointer forming a non-circular singly linked list (see Figure 4.11 (b)). A free pointer list is maintained and its head is linked by a special register, called *head of free list* (HFL).

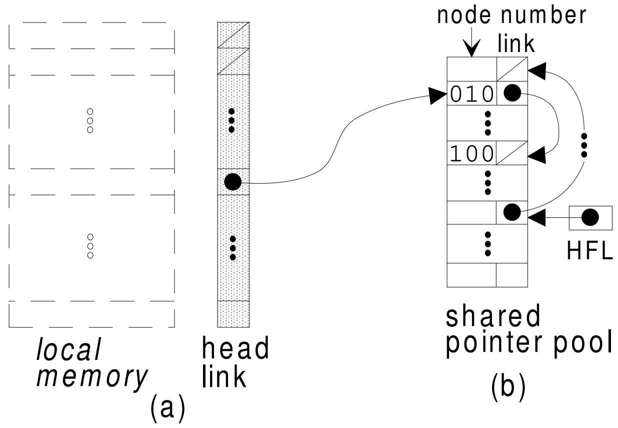


Figure 4.11: Directory organization in CC-NUMA

The number of pointers in one linked list (except the free list) is limited to the system size, n. Since there are c pointers in the pool, the minimum number of the linked lists in the pointer pool is $\frac{c}{n}$. That is, the number of line addresses whose directory information can be kept in a pointer pool can vary from $\frac{c}{n}$ to c. To facilitate access to the linked list associated with a line address, each line address in the local memory could have a link (shown as "head link" in Figure 4.11 (a)) which points to the head pointer of the associated linked list.

A link is of the size $\log(c)$ bits, which is very small compared to the line size. However, the total number of the links can grow proportionately to the number of memory lines if each line address in the local memory is associated with such a link (see Figure 4.11 (a)). Even though this storage overhead could be very small in practice, we could hash the line address to find the associated list in the pointer pool and avoid the extra overhead. This requires one more field in the head link to store the address tag of the associated linked list. Also, it could make pointer access a little more complicated with some extra time overhead if conflicts occur during the hashing. However, it allows directory size to scale only with cache size instead of with memory size. In fact, we can optimize the hashing function by using the fact that only $a \times n$ copies among $2^{I_n - I_s}$ lines in a group are cached at any time, where a is the cache set associativity. Thus, in the worst case, the number of the hashing conflicts can be limited to $a \times n$.

As assumed in CC-NUMA, the state information (valid bit) which indicates whether a memory copy is valid or not can be included in the directory. If a memory copy is not cached, the memory copy should be valid. Thus, the valid bits can be associated only with the linked lists for cached copies.

When a valid non-ownership copy is being replaced from the cache, a replacement-hint request might be generated in the pointer-based directories [Sim95]. In a full-map directory scheme, the replacement-hint request is not always necessary. Even though the bit corresponding to the replaced copy will remain set in the full-map directory, there would be no significant side effect as long as data consistency is guaranteed. However, in the pointer-based scheme, the pointer corresponding to the replaced non-ownership copy will become "stale," and not returned to the free list. If the number of such "stale" pointers increases over time, the available free pointers will be depleted. To avoid such a situation, we need to force all of the replaced cached copies, either ownership or non-ownership, to inform the directory by generating requests, and free up the pointers associated with the replaced copies. The extra network traffic needed for this notification could be very minimal. This is explained in the following.

Another possible modification is needed when we try to replace a cache copy. We may want to fetch the new line first and perform the replacement later to minimize the processor stall time. According to scheme proposed here, the replaced line and the line to be fetched are mapped to the same cache set, i.e., they have the same home directory. However, if we only provide c pointers in the pointer pool as proposed (where c is the number of frames in each cache), we implicitly assume that the replaced line will free up the pointer first to allow the new line to use the pointer. If we reverse that order, it is possible that when the request for the new line arrives, it may find that there is no longer a free pointer available in the free list. Since this situation will occur only temporarily, a small number of extra pointers could be added to avoid such temporary starvation. When the directory receives a data request, all previous replacement requests from the same requester are completed and disappear from the directory. This is because we assume the FIFO order between the same source and destination nodes. Therefore, the maximum number of replaced copies whose pointers are not freed up at any time is equal to the number of nodes (n), assuming each processor generates one data request at a time (i.e., sequential consistency [AG96]). In this case, the total number of pointers needed without having to "victimize" any pointers is $c + n$. If the replaced copy is available at the same time when a cache miss occurs, the extra pointers can be eliminated by sending the data request and the replaced copy to the same home node together. Usually, the address of the replaced copy is available without increasing the cache access time. Thus, the extra network traffic needed for the replacement-hint request could be very minimal.

Each element in a linked list can have multiple pointers for node numbers and one link for the next element. This scheme might be effective, especially when the pointer pool is implemented with DRAM and an access to the DRAM handles a large chunk of bits. For flexibility, one might want to use a part of the local memory as the pointer pool and use micro-instructions to control the directory [ABC$^+$95, KOH$^+$94]. Then, the system can increase the size of the cache or the number of nodes simply by modifying the micro-instructions.

As mentioned previously, the I/O protocol for the proposed scheme needs to be different from traditional ones if the I_n field is not inside of the I_p field (see Figure 4.10 (b) or Figure 4.7 (b)). To collect the lines for the page replaced from main memory, and to distribute the lines in the new page loaded from I/O storage, the I/O controller has to send several requests to the home nodes of the lines. After learning that all the requests are successfully completed, the I/O controller can interrupt the processor.

4.3.3 Blocking directory protocol

In this section, we address two issues regarding the efficiency of directory-based protocols. The first issue focuses on two metrics: (1) the number of requests required on the *critical path* in a protocol; and (2) the total number of requests needed (including all data arrangement requests) to complete a memory reference. The critical path of a data access is the path with the largest number of requests on any possible path from the issue of a processor's request to the supply of valid data. Thus, it is directly related to data access latency. Some requests which are not on the critical path are necessary for a new arrangement of data. Those requests can increase the time for other data references due to possible contention for system resources. Thus, data access latency might be less sensitive to the second metric, but still be affected by the metric.

The second issue is serialization of memory accesses. The time between the issuance of a memory request from a requester node and its arrival at the home directory is usually nondeterministic because of various possible network conditions. Most directory-based protocols use the order in which memory accesses arrive at their home directory as the basis for serialization. In most existing protocols, when two requests to the same line arrive at the home directory at about the same time, one has to be blocked until the first request is completed — we call such protocols *blocking* protocols. Such protocols could affect the latency of memory accesses to the shared cache lines. Those shared cache lines could be the result of shared variables such as synchronization variables, or the result of false sharing [LYL87, TLH94].

The directory protocols proposed in the literature differ primarily in the way the data request is issued for a line whose valid copy is present in a remote node. One possible approach is to send a request to the home node, have the home node forward the request to the owner node, retrieve the requested data to the home node, and then have the home node return the data to the requester node. This flow of requests is shown in the simple protocol of

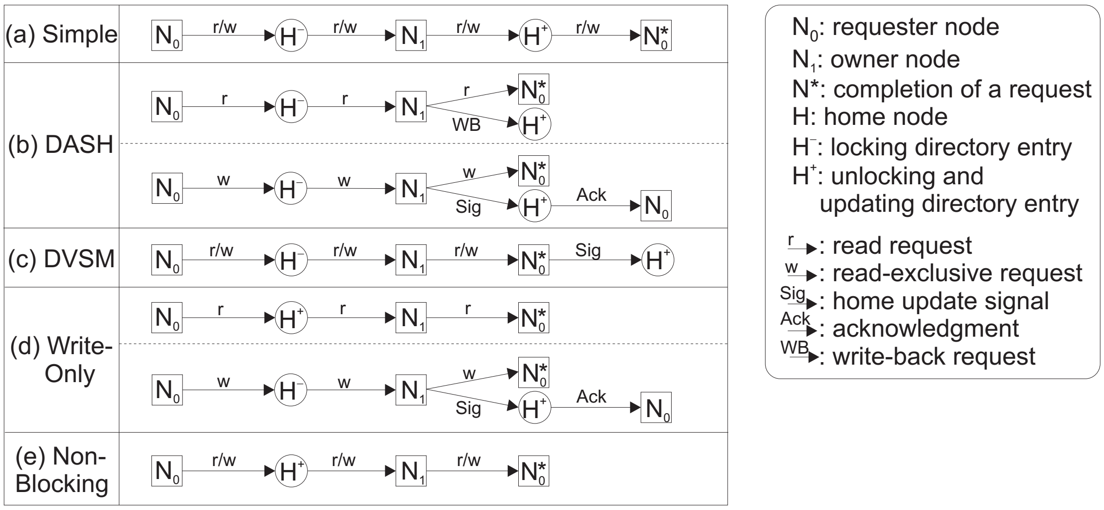


Figure 4.12: Flow of a data request in each directory protocol

Figure 4.12 (a). In the figure, N_0 is the requester node and N_1 is the owner node. When N_0's request is successfully forwarded to the owner node N_1 from the directory, the associated directory entry is marked to indicate that a request for the entry is being serviced (shown as H^- in the figure). Subsequent requests for the same line which arrive at the directory are suspended. A technique for the suspension is to buffer the subsequent requests at the directory. To eliminate buffering hardware, bouncing subsequent requests back to their requesters as unsuccessful requests is another possible technique. An unsuccessful request will force the requester to retry later. Both of these techniques require a lock bit in each directory entry indicating whether the entry is locked or not. When service is completed, the directory is unmarked while updating the directory information (shown as H^+ in the figure). The completion of N_0's request is denoted as N_0^* in the figure. After the completion, the requested line is in N_0 and is ready to be used.

The request flow of the DASH protocol [LLG+90] is shown in Figure 4.12 (b). Similar to the simple protocol, only the owner node can update the directory information. The owner node N_1, upon receiving a read request, generates two messages: a message containing a copy of the requested line is returned to the requester node; and a message (a write-back (WB) request) is sent to the home node to update its copy and directory information. Ownership of the line is returned to the home node. Upon receiving a read-exclusive request, the owner node N_1 sends a message containing a copy of the line to the requester node N_0 while dispatching a signal to the home node to update the directory information. After updating the directory information, the home node forwards an acknowledgement to the requester node N_0. The requester node N_0 is not allowed to replace the newly requested line until it receives the acknowledgement from the home node indicating the directory update has been completed. This is to avoid the race condition in which the requested line is replaced in N_0 and updates the home directory before the directory is updated by the signal from N_1 [LLG+90].

Instead of blocking the directory while it is forwarding a request to the owner node, the DASH protocol allows subsequent requests for the same line to be forwarded to the owner node as well. However, the requests will be returned to their requesters as unsuccessful requests if the remote owner node no linger has ownership of the requested line. As can be seen in Figure 4.12, the DASH protocol would reduce data access latency in the critical path from four to three transactions, as compared to the simple protocol. However, the total number of transactions increases from four to five for a read-exclusive request.

The DASH protocol is not applicable to COMA and DYMA since there is no dedicated space for the write-back request at the home directories of COMA and DYMA. The protocol for COMA-F [Joe95] replaces the write-back request with a signal which just updates the directory information. When the requester node retrieves the copy requested by its processor, the copy has ownership to avoid buffering subsequent requests which may arrive at the directory prior to the signal updating the directory information. Subsequent requests which arrive at the previous owner node will be returned with a negative acknowledgement to their requesters since subsequent requests cannot find the ownership copy. The protocol in COMA-F has almost the same overheads as those for the DASH protocol.

The software-based protocol implemented in distributed virtual shared memory (DVSM) [LH89] adopts a different approach. In the fixed distributed manager protocol of DVSM, the responsibility for updating the directory information is given to the requester node, not to the owner node. On receiving the requested copy, the requester node launches a signal to the home node to update the directory information. This is shown in Figure 4.12 (c). Although data access latency in the critical path consists of three transactions, the total number of requests is four. Also, the directory locking time is longer than for the other blocking protocols (see Figure 4.12) and has a higher probability of blocking successive requests.

Before describing the write-only blocking protocol (Figure 4.12 (d)), let us first take a look at the states of a line in the cache. Under a write-invalidation protocol, a state which indicates the validity of a cache copy is needed. One or more than one memory location can have the same copy of data. The data in the cache can be up-to-date but different from the copy in the home node. If we allow several caches to share an up-to-date cache copy which is different from the copy at the home node, the write-back request in the DASH protocol can be eliminated. In this case, another state which allows for the ownership of the shared cache copies is needed for efficient replacement of the up-to-date cache copy. Given this condition, if the previous request is a read request, subsequent requests can be forwarded to the current owner node before completion of the previous request. That is, if the previous request was a read request, blocking the directory is not necessary since the owner node has the same copy before and after the completion of the read request. This observation means that the read request only requires a maximum of three network transactions. Note that the directory information is updated when the read request first arrives at the directory (see the write-only protocol in Figure 4.12 (d)).

If the previous request was a read-exclusive request, for data consistency, all subsequent requests should have the copy updated by the read-exclusive requester node. Thus, the write-only protocol in Figure 4.12 (d) blocks subsequent requests if the current request is a read-exclusive request. The signal to unblock the directory is issued by the owner node as in the DASH protocol. After it is unblocked, the directory sends an acknowledgement to the requester node. The purpose of the acknowledgement is to prevent an early replacement of the newly acquired line, which now has ownership. That is, the requester node cannot replace the newly acquired line until it receives the acknowledgement from the directory.

Another interesting case can arise when the unblocking signal from the owner node N_1 arrives at the home directory too early. The acknowledgement issued from the directory arrives at the requester node N_0 even before the requested line arrives from N_1 to N_0, and turns N_0 to a new owner node of the requested line. The acknowledgement signal has to be preserved. Furthermore, any subsequent request for the line will be buffered at N_0.

The blocking schemes described so far have a minimum of three transactions on their critical paths. However, they require extra signals to unblock the directory entry. Another key drawback of these blocking schemes occurs when simultaneous accesses are made to the same line. This is shown in Figure 4.13, which assumes that a subsequent read request from N_2 arrives at the directory right after a read-exclusive request from N_0. As can be seen, both the length of the critical path and the number of requests for the second read request are substantially increased as compared to those in Figure 4.12. Table 4.1 summarizes the results when two requests to the same line are in progress simultaneously. In the table, it is assumed that each request transaction on the datapath takes two cycles and the second request starts one cycle after the first request (see Figure 4.13). Generally, the second request creates a larger number of requests and increases data access latency compared to the first request. Buffers at the directory usually reduce the number of requests and the data access latency.

Our proposed non-blocking directory scheme updates the directory information when

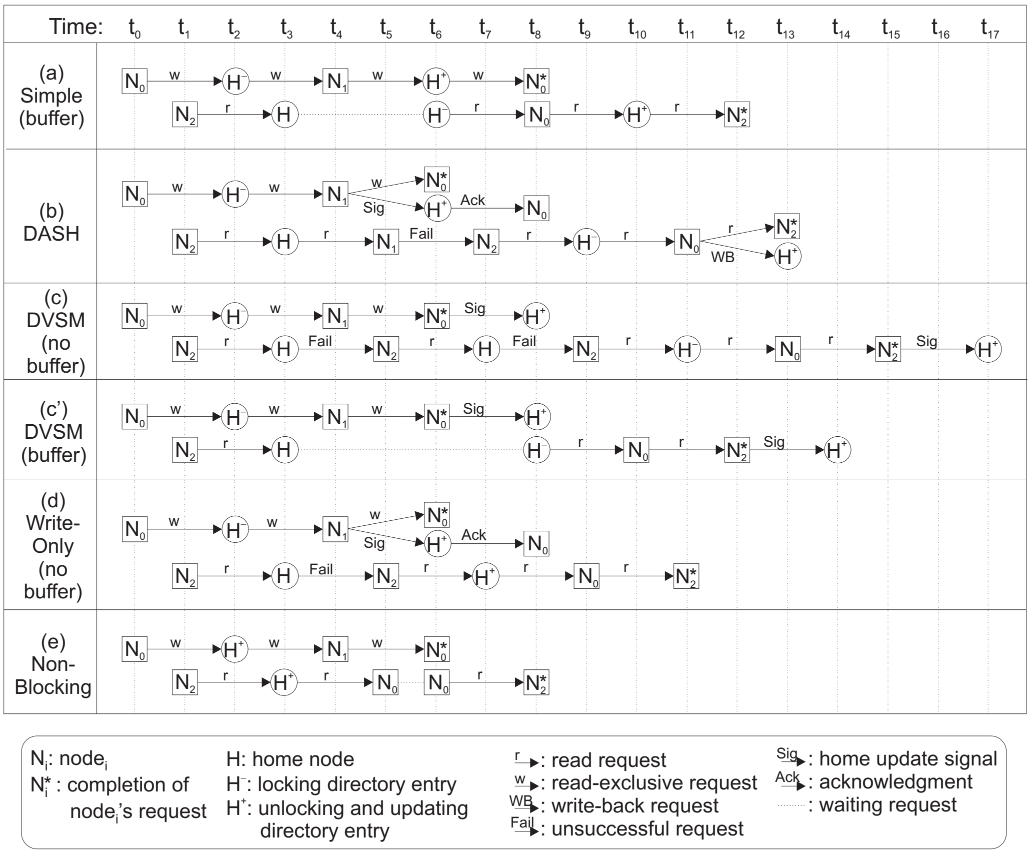


Figure 4.13: Flow of two simultaneous data requests in each directory protocol

a read or read-exclusive request first arrives at the directory (see Figure 4.12 (e)). This eliminates the final signal to unblock the directory entry. The non-blocking protocol requires up to three transactions for all data accesses (including read-exclusive requests) without generating extra traffic. It performs quite well especially when several requests are made simultaneously (see Figure 4.13 (e) and Table 4.1).

4.3.4 Non-blocking directory protocol

Figure 4.14 shows an example of the non-blocking protocol. In the figure, N_1 is the owner node of a line at time t_0. A read-exclusive request is issued to the home node H at time t_0, and is forwarded to N_1 after the directory changes the owner node from N_1 to N_0 at time t_2. A subsequent read-exclusive request from N_2 arrives at the home node at time t_3. In the blocking protocols, the request is blocked, i.e., it is buffered at the directory or returned to N_2 as an unsuccessful request. In the non-blocking directory protocol, the read-exclusive

protocols	Latency (cycles)			Number of requests		
	first request (r/w)	second request (r/w)		first request (r/w)	second request (r/w)	
		first req. is a read	first req. is a write		first req. is a read	first req. is a write
Simple (buffer)	8	7/11	11	4	2/6	4
DASH	6	10/12	12	4/5	5/9	7/8
DVSM (no buffer)	6	14	14	4	8/10	8
DVSM (buffer)	6	11	11/13	4	4/6	4
Write-Only	6	6	10	3/5	3/7	5/7
Non-blocking	6	6	7	3	3/5	3

Table 4.1: Simple comparison among protocols for two simultaneous requests

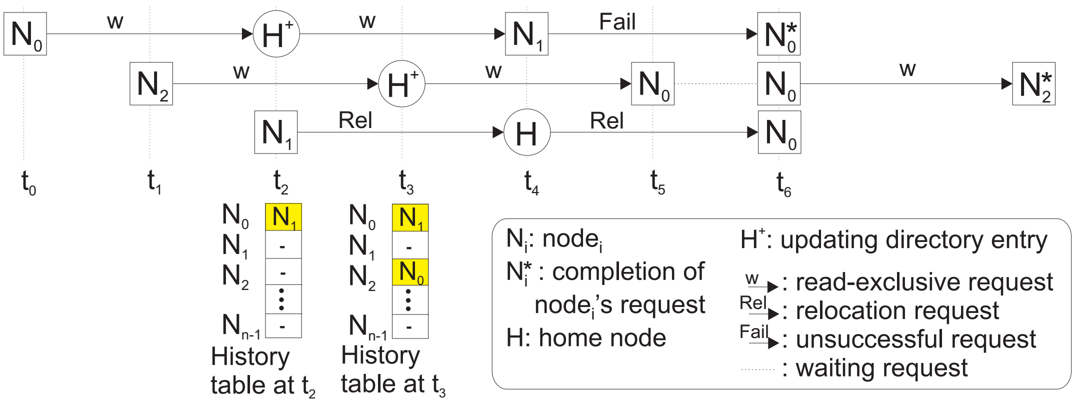


Figure 4.14: A simple example of a non-blocking directory protocol

request is forwarded to N_0 after the directory changes the ownership from N_0 to N_2. When N_2's read-exclusive request arrives at N_0 at time t_5, the request waits for the completion of N_0's request. If N_0's request successfully returns from N_1 with the requested line, the line is updated at N_0 and forwarded to N_2.

However, a problem arises when the line owned by N_1 is replaced at time t_2 before N_0's read-exclusive request arrives at t_4 as in Figure 4.14. N_0's request will not be able to find the requested copy at N_1. When the replaced line in a relocation request is delivered to the directory at time t_4, it should be forwarded to N_0. However, since N_2's request has updated the owner node at time t_3 in the non-blocking protocol, the information about N_0's request is lost. Although this may happen very infrequently, the directory must provide a means of finding the correct node when such replacement occurs.

To tackle this problem, the history of the owner nodes could be maintained in a small table, called the *history table*, in the directory. As can be seen in Figure 4.14, the owner

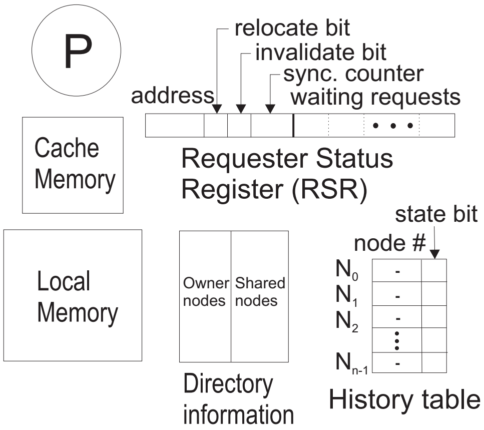


Figure 4.15: Hardware requirements for the non-blocking protocol

node N_1 is recorded in the table entry of the new owner node N_0 at time t_2, and the owner node N_0 is recorded in the entry of the new owner node N_2 at time t_3. When the relocation request from N_1 arrives at the directory at time t_4 and finds that the owner node has changed in the directory, it will search the table entry of the current owner node N_2. The history table entry of N_2 contains N_0 which is not the replacement requester node N_1, so it continues to look for the table entry of N_0. Since N_0's entry points to N_1, the search stops, and the relocation request is forwarded to N_0. Here, note that the sequence of data updates is based on the order in which the requests arrive at the directory. In the following, the non-blocking directory is described in more detail.

(a) Hardware requirements

Figure 4.15 shows some basic hardware requirements for the non-blocking protocol. The information about owner nodes and nodes sharing the line is maintained in the directory. The directory information can be kept either in a full-map or pointer-based directory structure. The history table described above is also located in the directory. The history table has n entries, where n is the number of nodes in the system. Each entry has a node number and a state bit. When the ownership of a line is changed by a read-exclusive request or by an invalidation request at the directory, the previous owner node is saved in the table entry of the requester node (new owner node). The state bit in the history table indicates whether the change of ownership is caused by a read-exclusive request or an invalidation request. Although our non-blocking scheme can support multiple requests from one node by relaxing the reference order in a program context [SS88, AG96], here to simplify our presentation we assume the processor can issue only one request at a time under our sequential consistency

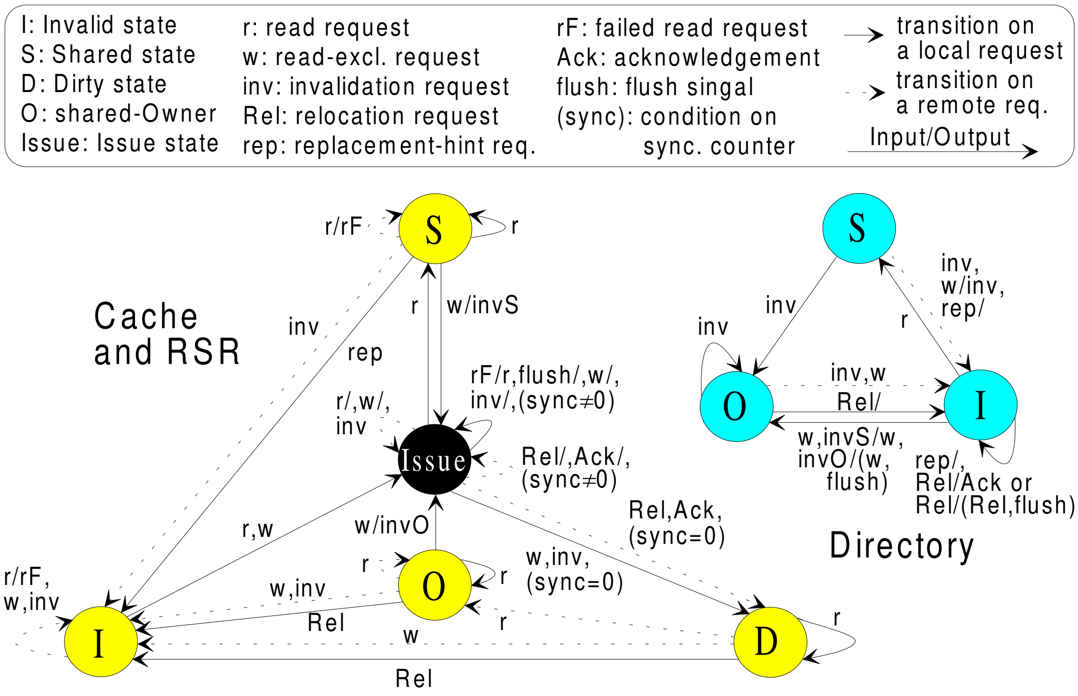


Figure 4.16: State transition diagrams for the cache and the directory

model [Lam79].

To support the non-blocking directory protocol, we also need some extra hardware at the cache. The information of the line being requested by the processor needs to be specified in a register, called the requester status register (RSR) (see Figure 4.15). To indicate whether the requested line is invalidated or relocated while the processor is waiting for completion of the request, two bits, called the *invalidate* and *relocate* bits, are used. The synchronization counter in the RSR counts the number of requests that the processor needs to receive. The synchronization information is also needed for most of the blocking protocols. Additionally, the non-blocking protocol requires some buffers to keep requests that are waiting at the cache. In the following section, we will see how the hardware components described here are used.

Figure 4.16 shows how the states in the cache and the directory are changed based on input requests. The output requests for each transition are also shown in the figure. We have five cache states of each line: invalid (I); shared (S); dirty (D); shared-owner (O); and issue (Issue) states. The first four states are the same as in [KEW+85]. However, note that the I-state also includes the situation when the requested line is missing in the cache due to a replacement. The Issue-state signifies the situation when the cache is issuing a request and the request is not completed yet. The home directory maintains states for each requester node as well. If the requester node is the owner node, the directory is in the owner (O) state. If the requester node is a node sharing the line, it is in the shared (S) state. Otherwise, the directory is in the invalid (I) state. That is, the requester node does not have a valid copy for the given request.

(b) Read request

Initially, both the cache and the directory states are in the I-states. When a processor's read request hits at the cache, there is no state change. A read miss launches a read request to the directory after changing the cache to the Issue-state. When a read request from N_i arrives at the home node, the directory forwards the request directly to the owner node N_j and considers N_i as a sharing node (i.e., change to the S-state). The history table remains unchanged. When N_i's read request arrives at the owner node N_j, N_j is in one of three possible situations. (1) If N_j is accessing the same line (i.e., in the Issue-state), N_i's request waits at N_j, and will be returned with a valid copy when N_j's request is completed. The state of N_i will be changed to the O-state. (2) If the requested data is missing in N_j due to a replacement (i.e., the I-state) or the requested data in N_j does not have ownership (i.e., the S-state), N_i's request is returned to N_i as an unsuccessful request. The unsuccessful request compels N_i to reissue the read request. (3) If N_j has the ownership copy (i.e., the O-state or the D-state), N_j sends a reply message containing the requested line to N_i and changes its state to the O-state. When the reply message arrives at N_i, its processor retries the request if the line is invalidated by a remote invalidation request during the issue of the N_i's request (i.e., invalidate bit in RSR is set). Otherwise, the processor resumes its execution after changing the cache to the S-state.

(c) Read-exclusive request

When a read-exclusive request from N_i reaches the home node, the request is directly forwarded to the current owner node N_j. The directory also sends invalidation signals to all of the nodes which have the same line. The previous owner N_j is saved in the history table entry of the requester N_i while setting its state bit to zero. The directory considers N_i as the owner node (i.e., the O-state). On receiving a read-exclusive request, the owner node handles it almost the same way it would a read request. However, the requester N_i does not retry the access even if it receives an unsuccessful request from N_j. This is because the requested line will be delivered to N_i as a relocation request (see Figure 4.14). The requester node N_i collects all of the invalidation acknowledgements, as well as the

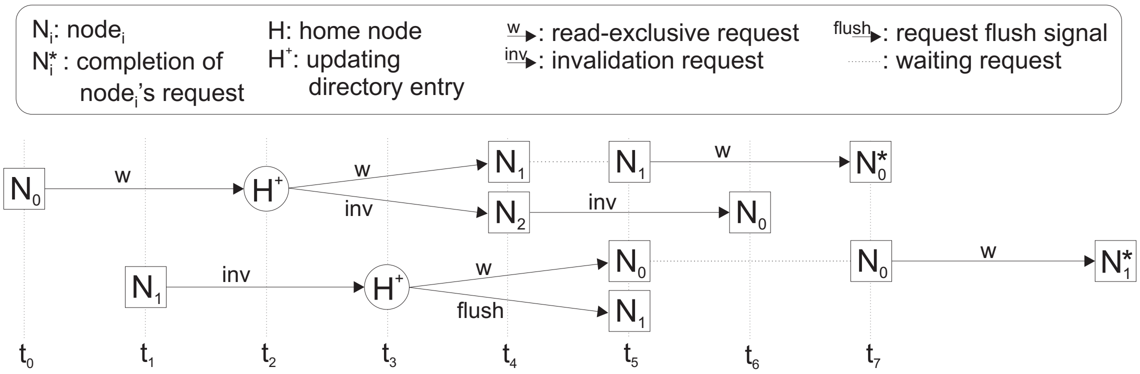


Figure 4.17: An invalidation request which creates a flush signal

requested line, using the synchronization counter. When all of the acknowledgements and the requested line have arrived (i.e., the synchronization counter is decreased to 0), the processor continues its execution after updating the line (i.e., change it to the D-state). If there are any remote read or read-exclusive requests waiting at N_i, the requests are returned successfully after changing N_i's cache to O-state or I-state respectively.

(d) Invalidation request

If N_i's write hits on a shared line (i.e., it is in the S-state or the O-state) in its own cache, N_i launches an invalidation request to the directory. When the invalidation request arrives at the directory, it is possible that the directory no longer considers N_i as a sharing node (i.e., it is in the I-state). This could happen if a node N_j's read-exclusive or invalidation request arrives at the directory before N_i's invalidation request. Since the order of the requests arriving at the directory determines the sequence of the data updates, the requested line should be updated by N_j first. Thus, N_i's invalidation request needs to be forwarded to the current owner N_j as a read-exclusive request. At the same time, if the invalidation requester node N_i is the previous owner node (O-state in N_i), a flush signal is sent to N_i to flush N_j's read-exclusive request back to N_j with a valid cache copy.

Figure 4.17 shows an example of an invalidation request which creates the flush signal. At time t_0, N_1 and N_2 share a line, and N_1 has the ownership. When N_0's read-exclusive request arrives at the directory at time t_2, it is forwarded to N_1 while sending an invalidation signal to N_2. Before N_0's request arrives at N_1, if N_1 wants to update the shared-owner line, an invalidation request is launched, and arrives at the directory at time t_3. At this time, the invalidation request is delivered to N_0 as a read-exclusive request since the directory does not consider N_1 a sharing node. Also, the directory sends a flush signal to N_1. At time t_4, N_0's read-exclusive request waits for the completion of N_1's invalidation request. By receiving the flush signal at N_1, N_0's read-exclusive request is returned to N_0 with the requested line while invalidating the copy in N_1 at time t_5. When N_1's read-exclusive (originally invalidation) request arrives at N_0 at time t_5, it also waits for the completion of N_0's read-exclusive request. At time t_7, N_1's read-exclusive (originally invalidation) request waiting at N_0 is returned to N_1 with the line updated by N_0.

If the directory considers N_i as a sharing node (i.e., the S-state or the O-state) when N_i's invalidation request arrives at the directory, the directory dispatches invalidation signals to all of the other sharing nodes. The requester node N_i becomes the owner node (O-state). If the previous owner node is different from N_i, the previous owner node is saved in the history table. The table's state bit is set to 1 indicating the line is invalidated. Unlike the read and the read-exclusive requests, the invalidation requests transmitted to the sharing nodes do not have to wait for the completion of any pending actions of the sharing nodes. This is because the copy to be updated by N_i is in N_i. They just invalidate any same cache copies (i.e., change to the I-state) in the sharing nodes. To invalidate the copy being issued by a sharing node (i.e., in the Issue-state), the invalidate bit in the RSR is set to 1.

(e) Relocation request

When a relocation request brings a replaced copy from the requester node N_i into the directory, the directory first compares the current owner node with the requester. If they are the same node (i.e., it is in the O-state), the replaced copy is simply written into the local memory. If not (i.e., it is in the I-state), then there exists a node N_j whose read-exclusive or invalidation request arrives at the directory before the relocation request, and changes the owner node from N_i to N_j. The copy in the relocation request must be updated by N_j. Since the directory forwards all requests after updating the directory information, it is possible that a different node N_k can be the owner node when the relocation request arrives at the home directory. To find the node whose request first changes the ownership (in this case, it is node N_j), the directory searches the history table recursively starting from the entry of the current owner node N_k. When the recursive search arrives at the entry of N_j, the directory knows that N_j took the ownership from N_i. If the state bit of the entry is zero, the directory forwards the relocation request to N_j and sends a flush signal to N_i. Or, if the state bit is set to one, the replaced copy is discarded and an acknowledgment is sent to N_j. The acknowledgment prevents N_j from generating another request before the replaced copy is discarded. Nothing in the directory needs to be modified at this time (i.e., it remains in the I-state).

If we omit the flush signal in the above case, a deadlock could occur. Figure 4.18 (a)

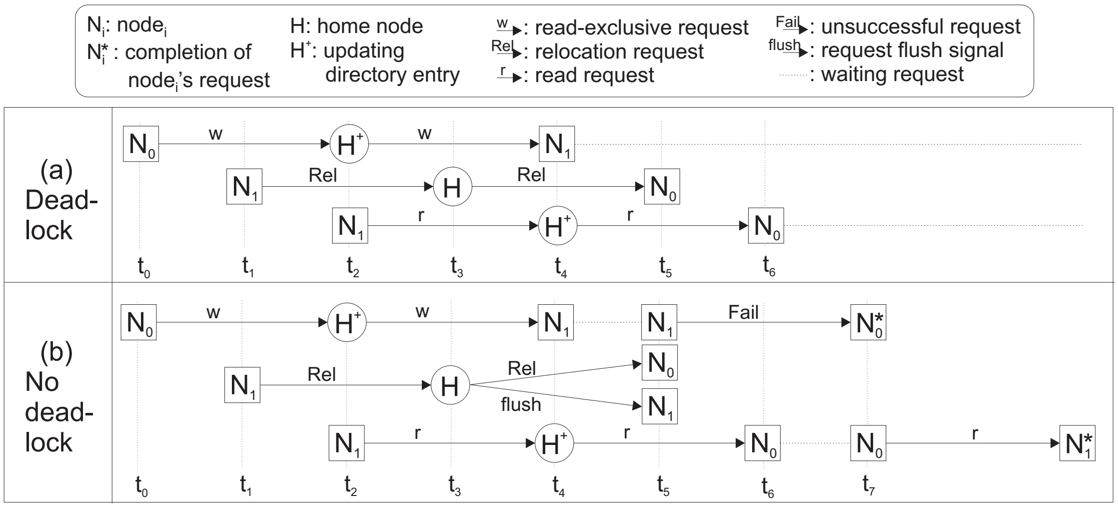


Figure 4.18: A relocation request involved in a deadlock

shows an example of the possible deadlock. At time t_0, N_1 is the owner node and N_0 generates a read-exclusive request. Before N_0's read-exclusive request arrives at the owner node N_1 at time t_4, the ownership copy is replaced from N_1 at time t_1 and wanted by N_1 again at time t_2. Thus, N_0's request waits for the completion of N_1's request at time t_4. Similarly, at time t_6, N_1's request, which follows the relocation request, waits at N_0 for N_0's request to be completed. That is, a deadlock occurs. The flush signal at time t_3 in Figure 4.18 (b) wakes up N_0's request waiting at N_1. N_0's unsuccessful request is returned to its requester. The wanted copy is delivered by a relocation request at time t_5. When N_0 collects all of the necessary requests at time t_7, it updates the copy and sends the copy to N_1 through the waiting request.

If the copy that a processor seeks to fetch is in a relocation request, the processor does not generate another request until its current request is successfully completed. Also, the requester whose request first changes the ownership always receives the requested copy. Thus, the history of owner nodes for a line in the table is not corrupted by a different request. This is the reason why the recursive search always finds the node whose request first changes the ownership of the line. If a replacement-hint request from N_i arrives at the directory, N_i is deleted from the sharing node list (i.e., it is changed to the I-state).

Figure 4.19 shows an example of some request flows. Initially, N_0, N_1, and N_2 share a line, and N_1 is the owner. If N_0 writes on the shared copy at time t_0, an invalidation request is launched to the home node H. Since N_1 and N_2 have the same copy, invalidation requests are sent to the nodes. Before the arrival of the invalidation at N_1, the relocation

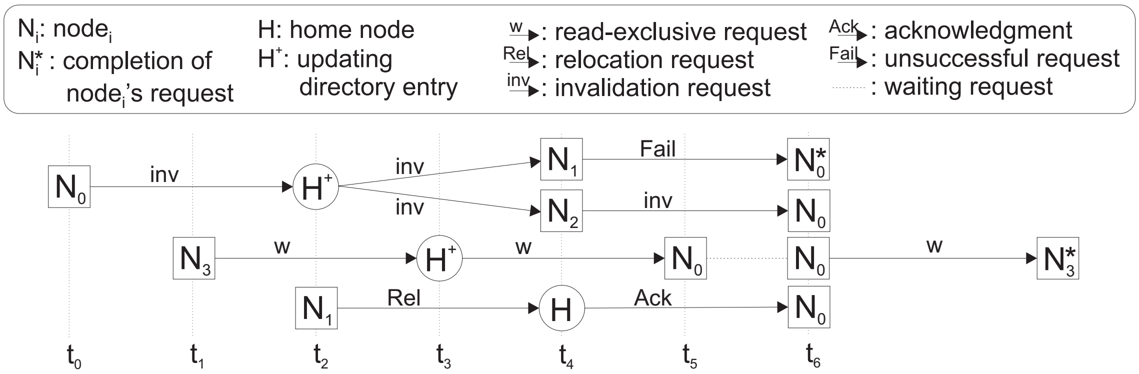


Figure 4.19: An example of some request flows

request is generated from N_1 at time t_2. Also, the read-exclusive request from N_3 arrives at the directory before the relocation request. When the relocation request arrives at the directory at time t_4, the directory finds that N_0's request changes the owner node from N_1 to N_0, and that N_0's request is an invalidation request. Thus, the replaced copy in the request is discarded, and an acknowledgment is delivered to N_0. The invalidation request at N_1 fails to invalidate the copy, and is returned to N_0. If the acknowledgement from the home node and the invalidation request from N_2 arrive, the copy in N_0 is updated at time t_6, and the waiting request is returned to N_3 with the updated copy. The copy in N_0 is invalidated.

Here, note that the maximum number of read-exclusive requests which could be waiting for a pending request at a node is one because no other waiting requests to the node can be made after the first read-exclusive request. Thus, the buffer to keep the requests waiting at a node can be restricted to a small size. When a read-exclusive request arrives at the node and the request has to wait for the completion of the node's request, all requests existing in the buffer have to be read requests. Therefore, if the buffer is full, the read-exclusive request can be kept in the buffer by victimizing a read request from the buffer. The victimized read request is returned to its requester node as an unsuccessful request.

To support multiple requests from a processor, we assume that the number of requests issued by a processor is limited by a constant number, k, and that each request has a unique request number associated with it. We need only increase the number of entries in the history table from n to kn, and the directory will be able to keep the information about the request number as well as the node number. When a read-exclusive or an invalidation request changes the ownership of a line, the previous owner node and its request number are saved in the table entry of the request.

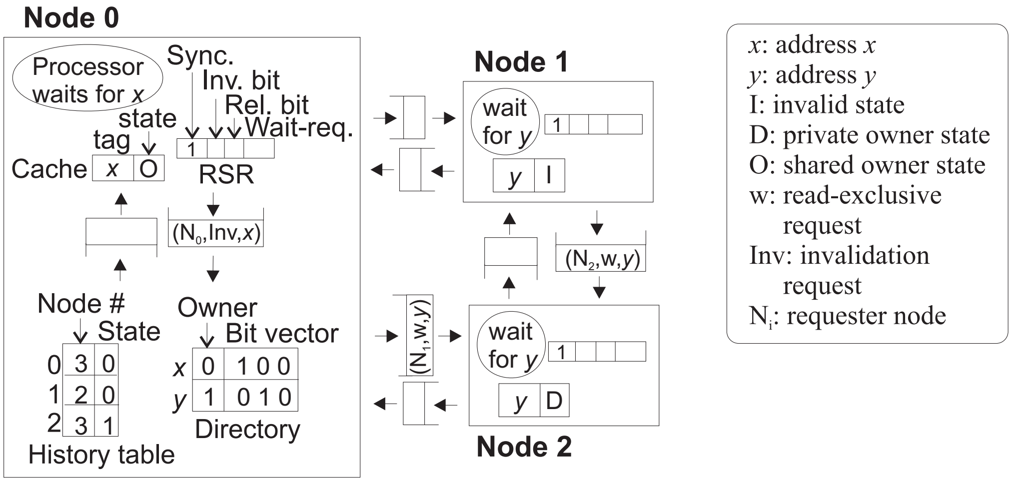


Figure 4.20: An instance of an abstraction model for protocol verification

4.3.5 Protocol verification

The number of different instances for a memory system is finite due to its limited size. If we look at all possible instances that can be reached from an initial instance with a given protocol, we can check the correctness of the protocol. However, a manual construction of all reachable instances is almost impossible because of the huge number of instances. Thus, in most cases, we rely on an automatic formal protocol verification tool. Here, we use the state enumeration method proposed in [ZWR$^+$80] as our automatic verification tool. The main obstacle of this method is the state explosion problem. To tackle this problem, we first choose an abstraction model from the original system (see Figure 4.20). The abstraction model assumes three nodes (Node 0, 1 and 2) and an address space of two memory locations (address x and y) which are mapped to the same one-line cache. Node 0 is the home node of the addresses.

Communication among the requester node, the home node, and the owner node is modeled as a completely connected FIFO queue system in both directions. The size of each queue is not limited. The processor state in the figure indicates one of three possible cases: processor running state; processor waiting for x state; and processor waiting for y state. Only one remote request can be pending at each node. Each request has three fields: address field; requester field; and request type field. Although the synchronization information should be in the request, to simplify the model, we do not include synchronization information in the request. When a valid copy is replaced from the cache due to a cache miss, the replacement request is combined with the data request. This is possible since both the miss copy and the replaced copy have the same home node.

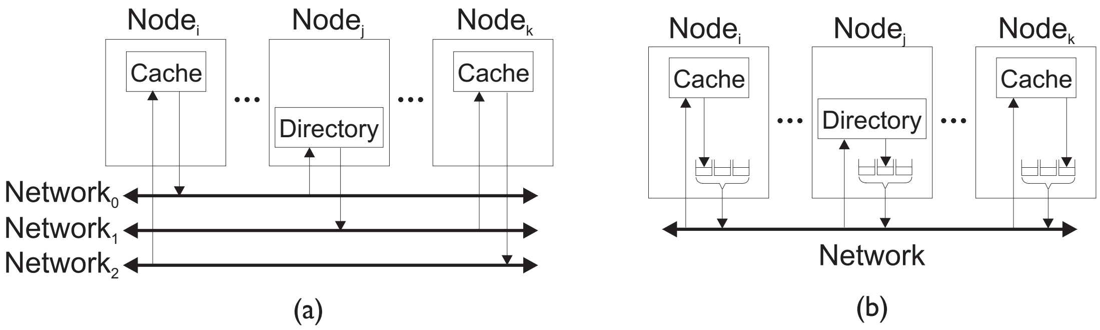


Figure 4.21: Two solutions to the deadlock problem

To reduce the number of instances further, we exploit two types of symmetry: address symmetry and structural symmetry. In the address symmetry, we exchange two addresses (address x and y) while leaving the structural configuration unchanged. In the structural symmetry, two nodes (Node 1 and 2) are swapped. This means the names of the two nodes are exchanged. Thus, all node numbers in an instance should be swapped and the queue positions related to the nodes also changed accordingly. The instance which adopts both symmetries at a time is also considered in our enumeration method.

Our verifier explores 26,174,955 different reachable instances with maximum depth of 79 in a breadth-first search. The maximum number of requests that can be in one queue at a time is six. In the breadth-first search tree, every instance has a successor other than itself (i.e., no deadlock). In each instance, the verifier examines two essential invariant conditions. The first invariant condition is whether the instance has one and only one ownership copy for each address or not. The second invariant condition is to check whether there is any valid copy in a different location when a processor updates the data in the cache. For data consistency, there must not be a valid copy in any different location when the copy is updated.

For completeness of our verification, we need to expand our abstraction model to a generic model (a non-finite configuration with n number of nodes and m number of addresses). Although some automatic techniques have been proposed for proving the correctness of the protocol in the generic model [PD95], it is not easy to specify the generic model using their notation. Moreover, verifying the specification for the generic model itself is another problem.

Another consideration in the correctness of a protocol is the occurrence of deadlock due to limited system resources. In the above verification, we assume that each pair of nodes has a communication path and that the size of the queue is not limited. In fact, if we know the maximum number of requests that can be in a queue at a time and the number is small, this problem can be solved by simply allocating enough space in each queue. However, it is not simple to find out the maximum number of requests that can be queued at a time in the generic model. Even if we figure out the number, the number could be too large to be allocated for each queue in a large-scale multiprocessor. When the queue does not have enough buffers, a deadlock situation can arise. As a solution to this, we utilize the fact that the non-blocking scheme proposed in this paper allows every processor's request to be completed in three datapath transactions. Thus, if we allocate a datapath for each transaction, we can prevent the deadlock. This is shown in Figure 4.21 (a). In the figure, the first, the second, and the third networks are used for the requests on the first, the second, and the third transactions respectively. Since all of the requests on the last transaction are completed through the third network, the requests on the second transaction can proceed, and so on. However, this solution requires a significant hardware cost. To reduce the hardware cost, if a separate queue is dedicated for each transaction (see Figure 4.21 (b)) and if the requests in the queue for the last transaction are serviced first (i.e., have the highest priority in the use of the datapath), we can prevent the deadlock from occurring.

4.3.6 Directory implementation

Figure 4.22 suggests one possible hardware layout for the directory. As shown in the figure, dedicated hardware is used for the pointer store by sacrificing some flexibility of system design. DRAM is utilized for the head links in the pointer store. The pointer store for the cache copies is assumed to be fast SRAM due to its small size. Six buffers (*receive, send, hBuffer, freeHd, temp0* and *temp1*), eight selectors (*id, freeSel, pAddr, pLink, pPtr, hDirty, hLink* and *hEnd*), and one comparator (=) are additionally necessary in the hardware design. Control signals for the hardware components are generated by the controller. Figure 4.23 shows the finite state machine control for the hardware design. The finite state machine in the figure performs the three functions described in the previous section. In the figure, we assume that a read and a write on SRAM for the same address can happen in a single cycle. If the directory sends just one request, the synchronization count (*sync*) of the request is set to 0. If it sends more than one request, all requests except the last one have the *sync* set to 1. The *sync* of the last request is the exact number of requests generated by the directory. The *sync* is used for synchronization at the requester. If a transition in the finite state machine from one state to another needs to have access to the DRAM pointer store, it might take longer time than the other transitions. Here, note that messages are generated in the middle of state transitions. Thus, directory processing could be overlapped with forming and delivering messages.

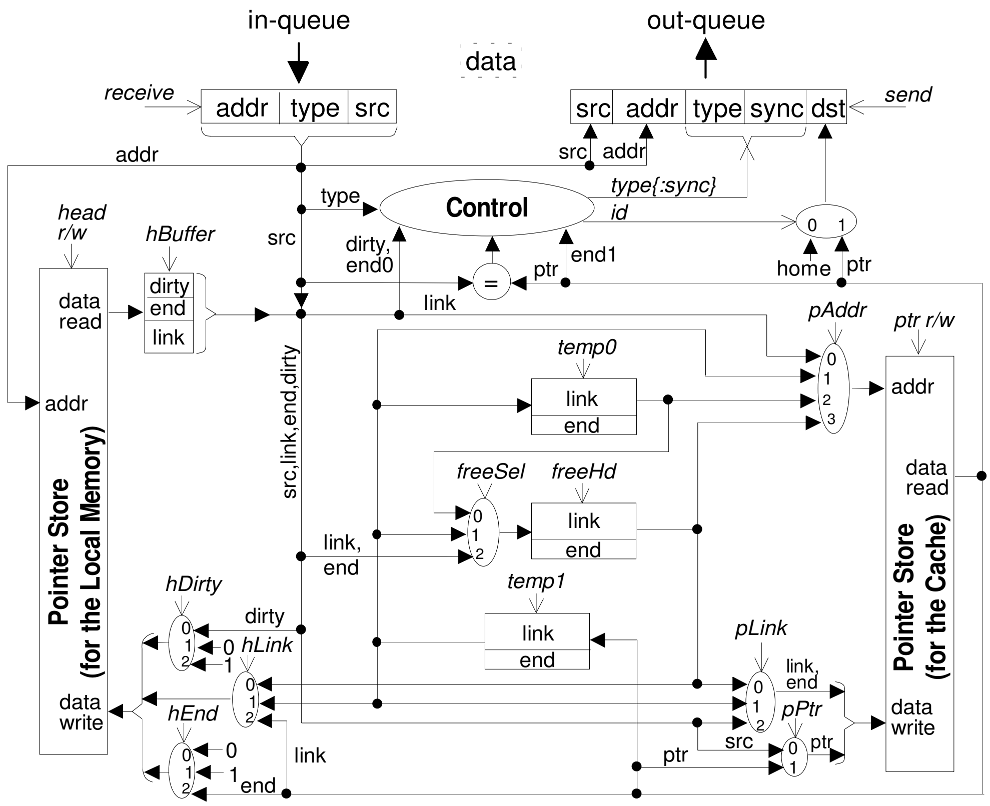


Figure 4.22: A datapath and control implementation for the directory

4.4 COMA protocols

Although the line addresses in the AM of COMA are changed at a node miss, directory information about a line address is maintained in a fixed node, called the home node. Thus, every request that is not satisfied in its node can be delivered to its home node. Like the directory of CC-NUMA, the directory of COMA always keeps information about the owner and sharing nodes for each line address. However, since the home node does not always have space for the replaced data in COMA, the directory of COMA needs to keep additional information about nodes which have space for the relocation request.

4.4.1 Directory organization

Figure 4.24 shows two directory organizations for COMA — full-map and pointer-based directories. From the figure, we can see the relationship between the full-map and pointer-based directories. Let us first look at the full-map directory. Each home directory has information on all the lines whose addresses are mapped to the same AM set. In the figure,

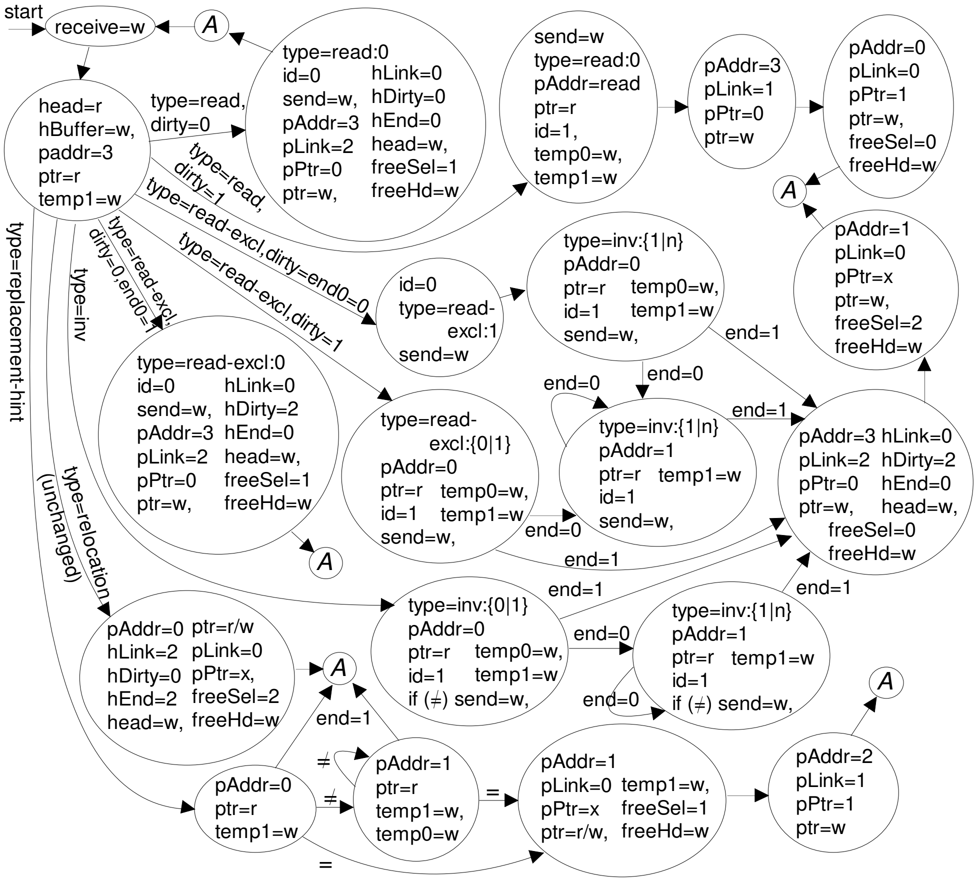


Figure 4.23: A finite state machine control for the directory

the lines that are mapped to the same set are shaded with the same color. Let us call the part of the directory shaded with the same color a *subdirectory.* As can be seen in the figure, the owner and sharing node information is associated with each line address. Here, note that the address mapping scheme on the home directory can not only reduce the directory size but also provide an efficient relocation algorithm. That is, just by checking the home directory of a line to be relocated, the node which has space for the line can be chosen.

For example, let us examine a subdirectory in the full-map directory. When a relocation request arrives at the home directory, the directory first tries to select a node which has the same copy by checking its row in the subdirectory. If such a node does not exist, the directory then tries to select a node which has an unused space by checking each column of the subdirectory. If the bits of a column are all 0s, there is no valid line in the AM set of the corresponding node. If there is only one 1 in a column, this means the corresponding AM set has only one valid line. Since the AM in Figure 4.24 is two-way set associative,

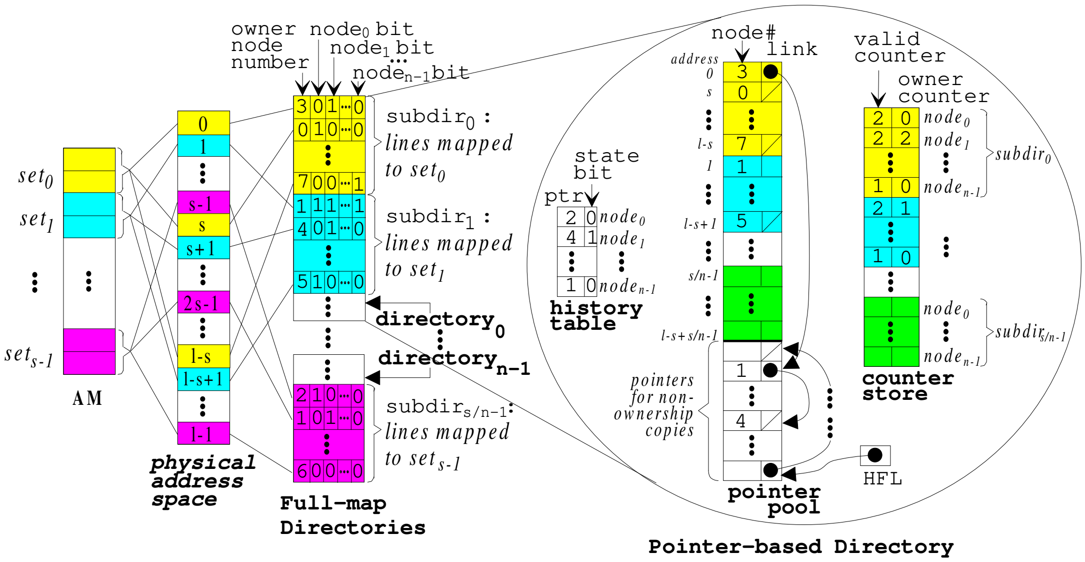


Figure 4.24: Directory organization for COMA

In the figure, l is the number of memory lines, s is the number of sets in the AM, and n is the number of system nodes.

the node still has one more available space. If all the columns have two or more 1s, the directory tries to find a node which has a shared non-ownership copy in its set by checking the subdirectory row by row. In the following, we will see how this relocation algorithm can be implemented in a pointer-based directory.

The circle in the right-hand side of Figure 4.24 shows the organization of a pointer-based directory in a node. The directory consists of a pointer pool, a history table, and a *counter store*. The node information about shared copies and the ownership copy of a line is stored in the pointer pool. The history table keeps the history of owner nodes for the non-blocking directory scheme. The usage of the history table in COMA is the same as in CC-NUMA. For efficient data relocation, the numbers of valid and ownership copies in an AM set for a node are counted in the counter store.

The number of subdirectories in a directory is $\frac{s}{n}$, where s is the number of sets in an AM and n is the number of nodes. If the AM set associativity is a, at most a lines from a subdirectory can reside in each AM at a time. Thus, the maximum number of pointers needed in a subdirectory is $a \times n$, and the number needed in a directory is $a \times n \times \frac{s}{n} = a \times s = m$, which is equal to the number of AM frames in a node.

Now, let us look at the pointer pool more closely. The data structure of the pointer pool is an array of singly linked, non-circular lists like the pointer pool in CC-NUMA. The number of line addresses allocated to a directory is $\frac{l}{n}$, where l is the total number of line addresses in the system. Since there is always one ownership copy for each line address, no linked lists are empty. The first $\frac{l}{n}$ pointers (*owner* pointers) among the m pointers are always allocated to point to the owner node, and they are used as the heads of the lists. The remaining $(m-\frac{l}{n})$ pointers (*non-owner* pointers) are used for information on the nodes which have non-ownership shared copies. Since the link field in each pointer identifies one of the non-owner pointers, the link field is $\lceil \log_2(m-\frac{l}{n}) \rceil$ bits wide.

It seems impractical to implement the relocation algorithm for the relocation request with the full-map directory. This is because the algorithm requires column and row searches in a subdirectory. To make it practical, the counter store is used in the pointer-based directory. Each subdirectory has n counters to count the number of valid or ownership copies which reside in the associated AM set in each node. This is shown in Figure 4.24, where each counter has two fields — valid and owner counter fields. The valid and owner counter fields count the numbers of valid and owner copies in the given AM set respectively. For example, the ith valid counter in the subdirectory for the AM set_j has the number of the valid copies in the AM set_j of node_i. Since the AM set associativity is a, the maximum number of copies that can be in an AM set is a. If we want to fetch the missing line first and carry out the replacement later to minimize processor stall time, the maximum value in the counter store is $a+1$ assuming each processor will generate only one data request at a time. The value in the valid counter is always greater than or equal to the value in the owner counter since the ownership copy is a kind of valid copy. If $\mathrm{vc}[i]$ and $\mathrm{oc}[i]$ respectively denote the counts of valid and ownership copies for node_i in a subdirectory, then $0 \leq \mathrm{oc}[i] \leq \mathrm{vc}[i] \leq a+1$ for any node_i. The values in the counter store are used for selecting a node which has space for the relocation request. The width of a counter store entry is $2\lceil \log_2(a+2) \rceil$ bits. The total number of counter entries in a directory is $n\frac{s}{n} = s$.

4.4.2 Directory protocol

Although the directory protocol of COMA is very similar to that of CC-NUMA, there are some differences due to the replacements at the AM. These differences are emphasized in the following discussion. Initially, the values in the counter store are all set to zero, and the free list, starting from the head of the free list (HFL), links all of the non-owner pointers in the pointer pool. If a new page is first loaded into main memory, all of the lines in the page are sent to the node number i (N_i) which causes the load. At the home directories of the lines, the owner pointers for the lines point to N_i. Both $\mathrm{oc}[i]$ and $\mathrm{vc}[i]$ in the associated subdirectory increase by one.

When a read request from the requester node N_i arrives at the home node, the request

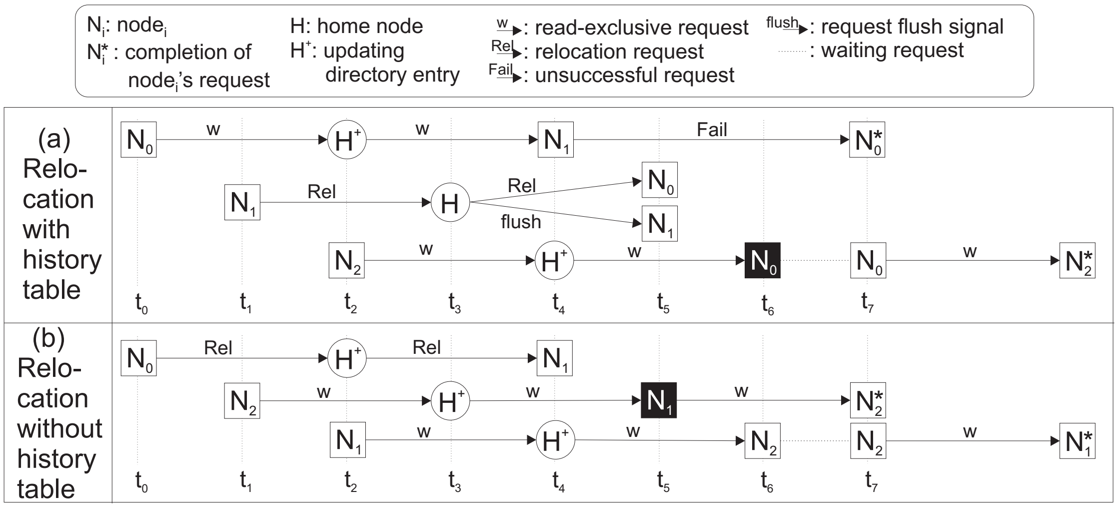


Figure 4.25: Two types of relocation requests in COMA

is forwarded directly to the owner node pointed by the owner pointer. At this time, vc[i] increases by 1 if the owner node is different from the requester node. An unused pointer needs to be allocated to keep the information of the requester node N_i, and is either allocated from the free list or retrieved from a replacement-hint request if the replacement-hint request arrives at the directory together with the read request. The allocated pointer is inserted after the owner pointer while copying the link of the owner pointer.

When a read-exclusive request from N_i reaches the home node, the request is directly forwarded to the current owner node N_j. At this time, while moving through the associated linked list, the directory also emits invalidation signals to all of the nodes which have non-ownership copies. The non-owner pointers in the list are deallocated to the free list by being inserted after the HFL. If the owner node (N_j) is not the requester node (N_i), vc[k] for each N_k (including N_j) in the list and oc[j] drop by one, while vc[i] and oc[i] increase by one. If the owner node is the requester node ($N_j = N_i$) due to a relocation, no counter is changed.

There are two types of relocation requests at the directory. When a relocation request arrives at the directory, the directory compares the current owner node with the requester. If they are different, the relocation request is forwarded to the node whose read-exclusive or invalidation request took the ownership from the requester. The history table is used to find the node as in CC-NUMA. At this time, no directory information is changed. Figure 4.25 (a) shows the example of this. In the figure, the read-exclusive request from node N_0 arrives at the home node H at time t_2, and the relocation request from N_1 arrives at the home node at time t_3. Since the owner node is changed, the relocation request is forwarded to N_0 using the history table. When the read-exclusive request from N_2 arrives at N_0 at time t_6, N_2's request has to wait for the completion of N_0's request since N_0's request arrives at the directory earlier than N_2's request. In this case, there is no difference between COMA and CC-NUMA.

If the owner node is the requester node, when a relocation request arrives at the directory, the second type of the relocation happens. In CC-NUMA, the relocation request is saved in the local memory of the home node. In COMA, the relocation request is forwarded to a node without using the history table. In order to do this, the directory first tries to select a node which has a shared copy of the replaced line. That is, if there is a non-owner pointer in the associated linked list, the relocation request is forwarded to the node pointed by the non-owner pointer. The node becomes the new owner of the replaced line by copying the values of the non-owner pointer into the owner pointer. The non-owner pointer is deallocated. Here, note that the relocation request forwarded to the sharing node cannot find the shared copy in the node due to a replacement at the node. This could induce another relocation request from the node and, hence, a deadlock. In order to prevent the occurrence of such chained relocation requests, we can skip this node selection step.

If there is no shared copy (no non-owner pointers), the directory tries to select a node whose value in the valid counter of the corresponding subdirectory is less than the AM set associativity, a. Since a is usually a power of two, we just need to check the most significant bit in the counter field to see whether the value is less than a or not. If such a node exists, the relocation request is forwarded to the node. The node becomes the new owner of the line by saving the node identification into the corresponding owner pointer. If such a node does not exist, the directory now tries to find a node whose value in the owner counter is less than a. The last search always selects a node as long as the number of AM frames is greater than or equal to the number of line addresses in the AM, i.e., $m \geq \frac{l}{n}$. If a replacement-hint request from N_i arrives at the directory and N_i is not the current owner node, $vc[i]$ decreases by one and the non-owner pointer for the replaced copy is deallocated.

Figure 4.25 (b) shows an example in which a relocate request is forwarded to a node without using the history table. In the figure, since the owner node (N_0) is not changed when the relocation request from N_0 arrives at the home node at time t_2, it is forwarded to node N_1 which is selected by the relocation algorithm in COMA. The relocation request is relocated at N_1 at time t_4 by setting the relocate bit in the requester status register (RSR). N_2's read-exclusive request which arrives at the directory at time t_3 is forwarded to the new owner node N_1. If a read-exclusive request from N_1 is generated before the arrival of N_0's relocation request, the directory forwards N_1's request to N_2 at time t_4. When N_2's request

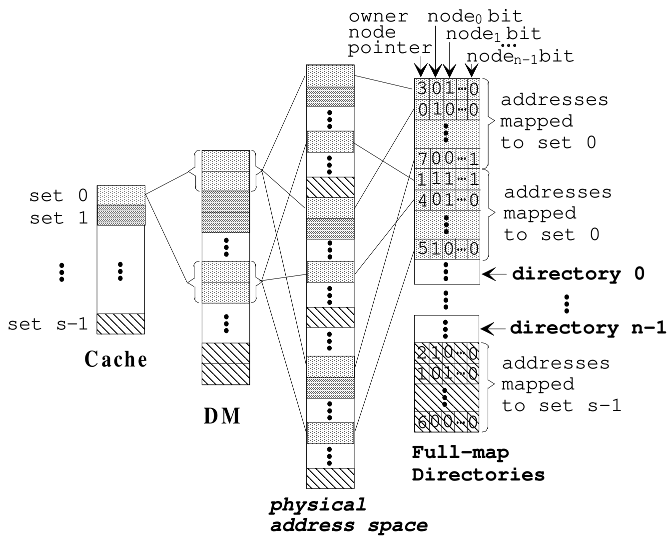


Figure 4.26: Address mapping in the directory of DYMA
The addresses mapped to the same cache set are allocated to a directory. Also, the addresses in a subdirectory are mapped to the same DM set.

arrives at N_1 at time t_5, N_1's request is not completed yet. In this case, N_2's request must be returned to N_2 with the relocated data without waiting for the completion of the N_1's request. This is because N_2's request arrives at the directory earlier than the N_1's request. Therefore, the two types of relocation requests must be distinguished in COMA so as to handle the read-exclusive request differently at the owner node.

4.5 DYMA protocols

In DYMA, each home directory has the information of all the lines mapped to the same set of the cache as in CC-NUMA. Also, the directory is divided into subdirectories which have the information of all the lines mapped to the same set of the DM, like the directory of COMA. The address mapping in the full-map directory of DYMA is shown in Figure 4.26. The reason we use this kind of address mapping is to reduce the directory size while still allowing an efficient relocation algorithm. The pointer-based directory of DYMA is very similar to that of COMA (see Figure 4.24). Since the inclusion property is not enforced between the cache and the DM, the number of pointers for the non-ownership copies in DYMA is larger than that in COMA by the number of cache frames. Also, the maximum value in the counter store is the sum of the DM and cache set associativities.

4.5.1 Memory access mechanism

We optimized memory access latency by avoiding the accesses to the DM tag when cache coherence misses happen. This optimization, the use of data states, and the memory access mechanism are described in the following:

[Cache]

Processor's request: If a write hits on a non-shared line or a read hits, give the data to the processor. If a write hits on a shared line, an **invalidation request** is created and delivered to the home directory. Depending on the reference type, a **read** or **read-exclusive request** is sent directly to the home directory if a cache coherence miss happens; otherwise, the request is sent to the DM. At the same time, relocation activity can start. If there is no space (i.e., no invalid data) to save the requested line in the cache, select a valid line to be replaced. Here, although any replacement algorithm can be used, the random selection is applied. If the selected line is an updated line or an exclusive line (i.e., a line new to the DM), a **relocation request** is launched to save the line at the DM. This action is overlapped with fetching a requested line or running of the processor. If the selected line is invalidated and it is an inclusive line, a local **invalidation request** is launched to the DM.

Remote data request: If the cache is issuing a request for the same address, the remote data request waits for the completion of the local request. However, if a relocation request which arrived at the directory earlier than the local request is already waiting for the completion of the local request, the remote data request is returned successfully to its requester node with the requested line. If the cache is not issuing a request for the same address and the requested line exists at the cache, the line is copied from the cache and returned to the requester node as a successful request. At this time, the existing copy is invalidated if the remote request is a read-exclusive request. If the line does not exist in the cache, the remote request is forwarded to the DM.

Remote invalidation request: The remote invalidation request does not wait for the completion of any pending action of the cache. The request invalidates the line if the line is at the cache. If a request for the copy is being issued by the cache, the copy is also invalidated by setting the invalidate bit in the requester status register (RSR). An acknowledgment signal is sent directly back to the requester node. If the copy is not in the cache, the invalidation request is sent to the DM.

Remote relocation request: If the cache is issuing a request for the same address, the remote relocation request waits for the completion of the local request by setting the relocation bit in the RSR. If not and the same copy exists in the cache, change the state to the owner, and the relocation is successfully completed. Otherwise, the relocation request is delivered to the DM.

Returned request: After collecting all of the necessary requests, if the data request successfully brings a valid copy, the missing copy is filled into the cache, and the processor continues its execution. If there exist waiting requests for the copy, the waiting requests are returned to their requester nodes containing the requested copy. If a read request is not returned successfully and the line is not relocated, the read request is launched again.

[DM]

Local data request: If a read-exclusive request hits on a non-shared line or a read request hits, return the request to the cache as a successful request. If a read-exclusive request hits on a shared line, deliver an **invalidation request** to the home directory while supplying the requested copy to the cache. If a DM miss occurs, forward the request to the home directory.

Remote data request: If the line exists in the DM, copy it from the DM, and deliver the copy to the requester node. If the requested copy is not available due to a replacement, the request is returned to the requester node as an unsuccessful request.

Invalidation request: Invalidate the line if it is at the DM. Send back an acknowledgment signal to the requester node if the request is from a remote node.

Local relocation request: If the replaced line from the cache is an inclusive line, the cache line is saved without further replacement. If the cache line is an exclusive line and there is no space (invalid lines) in the DM, a valid DM line is selected for replacement with the following priority sequence; a shared non-ownership line, a shared ownership line, and a private ownership line. If a shared non-ownership line is selected, the node sends a **replacement-hint request** to the home directory of the DM line. Otherwise, send a **relocation request** to the home directory.

Remote relocation request: If the same line exists in the DM, just change the state to owner. If there is an invalid line, overwrite the replaced line on the invalid line. If a different, shared non-ownership line is in the DM, substitute the requested line with the non-ownership line. A **replacement-hint request** for the non-ownership line is sent to the home directory.

[Directory]

Data request: The data request is forwarded directly to the owner node pointed by the owner pointer. At this time, if the request is a read-exclusive request, the directory also emits invalidation signals to all of the nodes which have non-ownership copies.

Invalidation request: If the directory considers the requester node a sharing node, invalidation signals are launched to all of the shared nodes except the requester. If not, the invalidation request is treated as a read-exclusive request.

Replacement-hint request: If the requester node is not the current owner node, the non-owner element for the replaced copy is deallocated.

Relocation request: If the requester node and the current owner node are different, then find the node whose read-exclusive or invalidation request took the ownership from the requester node by searching the history table recursively. The relocation request is forwarded to the node if the state bit is set to zero, or the replaced line in the

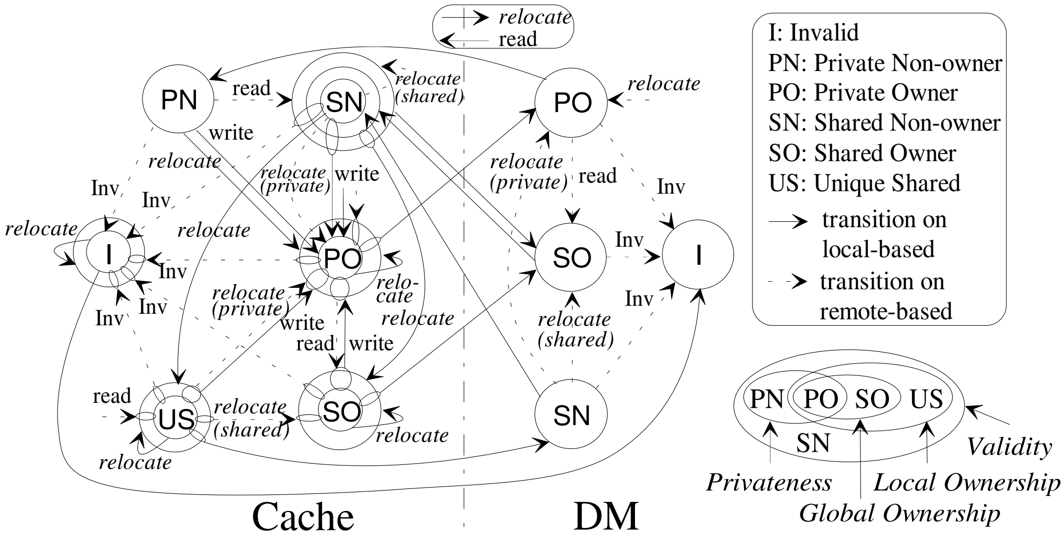


Figure 4.27: Data state transition for DYMA

request is discarded. If the owner node is the requester node, the directory first tries to select a node whose value in the valid counter of the corresponding subdirectory is less than the DM set associativity, a . If such a node exists, the relocation request is forwarded to the node. If such a node does not exist, the directory finds a node whose value in the owner counter is less than a and forwards the relocation request to the node.

4.5.2 Cache and DM state transition

This section describes how the states of the lines in the cache and the DM are changed based on input requests. The state transition for the cache and the DM can be found in Figure 4.27.

Even though this is a non-standard cache coherence protocol, stock microprocessors can be utilized with little difficulty by modifying the protocol. A coherence protocol with four states and one local bit for each cache line, which more closely resembles the traditional MESI protocol [SS86], can also be developed [KL95]. Further, if we assume that the cache in our discussion is the off-chip level-2 cache, implementing this optimized protocol has little bearing on the specifics of microprocessors. All modern microprocessors with on-chip cache provide the necessary signals for implementing multi-level caches.

Initially, all of the cache and the DM lines are in invalid (I) states. When a page is first loaded from disk, all lines of the page are saved in the DM with a private owner (PO) state. When a read access hits at the cache, there is no state change. When a line is copied from its local DM for a read access, the line becomes private non-owner (PN) if its DM state is PO; otherwise it becomes shared non-owner (SN). When a line is copied from a remote node for a read access, the line is loaded into the cache in a unique shared (US) state. At this time, the state of the remote node becomes shared owner (SO). The write access makes the cache line PO-state while all other copies of the line become the I-state.

If a PO-/SO-state line is selected for replacement, the line is saved at its DM with the same state. If a US-state line is selected, the line is saved in the DM with SN-state. If a SN-(from PN)/I-state (inclusive) line is selected, the DM line becomes SO-/I-state. An interesting feature of the protocol is the way that the non-inclusive line discarded from the cache can be saved in the DM efficiently. Since the cache and the DM function individually, it is not possible to search the tags of the DM when the replacement action is taken at the cache. The technique we use to find the DM space for a non-inclusive cache line without searching the DM is as follows. If a non-inclusive line is selected for replacement at the cache, find an inclusive line mapped to the same DM set of the non-inclusive line in the cache set. If there is an inclusive line, send the non-inclusive line to the DM, where the non-inclusive line is saved at the DM frame of the inclusive line. The cache state of the inclusive state line is changed to the non-inclusive state. More precisely, if a copy of the DM line replaced by a non-inclusive cache line exists in its cache set with PN-/SN-state or with inclusive PO-/SO-/US-/I-state, the state of the cache line changes to the non-inclusive PO-/(US- or SO-) state or to the non-inclusive PO-/SO-/US-/I-state respectively (local-based relocation transitions in Figure 4.27). If there is no inclusive line, check if the missing line, which causes the replacement, is mapped to the same DM set as the selected non-inclusive line. If it is, save the non-inclusive line at the DM frame of the missing line if it is in the DM. The missing line is filled into the cache with a non-inclusive state. If neither of the cases occur, the non-inclusive line can be saved in any frame of the DM set without changing cache states since no line in the DM set has a copy at the cache. The number of DM sets should be n times larger than that of cache sets for this technique to work correctly, where n is a positive integer. When a DM line is relocated to a remote node (remote-based relocation transitions in Figure 4.27), it becomes SO-state if its copy exists at more than one node. Otherwise, the line is relocated with PO-state.

From the above state transitions, we can find that the cache of DYMA has a larger number of states for additional functions than do CC-NUMA and COMA. However, since the additional functions can be accomplished with just one tag access and happen only for replacement, we assume the service time for a request at the cache is the same in the three DSM architectures. For modulization, DYMA sacrifices some replacement overhead. That is, to relocate an exclusive cache copy at the DM, our DYMA protocol selects an inclusive cache copy at the cache and relocates the exclusive cache copy at the DM frame

size (bits)	CC-NUMA	COMA	DYMA
pointer pool	$m(\lceil\log_2 c\rceil + 1) +$ $c(\lceil\log_2 n\rceil + 1 + \lceil\log_2 c\rceil)$	$m(\lceil\log_2 n\rceil + 1 +$ $\lceil\log_2(m - \frac{l}{n})\rceil)$	$(m + c)(\lceil\log_2 n\rceil + 1 +$ $\lceil\log_2(m - \frac{l}{n} + c)\rceil)$
counter store	-	$2\lceil\log_2(a + 1)\rceil s$	$2\lceil\log_2(a + a' + 1)\rceil s$
history table	$(\lceil\log_2 n\rceil + 1)n$		

Table 4.2: Comparison of DSM directory sizes in a node

In the figure, n = number of nodes, l = number of line addresses, m = number of frames in a local memory, c = number of frames in a cache, s = the number of local memory sets, a = local memory set associativity, a' = cache set associativity.

allocated for the inclusive cache copy even if the exclusive cache copy can be saved in a different DM frame without generating global replacements. The state of the inclusive cache copy is changed to the exclusive state at the cache. In this case, the relocation for the newly transformed, exclusive cache copy is extra overhead if the copy is not modified when selected for replacement. The influence of this technique on the performance of DYMA will be shown in the next chapter. Although the modulization is hard to accomplish in COMA due to the inclusion property, we assume that probing the cache for accesses from remote nodes can be done atomically without adding penalty.

Frequent accesses to the cache state in any of the three DSM architectures may impede the processor's access to the cache. Thus, one may want to provide the controller with a duplicate copy of both cache tag and state; this will shield the processor's access to its own cache from the interference of unsuccessful attempts to match cache tag and state caused by remote accesses and invalidations. If the copy from the cache can be provided much faster than the copy from the local memory, the copy from the cache is favorable in terms of total system performance since it decreases the stall time of the processors generating the remote access. In our organization, the tag and state of the cache is not duplicated. Also, accessing the tag and state and accessing data in the local memory occur concurrently.

4.6 Directory size comparison

Before analyzing data access latency in each architecture, let us compare the DSM architectures in the context of directory size. Table 4.2 summarizes directory sizes in the DSM architectures. Although standard commercially available chips can introduce different directory sizes, we calculate the number of bits actually used for directory information. In the table, we assume that in CC-NUMA each line address in the local memory has a head link which points to the associated linked list in the pointer pool. We use a 512 multiprocessor organized with one processor per node (n = 512). Each node has a 512M byte local memory and a 8M byte cache with 64 byte line size ($m = 2^{23}$ and $c = 2^{17}$). The cache and the local memory have 4 and 32 way set associativities respectively ($a = 32$ and $a' = 4$). The unallocated space in the local memories of COMA and DYMA is 32M bytes (memory pressure, r, $= \frac{512-32(\text{data size})}{512(\text{memory size})} = 0.9375$), i.e., the physical address space (l) in CC-NUMA is 4G lines while it is 4G−256M lines in COMA and DYMA. In this configuration, the directory size overheads compared to the local memories in CC-NUMA, COMA, and DYMA are respectively 3.60%, 5.74%, and 6.02%.

4.7 Data access latency comparison

For fair comparison among the CC-NUMA, COMA, and DYMA machines, we analyzed data access latency for each machine. Even though the analysis is a simplified one, it helps us to see some parameters related to latency in each memory architecture. In the analysis, data accesses are divided into four different cases: local-cache hit; local-memory hit; remote-cache hit; and remote-memory hit. Let us say that L_{lc} is latency for a local-cache hit, L_{lm} is latency for a local-memory hit, L_{rc} is latency for a remote-cache hit, and L_{rm} is latency for a remote-memory hit. Then, the *average* latency per reference in each memory architecture can be expressed as follows:

$$H_{lc}L_{lc} + M_{lc}H_{lm}L_{lm} + M_{lc}M_{lm}H_{rc}L_{rc} + M_{lc}M_{lm}M_{rc}L_{rm}, \tag{4.1}$$

where H_{lc} ($= 1 - M_{lc}$) is the local-cache hit *rate*, H_{lm} ($= 1 - M_{lm}$) is the local-memory hit *rate*, and H_{rc} ($= 1 - M_{rc}$) is the remote-cache hit *rate*. In equation (4.1), the average node latency is the sum of the first two terms, and the average remote latency is the sum of the last two terms. Figure 4.28 shows the latencies for each individual data access in the memory architectures.

Local cache hit: When a local-cache hit occurs in DYMA or COMA, there are two possible courses of action: (i) write access to a non-shared line or read access; or (ii) write access to a shared line. In the first case, the latency is the time for cache access (t_c). Here, we assume that the read access time and the write access time are the same. When a write access hits on a shared line, coherence action is necessary. This coherence action may need three network traversals: one for sending a request to the home node, another for sending signals to the nodes which have copies of the line to be invalidated, and one more for the acknowledgment. Here, the processing time at the home directory is assumed to be α. The acknowledgment does not have to go to the home directory since the acknowledgment has synchronization information. Another assumption is that the requester node, the home node, and the responding remote nodes are different. Changing the states of the cache and

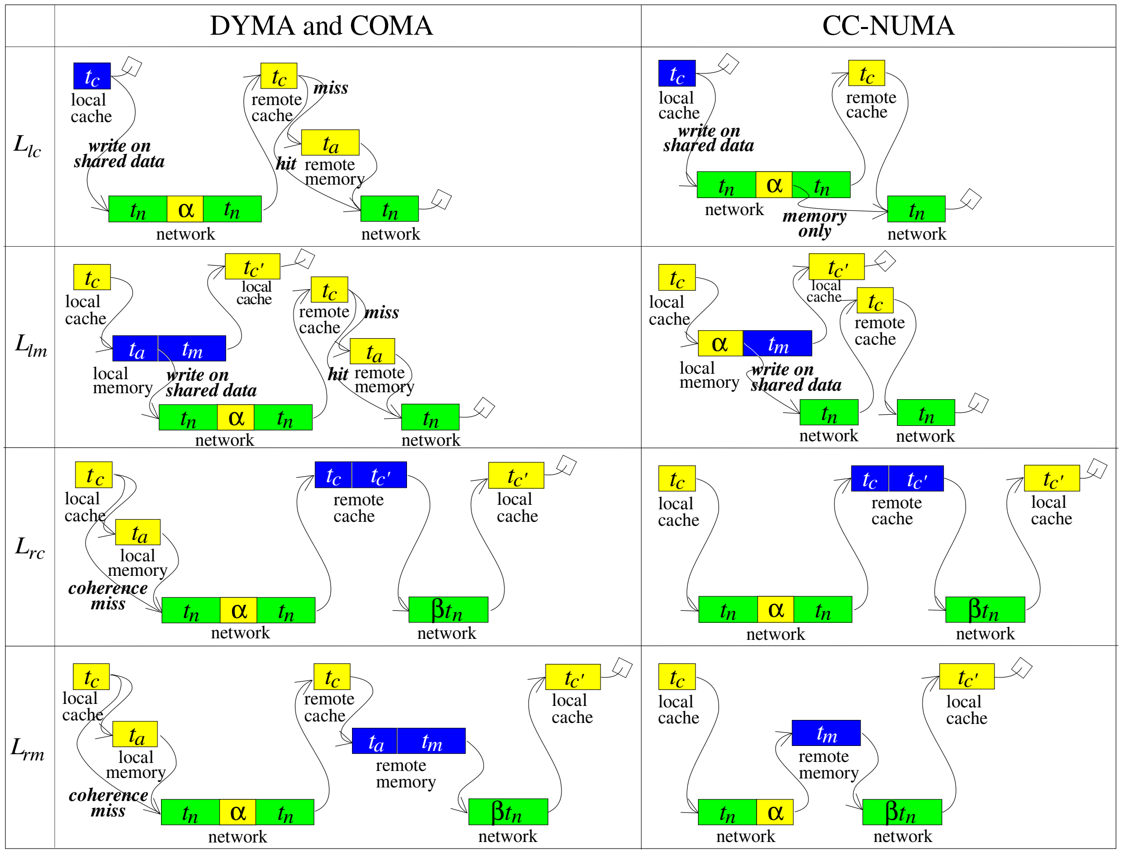


Figure 4.28: Data access latency in the DSM architectures

In the figure, t_c = cache access time, $t_{c'}$ = cache fill time, t_m = memory access time, t_a = memory tag access time, t_n = network traversal time without data, α = directory processing time, βt_n = network traversal time with data.

the local memory takes t_c and t_a respectively. Thus, if each network traversal takes t_n, the time for coherence action at the cache hit in DYMA or COMA is $t_c + t_n + \alpha + t_n + t_c + t_n$, assuming that a copy of the line in a remote node is at the cache. Here, note that a local memory access is not necessary even if a copy of the line is in the local memory. The reason is that an invalidation signal for the DM line is generated when the invalidated inclusive line is replaced later. This makes the coherence action faster. Also, a local memory access can be eliminated if a coherence miss for the invalidated line happens (see L_{rc} and L_{rm} in Figure 4.28).

Overall, the local-cache hit latency, L_{lc}, in COMA and DYMA is $(1 - P^s_{lc})t_c + P^s_{lc}(2t_c + 3t_n + \alpha + P^m_{rc}t_a)$, where P^s_{lc} is the probability that the reference is a write access to a shared line when there is a local-cache hit, and P^m_{rc} is the probability that the copy to be invalidated is not in the remote cache when the invalidation request arrives at the remote node.

In CC-NUMA, when the processor writes on a shared line, the shared copy can exist either in a remote cache or in the local memory of the home node only. If P_m^{only} is the probability of incurring the second case, the local-cache hit latency is $(1 - P_{lc}^s)t_c + P_{lc}^s(1 - P_m^{only})(t_c + 2t_n + \alpha) + P_{lc}^s P_m^{only}(2t_c + 3t_n + \alpha)$. To minimize the remote access overhead, the directory information (dirty bit) may be updated when a write hits on a clean cache copy. Otherwise, all cache misses should be forwarded to a remote node which has the copy in the cache since the memory copy may not be up-to-date. L_{lc} of COMA and DYMA is larger than that of CC-NUMA if P_{lc}^s is the same for the different architectures. Note that due to the inclusion property the access to the tag of the AM is needed when invalidating the copy of the line at a remote node even though the access time is not on the critical path. DYMA does not have the access to a remote DM if the copy is only in the remote cache.

Local-memory hit: When a local-memory hit occurs at the AM for the accesses missed at the cache, there are two possible courses of action: write access to a non-shared line or read access; or write access to a shared line. In the first case, the latency is t_c plus the time for the AM tag access (t_a) plus the time for memory access (t_m) plus the time for cache fill ($t_{c'}$), assuming that the processor restarts after filling the missing line into the cache. For clarity of analysis, we add the AM tag (t_a) to the latency although it can be overlapped with the AM data access (t_m). In the second case, the latency is $2t_c + t_a + 3t_n + \alpha + P_c^m t_a$. Here, we assume that the time for memory access and cache fill of the missing line are overlapped with the coherence action. In summary, the local-memory hit latency, L_{lm}, in COMA and DYMA is $(1 - P_{lm}^s)(t_c + t_a + t_m + t_{c'}) + P_{lm}^s(2t_c + t_a + 3t_n + \alpha + P_c^m t_a)$, where P_{lm}^s is the probability that, when there is a local-memory hit, the reference is a write access to a shared line.

Unlike in COMA and DYMA, each directory entry for a given memory line in CC-NUMA has information about the state of the line as well as directory information (owner node and shared nodes) about the line. Thus, all cache misses are sent to the directory in CC-NUMA although the misses could be satisfied at its local memory. However, COMA and DYMA do not need directory accesses for misses satisfied at its local memory. To check the state of lines in the local memory in CC-NUMA, directory processing time is required. L_{lm} of COMA and DYMA is longer than that of CC-NUMA due to the necessary coherence action at the AM and the DM. When invalidating the copy of a line at a remote node, access to the DM tag is not always necessary in DYMA, while COMA always needs it due to the inclusion property, though the access time is not on the critical path.

Remote-cache/memory hit: If neither a local-cache hit nor a local-memory hit happens in COMA and DYMA, the missing line is copied from a remote node. If the line is in the remote cache, the remote-cache hit latency, L_{rc}, is t_c plus t_a (assuming no coherence miss), plus the time for two network traversals ($2t_n$) (sending a request to the home directory and sending a request from the home directory to the remote node where the line exists), plus the directory processing time (α), plus the time for remote-cache access ($t_c + t_{c'}$) to get a copy of the missing line, plus the time for another network traversal to return the missing line (βt_n), plus $t_{c'}$. We assume that the time for returning the missing line is β times longer than the time for sending a head-only request, due to its large size. We also assume that remote-cache access is done sequentially, i.e., checking the tag of the cache (t_a) proceeds before copying the data from the cache ($t_{c'}$). Here, the copy of a modified remote-cache line does not have to be written back to the remote AM since the shared owner state can represent the state of the line. If the requested copy is in the local memory and not in the cache, the remote-memory hit latency, L_{rm}, becomes $2t_c + P^c_{lc} t_a + 2t_n + \alpha + t_a + t_m + \beta t_n + t_{c'}$. When filling the missing line fetched from a remote node into its local cache, an interesting question arises: should both the address and data of the line or only the address of the line be saved in the local AM concurrently? In pure COMA, one of them should be determined at the line-incoming time. Only the address of the line may need to be saved in the local AM if the processor issues a read-exclusive request.

From the analysis, L_{rc} and L_{rm} of COMA and DYMA are longer than those of CC-NUMA. Also, the coherence actions in L_{lc} and L_{lm} of COMA and DYMA make their node latency longer. Let us assume that there is no difference among the directory processing times (α) in the architectures. Then, the average data access latency of CC-NUMA could be the shortest of the three if partition and distribution of data and page migration [CDV$^+$94, SJG92] are done so well that the local-memory hit rate, H_{lm}, of CC-NUMA becomes close to that of DYMA and COMA. In general, the H_{lm} of DYMA and COMA is expected to be higher than that of CC-NUMA. With higher H_{lm}, DYMA and COMA can reduce the large overhead of remote access, replacing it with the small overhead of local memory access. Therefore, the advantage of COMA and DYMA over CC-NUMA increases as network latency (t_n) becomes large. The cut-off point depends on how much higher the H_{lm}'s of COMA and DYMA are than that of CC-NUMA, given network latency. Although COMA and DYMA have the same memory access latency, the different configurations of data derived from the different binding times cause different hit rates at each cache or local memory, which in turn causes different system overhead.

The main drawback of the latency analysis done in this section is that it does not account for contention for system resources. In particular, the relocation overhead in COMA and DYMA under high memory pressure seems to be essential to performance. Thus, in order to evaluate the potential performance of the DSM architectures more accurately, we use another measurement technique which can measure the contention overhead for system resources. In the following chapter, a simulation technique which models contention is described in detail. Using simulations, we can also see the memory hit rates, especially H_{lm}, under the patterns of reference in real applications.

Chapter 5

Simulation

As a performance measurement technique, we developed a program-driven simulator, and use the Splash-2 programs [WOT$^+$95] as workloads. The purpose of developing the simulation is not only to evaluate the performance of DSM architectures but also to check the correctness of the protocol for DYMA. From the results of the simulation, we found that the protocol is operational and can at least be implemented with software in real machines.

5.1 Benchmark programs

This section briefly describes the benchmark programs that will be run on the simulated machines. There are some questions that might be needed to justify the use of the benchmark programs.

- Are the benchmark programs representative of the applications that are run on real machines?
- How many processors are adequate to fully exploit inherent parallelism in each benchmark program?
- Could the algorithms, the data structures, and the problem sizes in each benchmark program be biased towards or against an architecture?

Although this thesis does not rigorously address these justification questions, the simulation results provide some clues to answer the questions.

Barnes: The Barnes application models the interaction of a system of bodies (galaxies or particles) in three dimensions (3D) over a number of time-steps. In each time-step, the positions and velocities of all n bodies are changed. At a time-step, if we express the gravitational forces ($G\frac{m_1 m_2}{r^2}$) exerted on a body by all of the other bodies, the computational complexity for the n bodies is $O(n^2)$. To reduce the complexity to $O(n \log n)$, the Barnes-Hut hierarchical N-body method utilizes the fact that the force of gravitational interaction drops rapidly with distance. Thus, a large group of bodies sufficiently far from a body can be considered as a single equivalent body.

The Barnes-Hut algorithm represents the 3D space as an octree (at most eight child nodes). The root of the tree represents the whole space. A child of a node in the tree represents a subspace (or subcell) of the space represented by the parent node. Each subcell has at least one body. All nodes in the tree are traversed once per body to compute the force exerted on a given body. The traversal starts from the root of the tree. If the visiting node (cell) is far enough away from the body, the bodies in the cell are represented as a single body. Otherwise, each of its subcells is visited [CSG97].

Cholesky: Cholesky factorization finds a real upper triangular matrix R such that $A = R^T R$, where matrix A is symmetric and positive definite[1]. Since the Cholesky decomposition preserves symmetry, only the $n(n+1)/2$ vector (packed form) is needed. Also, pivoting is unnecessary. The blocked sparse Cholesky factorization kernel is similar to the LU factorization kernel in structure and partitioning. However, since the Cholesky factorization kernel operates on sparse matrices, it creates relatively less computation and more communication than the LU kernel. Another difference is that the Cholesky factorization kernel is not globally synchronized between steps [WOT$^+$95].

FFT: The following shows the 1D unordered, radix-2 Fourier transform. Given a sequence $< X_0, X_1, ..., X_{n-1} >$, the discrete Fourier transform (DFT) is the sequence $< Y_0, Y_1, ..., Y_{n-1} >$, where

$$Y_i = \sum_{k=0}^{n-1} X_k \omega^{ki}, 0 \leq i < n. \tag{5.1}$$

In the equation, ω $(= e^{2\pi\sqrt{-1}/n})$ is the primitive n^{th} root of unity in the complex plane. Since the computation of each Y_i requires n complex multiplications, the sequential complexity of DFT is $\Theta(n^2)$. The fast Fourier transform (FFT) is based on following formula:

$$Y_i = \sum_{k=0}^{(n/2)-1} X_{2k}\tilde{\omega}^{ki} + \omega^i \sum_{k=0}^{(n/2)-1} X_{2k+1}\tilde{\omega}^{ki}, \tag{5.2}$$

where n is a power of two, and $\tilde{\omega} = \omega^2$. Since each of the two summations on the right-hand side in Equation (5.2) is a DFT of length $(n/2)$, it leads to the recursive FFT. Since 2^m

[1]An $n \times n$ real symmetric matrix A is *positive definite* if it is true that $\vec{x}^T A \vec{x}$ for *any* vector $\vec{x} \neq \vec{0}$. Alternatively, A is positive definite if all of its eigenvalues are positive.

FFTs of length $n/2^m$ are computed at the m^{th} level and the maximum number of levels is $\log n$, the overall sequential complexity of FFT is $\Theta(n \log n)$. More details about the parallel algorithms for FFT can be found in [KGGK94, WSH94].

FMM: The FMM simulates the interaction of a system of bodies in 2D over the time dimension using the adaptive Fast Multipole Method. Unlike Barnes, FMM traverses the tree in a single upward and downward pass per time-step [WOT+95].

LU: The LU kernel factorizes a dense matrix A as the product of a lower triangular and an upper triangular matrix ($A = LU$) using Gaussian elimination. The dense $n \times n$ matrix A is partitioned off in an $x \times x$ array of $y \times y$ blocks ($n = xy$) to exploit temporal locality on submatrix elements. The block size y is chosen to be large enough to keep high locality of reference, and small enough to maintain good load balance [KGGK94, WOT+95].

Ocean: For computer simulations of the motion of water currents in an ocean basin, the continuous physical problem is discretized in both space and time, i.e., the continuous equations for ocean movement are transformed into discrete functions. The ocean basin is modeled as a grid of equally spaced points. Important variables (temperature, pressure, friction, velocity, various currents, etc.) are associated with each grid point. The equations of motion are solved based on the values of the variables in the grid points at the current time-step, and the information of the grid points is updated as a result. The equations of motion are solved again at the next time-step. Every time-step itself consists of several computational phases. The first phase sets up values, and then solves partial differential equations. To solve the discrete functions at the second phase, an iterative method is used. Since all the phases involve sweeping through all grid points, how to make the sweeping efficient and parallel is the important issue of the program [WSH94].

Radiosity: The radiosity application computes the global illumination in a scene. The diffusely reflecting polygonal surfaces which describe the scene are divided into subsurfaces or patches as necessary to improve accuracy. The radiosity of a patch can be represented as a linear combination of the radiosities of all other patches. Thus, to compute the equilibrium radiosity in a linear system of equations, the application uses the iterative hierarchical method. At each iteration, computing the radiosity starts from the roots (polygons). If necessary, the visiting or interacting node is subdivided into four child nodes (patches). Each node has an interaction list of nodes for different polygons which are potentially visible from the node. The child nodes acquire the interaction lists when they are created. The quadtrees for the polygons are traversed depth-first. The area-weighted patch radiosities are added via an upward pass through the quadtrees. If the overall radiosity has not converged, the next iteration starts.

Radix: The radix kernel sorts the integer keys, relying on the radix r representation of the keys. At each iteration, the sorting performs on a radix r digit of the keys from right to left with stable state.

Raytrace: By using ray tracing, this application renders a 3D scene into a 2D image as seen from a specific viewpoint. The scene is represented as a Hierarchical Uniform Grid (HUG), a tree similar in structure to the octree used in the Barnes. The image being rendered is represented as a two-dimensional array of pixels whose color, opacity, and brightness values are to be computed. Initial rays are cast from the specific viewpoint through every pixel in the image plane and into the scene. The first object that a ray encounters is found by traversing the HUG. At the object, the ray is first reflected toward every light source to determine whether it is in shadow from the light source. If it is not, the contribution of the light source to its color and brightness is computed. Since the ray is also reflected from and refracted through the object as appropriate, it spawns new rays, which in turn do the same procedure for every object that they encounter. Thus, each initial ray generates a tree of rays. Ray tracing affords several tradeoffs between execution time and image quality.

Volrend: This application (volume rendering) produces a 2D image from a 3D array (volume) of sampled scalar data. Each element (voxel) of the array represents a small cube in space. By shooting a ray through the volume for each image pixel as in the ray casting technique, the values (color and opacity) of a pixel are computed by integrating the values (computed from the scalar data) of the voxels along the straight ray. At this time, blending the projected samples is based on the volume rendering equation reduced from an energy balance equation. To find the voxels interpolated along the ray, an octree data structure is used for the volume. By using the technique (called early ray termination) which terminates the ray as soon as the accumulated ray opacity reaches a threshold, volume rendering can be accelerated.

Water-Nsqr: This application computes forces and potentials in a system of water molecules in the liquid state. Pairwise interactions among molecules are required to compute the total potential. Thus, the complexity is $O(n^2)$. By computing the interactions over a series of time-steps, the system can reach a steady state.

Water-Sp: This application solves the problem of the N-body molecular dynamics more efficiently than Water-Nsqr. To compute the total potential, an $O(n)$ algorithm is used.

Program	Problem Size	Footprint (bytes)	Parallelism
Barnes	16K particles	$\approx$ 3540K	$\approx$ 91 %
Cholesky	d750.O	$\approx$ 12144K	$\approx$ 17 %
FFT	64K points	$\approx$ 3764K	$\approx$ 6 %
FMM	16K particles	$\approx$ 29512K	$\approx$ 73 %
LU(Cont.)	576×576 matrix	$\approx$ 2804K	$\approx$ 44 %
	640×640 matrix	$\approx$ 3384K	$\approx$ 43 %
Ocean(Cont.)	258×258 ocean	$\approx$ 16024K	$\approx$ 99 %
Radiosity	room	$\approx$ 29404K	$\approx$ 55 %
Radix	1M integers	$\approx$ 10236K	$\approx$ 98 %
Raytrace	teapot	$\approx$ 3892K	$\approx$ 14 %
Volrend	head (render only)	$\approx$ 22480K	$\approx$ 15 %
Water-Nsqr	1000 molecules	$\approx$ 3744K	$\approx$ 77 %
Water-Sp	1000 molecules	$\approx$ 852K	$\approx$ 29 %

Table 5.1: Characteristics of the benchmark programs

5.2 Program characteristics

In order to grasp the simulation results of a program, we need to know some profiling information about the program such as the amount of parallelism and its footprint in main memory. The profiling information about the Splash-2 programs is illustrated in Table 5.1. We acquire the information from the simulations with default machine parameters although the information varies very little based on machine parameters and architectures. The formula we use for calculating the parallelism of a program is as follows:

$$\left(\frac{\sum_{i=0}^{n-1} I_i}{\max(I_0, I_1, ..., I_{n-1})} - 1\right)\left(\frac{1}{n-1}\right), \tag{5.3}$$

where n is the number of processors and I_i is the number of instructions executed in processor number i. Although more accurate parallelism of a program can be computed by summing the weighted parallelism at all synchronization points, we are able to apply this formula once at the end of execution. Also, the number of references is used instead of the number of instructions. Despite these approximations, the numbers in the table can give us some hints about the execution time of the programs. Cholesky, FFT, LU, Radiosity, Raytrace, Volrend, and Water-Sp have relatively long serial execution sections. Although it seems unlikely to expect big performance differences among the architectures with those programs, we need to execute the programs since different architectures may well have different results even on those programs with low parallelism.

System Unit	Parameters	CC-NUMA	COMA	DYMA
number of data states		4/2 states	5/4 states	12/4 states
cache	size (per node)	16K bytes		
	access time (t_c)	1 cycle		
	cache fill time ($t_{c'}$)	4 cycles		
	associativity	4-way		
	replacement	random selection		
memory	(total) size	data size	73.0∼94.9% mem. pr.	
	access time (t_m)	24 cycles		
	tag access time (t_a)	8 cycles		
	associativity	-	8-way	
	replacement	-	priority selection	
directory	home node assignment	page interleaving (rr) and page fault (ft)	line interleaving (li)	
	processing time (α)	8 × (# of DRAM accesses) cycles		
	protocol	non-blocking		
network	type	multibus (4 buses)		
	service time (t_n)	30/10 cycles with/without data ($\beta = 3$)		
number of processors = 32, page size = 4K bytes, line size = 64 bytes				

Table 5.2: Default machine parameters

5.3 Simulation model and parameters

MINT [VF94] is used as the front end of our simulator. The simulator is running on a SGI challenge machine. To accurately simulate the data access mechanism for each architecture, as the back end of our simulator, we modeled the contention through a FIFO queue at each cache, local memory, directory, and network (see Figure 4.2). If a request arrives at a system resource which is busy serving other requests, the request waits for the completion of the services at the queue of the resource. No request for the use of resources is rejected due to lack of buffers. We also assume that the instruction access always hits in an instruction cache. Each non-memory instruction completes in a single cycle as assumed in MINT. The page fault cost is assumed to be zero while all the lines of a page are initially loaded to the same node. We use program execution time as our performance metric.

In chapter 4, we chose a hardware organization and a control protocol for each architecture. For the simulation, we have to resolve other machine parameters. The default parameters are shown in Table 5.2. The default cache size (16K bytes) is selected so as to fit between the first and the second important working sets of the benchmark programs used in our simulation. The working sets of applications continue to grow, and there are hundreds of applications whose working sets are too large to be supported even with main memory. This is the reason the cache size was chosen even though it is smaller than the size of the cache in a real machine. The cache access time (t_c) is the unit of our simulation clock cycle. Four cycles are assumed for the cache fill time ($t_{c'}$). By default, four-way set associativity for the cache is assumed. The line for replacement is randomly selected. The protocol for the cache of DYMA needs more hardware functions than that of CC-NUMA. Since the additional functions are for relocation and can be done without increasing the number of tag accesses, we assume that service time at the cache is the same in both DYMA and CC-NUMA. In COMA, achieving modulization seems to be difficult due to the inclusion property. However, in the following discussion, we assume that probing the cache from the AM can be done without any penalty even when the cache is busy serving other requests.

If the number of AM sets in COMA is not an integer times larger than that of the cache sets, an anomaly may occur: two addresses which do not conflict at the cache cause a conflict at the large AM. Also, the number of DM sets should be an integer times larger than that of cache sets for the control protocol of DYMA to work correctly. Thus, we choose the size of the local memory to be a multiple of the cache size. We made the memory access time (t_m) relatively large, because we believe that the processor-memory performance gap is widening not narrowing. We also assume that access to the data of the local memory starts concurrently with access to the tag of the local memory. That is, the tag access time (t_a) is overlapped with the memory access time (t_m). Here, note that the tag of the memory is assumed to be slow DRAM to minimize the memory cost. In the local memories of COMA and DYMA, each set has 8-way associativity by default. For replacement, a copy is selected with the following priority sequence: an invalid copy, a shared non-owner copy, a shared owner copy, and a private owner copy.

For more realistic simulation, the implementation choice for the directory is also considered. We use DRAM for the address table in the directory. The pointer pool which saves the pointer information is implemented with fast SRAM in CC-NUMA while it is implemented with slow DRAM in COMA and DYMA. In the table, the directory size overhead compared to the local memory comes from a 512 multiprocessor configuration with a 4-way 8M byte cache and a 32-way 512M byte local memory in each node. The number of DRAM accesses is counted for the processing time (α) at a directory since SRAM access time is not significant and can be overlapped with the DRAM access time. Eight cycles for each DRAM access are assumed. In most cases, there are fewer DRAM accesses in CC-NUMA than in COMA and DYMA. For instance, a read request requires 4 DRAM accesses in COMA and DYMA, while 2 DRAM accesses are needed in CC-NUMA. Also, if an invalidation request needs to send signals to invalidate copies shared by 32 nodes, the total directory processing

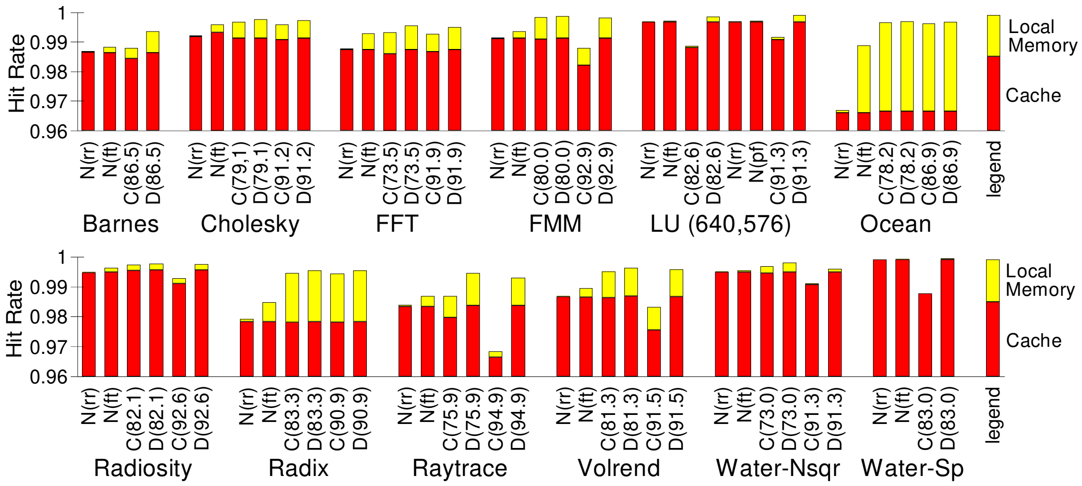


Figure 5.1: Node hit rates

time is 272 cycles in COMA and DYMA, compared to 16 cycles in CC-NUMA under our simulation.

The service time (t_n) for a network request which does not have data takes ten cycles, and the size is assumed to be four bytes. Service for a network request with data needs 30 cycles (βt_n), and its size is 68 bytes (64 bytes for data and 4 bytes for head information). A 32 processor configuration organized with one processor per node is simulated. Four network buses are used to connect the 32 nodes. The size of a line is 64 bytes, and the page size is 4K bytes.

5.4 Results and analysis

5.4.1 Overall results

Since a node hit substitutes a local data access for a remote data access, it makes data access latency short. Further, it changes the amount of network bandwidth usage, the number of accesses to the cache and the local memory, and the processing overhead at the directory. Thus, the node hit rate of an architecture is a solid predictor of how well the architecture will perform. Figure 5.1 shows the average node hit rate per reference (H_{lc} and $M_{lc}H_{lm}$) in the three DSM architectures. In the figure, N, C, D denotes CC-NUMA, COMA, and DYMA respectively. CC-NUMA(rr) places the missing page of the application data into a node in round-robin (rr) fashion (i.e., page interleaving). We also simulate another version of CC-NUMA(ft) where a page is allocated on the node which causes the page fault (ft). If the local memory is out of space when a new page fault happens, then we have to select an existing page for replacement. To maintain a working set of a program, the replaced page might be relocated to a remote node. For the relocation, we need a mechanism to clean up the directory information for the page and to select an appropriate remote node for efficiency. Although the mechanism for the replacement in CC-NUMA(ft) is important, we do not take account of the mechanism in our simulation due to the difficulty of designing an accurate timing model and a correct protocol. Thus, the local memory size in CC-NUMA(ft) varies among the nodes in our simulation. In the Splash-2 benchmark programs, there are recommendations for how to place pages initially. Although this might yield an improvement over other page placement schemes, we did not follow the recommendation because the recommended page placement requires predicting the reference patterns of a program before the execution of the program, which is the feature of the message passing architectures. The memory pressures in COMA and DYMA are represented by the values in the parentheses. We use different problem sizes in the LU benchmark program to get the simulation results with two different memory pressures. Problem sizes of 576×576 matrix and 640×640 matrix are selected for high and low memory pressures respectively.

Since different binding times lead to different data arrangements, there is diversity in the node hit rates of a program running on the architectures as shown in Figure 5.1. The size of the local memory in COMA is slightly larger than the size of the cache in some programs such as Barnes, FFT, LU, Raytrace, Water-Nsqr, and Water-Sp (see the footprints in Table 5.1). Thus, the redundancy of the data in the local memory is high. This explains why COMA has lower or only slightly higher node hit rates than CC-NUMA in those programs. Further, the inclusion property in COMA is negative to the cache hit rate while the cache hit rates of DYMA are almost equal to those of CC-NUMA. This is because the data arrangement at the caches of CC-NUMA and DYMA is not affected by the replacements. In DYMA, the random replacement algorithm at the cache does not weaken locality of data at the local memory. Since an architecture with higher memory pressure has smaller caching space and higher possibility of replacements, the node hit rates of COMA and DYMA decrease as memory pressure increases. However, the amount of degradation is different. The degradation in COMA is intolerable due to the lower cache hit rate. Low memory pressure in COMA is unattractive since it makes caching space of main memory for the data in disk small. The local memory hit rates of CC-NUMA(ft) are higher than those of CC-NUMA(rr) but smaller than those of DYMA. This implies that careful page placement at the beginning can increase node hit rates, and that few programs have strong patterns of reference where most of the lines are not referenced again after being replaced from the cache. As can be seen, the relative local memory hit rate (H_{lm}) in COMA and DYMA is not as high as the cache hit rate (H_{lc}) especially when the cache hit rate is high.

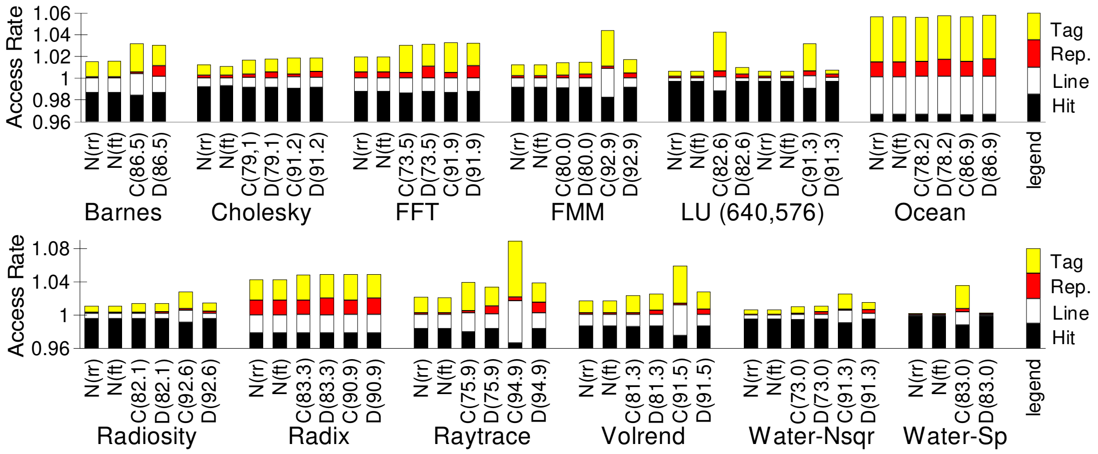


Figure 5.2: Number of accesses to the cache per reference

In other words, the higher the cache hit rate is, the lower the local memory hit rate is. Thus, unlike bindings to the cache, the overhead for bindings to the local memory cannot be easily amortized by faster data access latency. The question is whether the increment in the node hit rate can compensate DYMA for the extra overhead caused by binding at a cache miss time.

One type of system overhead is the number of accesses to the cache. The average number of accesses to the cache per reference is shown in Figure 5.2. We divide the accesses into four different cases; hit access, line access, replacement access, and tag access. The hit access happens when a reference from the processor hits at the cache. At this time, a datum finer than a line may need to be fetched from the cache. To copy a line from the cache for replacement, the replacement access is used. All other types of accesses for the cache lines are counted as line accesses. This includes the cache fill and the copy of a cache line for a remote access. If an access searches just the tag and state of the cache, it is classified as a tag access. The cache miss accesses for local and remote data requests, and the accesses for invalidation and relocation belong to this class. As can be seen in the figure, the number of replacement accesses in DYMA is usually higher than in CC-NUMA and COMA (particularly in Barnes, FFT, and Raytrace). This is because unlike the others DYMA additionally needs the access to the cache to have a copy of the clean line for relocation to the local memory. Also, for modulization, the control of data states at the cache in DYMA increases the non-inclusive lines although the increment can be eliminated in an ideal control mechanism. All node misses in COMA and DYMA require accesses to a remote cache while some node misses in CC-NUMA can get a valid copy from the local memory of the home node without an access to the remote cache. Also, to relocate the replaced line to a remote node, accesses to the tag and state of the cache are required in COMA and DYMA. This is the reason why the tag and line accesses in COMA and DYMA are generally larger than in CC-NUMA. Some of the overheads, such as replacement accesses, can be hidden from the critical path of data access, and some of the overheads, such as remote cache accesses, make data access faster. Thus, the larger number of accesses to the cache in COMA and DYMA does not always make the performance of COMA and DYMA worse than that of CC-NUMA.

In the figure, some irregularities of COMA can be found in FMM, LU, Radiosity, Raytrace, Volrend, Water-Nsqr, and Water-Sp. In general, if the hit access rate decreases, the line access rate increases proportionally since the large number of cache misses incurs an equivalent number of cache fills. However, the increment in the line access rate is much higher than what we would expect, especially in FMM, Raytrace, and Volrend. This implies that the number of remote cache hits increases in those programs. One possible reference pattern which can increase the remote cache hit rate is a pattern which actively shares some memory lines. Since COMA imposes a limit on the degree of simultaneous sharing of a line under memory pressure, all of the actively shared lines cannot reside in the nodes simultaneously. Thus, a node which does not have the shared line retrieves the shared line from the cache of a remote node since the line is actively referenced by the remote processor. When the node attempts to bind the line to its local memory, an ownership line replaced from the local memory is relocated to a remote node while removing the shared line from the remote node. Since the remote node will reference the removed line immediately, the same action occurs repeatedly. Thus, this kind of reference pattern can increase the overhead of different system resources as well as the line access rate. We will also see the impact of reference patterns on the other system resources.

Figure 5.3 shows the number of accesses to the local memory per reference. In the figure, the accesses for data fetch, save, and relocation are counted in the data access rate. Accesses which only check the tag and state of the local memory are categorized as tag accesses. Thus, accesses for invalidation and unsuccessful accesses are included in this category. In our hardware organization for CC-NUMA, all the cache misses are sent to the directory since the directory has the dirty bit which indicates validity and ownership of data. Thus, no accesses to the local memory tag are found in CC-NUMA. When a cache miss happens, one access to a local memory is required if the missing line exists only in main memory. Also, if the replaced line from the cache is updated, the line is relocated into a local memory. This is the minimal cost in all architectures. CC-NUMA(rr) and CC-NUMA(ft) do not allow other local memory accesses, which is why there is no difference in the access rates of the two. COMA additionally binds newly referenced lines to the local memory at

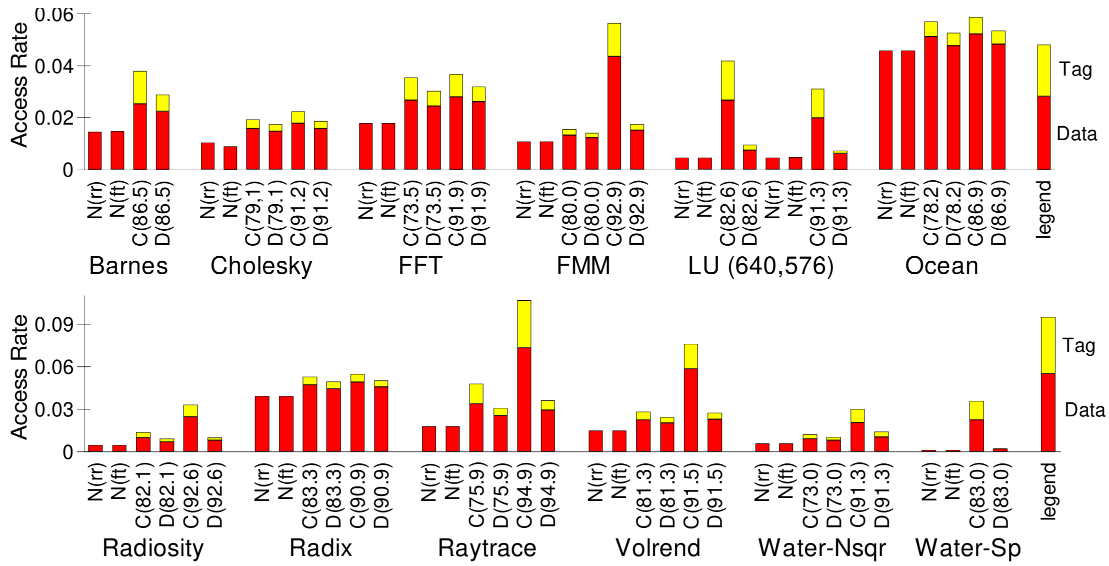


Figure 5.3: Number of accesses to the local memory per reference

a node miss time. Also, the replaced line can increase the number of accesses by more than one. In DYMA, some lines are invalidated at the cache before they are bound to the local memory, and the number of replacements from a node for a new data arrangement at a cache miss time can be smaller than in COMA. Therefore, COMA has the largest number of local memory accesses while DYMA is between those two extremes even with the highest replacement rate from the cache. This means that CC-NUMA has the highest performance, provided that the cost for networking and directory processing is zero. However, network overhead grows as the number of nodes increases. Also, directory processing time is not negligible. Thus, if a higher node hit rate can be achieved by increasing the number of cache and local memory accesses, COMA and DYMA can perform better than CC-NUMA.

The directory overhead is shown in Figure 5.4. In the figure, we divide the requests to the directory into three categories: data requests, replacement requests, and invalidation requests. Whenever a cache miss happens, the data request is sent to the home directory in CC-NUMA. At this time, if the line selected for replacement is valid, a replacement request is delivered to the home directory as well. This is one important reason why CC-NUMA has a higher request rate than DYMA. Since the node hit rates of COMA under high memory pressure are lower than the cache hit rates of CC-NUMA in some programs, COMA does not always have lower request rates than CC-NUMA. Due to the analogous cache hit rates, the number of directory requests is almost the same in both CC-NUMA(rr) and CC-NUMA(ft). Here, note that the number of invalidation requests is not related to

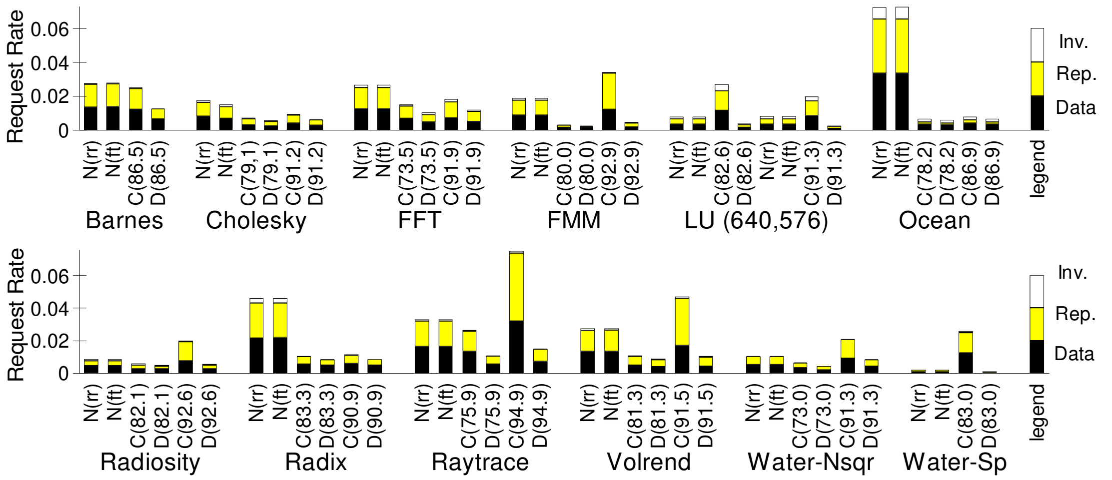


Figure 5.4: Number of requests to the directory per reference

the size of caching space but to the node miss rate. Also, the higher invalidation request rate in CC-NUMA suggests longer local-cache hit latency (L_{lc}).

The results in Figure 5.5 are the amount of bytes needed for network communication per reference. These results are strongly related to node miss rates. Also, data traffic is directly related to remote data access latency. A higher node hit rate can reduce the amount of data traffic, which in turn usually diminishes the amount of replacement and coherence traffic. Even though CC-NUMA has the smallest number of local memory accesses, the amount of network traffic in CC-NUMA is higher than the others, except for some cases of COMA under high memory pressure. Also, CC-NUMA(rr) has a higher traffic rate than CC-NUMA(ft) though they have almost the same cache access rates, local memory access rates, and directory request rates. This means CC-NUMA(ft) performs better than CC-NUMA(rr) if there is little difference in waiting times for system resources. If a program has a small working set, the program will terminate its execution before most of the lines in the working set are replaced from the cache. That is, most of the lines are invalidated before being replaced from the cache. For those lines, the binding to the local memory in COMA at a node miss creates unnecessary accesses to the local memory and unnecessary replacement traffic. Also, the new data arrangement in COMA at binding time increases the node miss rate (see Figure 5.1). This is the reason why COMA fails to reduce network traffic in the small data size programs (Barnes, FFT, LU, Raytrace, Water-Nsqr, and Water-Sp). Furthermore, COMA increases the amount of network traffic far higher than does CC-NUMA in the programs (FMM, Raytrace, and Volrend) which vigorously share some memory lines. The amount of network traffic is small in most of cases in DYMA, which

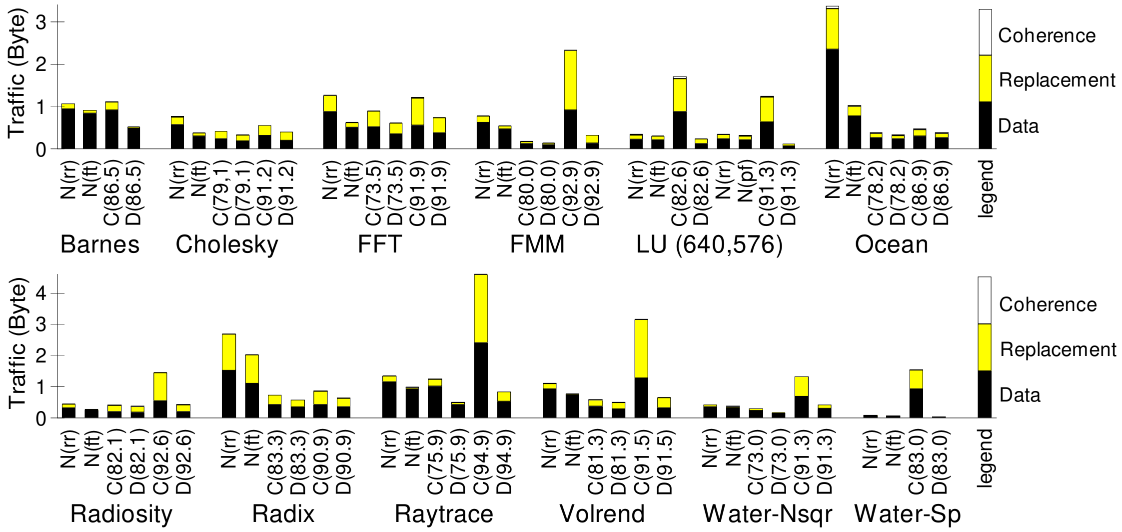


Figure 5.5: Amount of network traffic per reference

in turn minimizes the side effects of replacements. Also, DYMA is robust under various patterns of reference. Usually, a higher replacement traffic rate is found when the data traffic rate is large. However, the ratio of the replacement traffic rate to the network traffic rate is comparatively small in CC-NUMA. The primary reason for this is that COMA and DYMA have more possibilities of selecting ownership copy for replacement from a node than does CC-NUMA. Another reason is that a relocation request in COMA and DYMA usually takes three network traversals while it takes, at most, one in CC-NUMA. That is, the binding to the local memory increases the percentage of replacement traffic although it can reduce the total amount of network traffic.

The average waiting time per request for each system resource is shown in Figure 5.6. Here, note that each system resource is distributed across the system. Generally, there are two possible cases which make waiting time long, given the same number of requests to a system resource. The first case happens when the requests are centralized to one part of the system resource. The second case occurs when the requests are concentrated during a short period of time. Thus, a large number of requests for system resources do not always incur long waiting time. As an example, the average waiting times in CC-NUMA(rr) are shorter than in CC-NUMA(ft) in most cases, even though CC-NUMA(ft) has a lower network traffic rate and there are almost no differences in the other rates. This indicates that the loads on the system resources are not balanced well in CC-NUMA(ft). Indeed, the load is centralized to the master node (Node_0) and the first bus (Bus_0) which serves the requests from the master node in most programs. Thus, the page placement policy in CC-NUMA(ft) seems

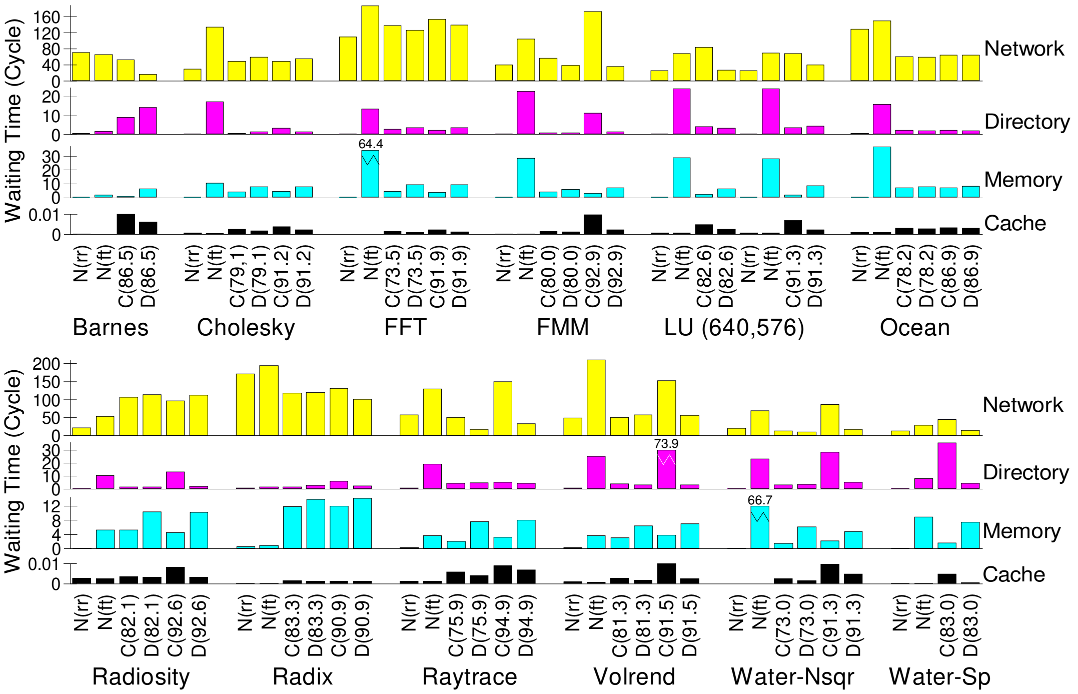


Figure 5.6: Waiting time per request at the system resources

not to be preferable from the perspective of load balancing. Waiting times at the network are considerably longer than the other waiting times. This means the system bottleneck generally occurs at the network. That is, if requests which need network traversals occur frequently during the execution of a program, the execution time can be long. DYMA has a small network traffic rate, and hence relatively short waiting times at the network for most programs. In the next simulation result, we will see how waiting time has an effect on data access latency.

The average data access latency per reference shown in Figure 5.7 consists of four different average latencies: local-cache hit, local-memory hit, remote-cache hit, and remote-memory hit latencies. The average waiting time caused by contention for the system resources is also included in the figure. Thus, the four individual latencies, without the waiting time, in the figure are equivalent to the four terms of Equation (4.1). As can be seen, the average latencies of DYMA are shorter than the others in most cases since DYMA reduces the number of long remote accesses and efficiently replace them with short local accesses. Also, few programs have strong reference patterns which can degrade the performance of DYMA. The average remote latency without waiting time in CC-NUMA(ft) is small compared to that in CC-NUMA(rr). The reason for this is that the node hit rate

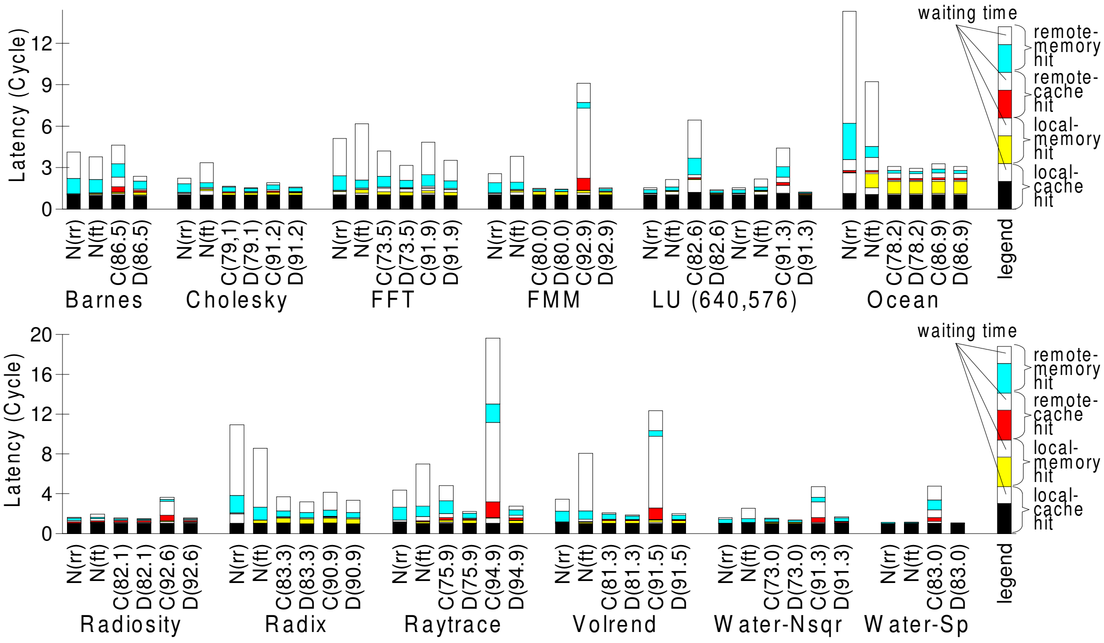


Figure 5.7: Average data access latency including the waiting time

of CC-NUMA(ft) is higher than that of CC-NUMA(rr). However, since a large number of pages are allocated to the master node (Node_0), due to the page placement scheme used in CC-NUMA(ft), the cache miss accesses concentrate on the local memory of the master node. Thus, the waiting time for service is long, at the network, at the directory, and at the local memory (see Figure 5.6). This is the reason why, even with a higher node hit rate, the execution times of some programs in CC-NUMA(ft) are longer than in CC-NUMA(rr). By reducing the number of long remote accesses in COMA, the average remote latency without waiting time can be smaller than that of CC-NUMA. However, heavy contention at the AMs in COMA makes waiting time for an access to the data in the AM longer. Also, the large amount of replacement traffic results in significant delays for remote accesses as well as a high possibility of the occurrence of remote accesses.

Interestingly, the remote-cache hit latencies of COMA with high memory pressure are significantly long, especially in FMM, Raytrace, and Volrend. The main reason for this is the reference patterns of the programs, as we pinpointed previously. In the reference patterns, most of the misses bring a copy of the line from a remote cache since the actively referenced lines are usually in the remote cache. Other noticeable results are observed in the local-cache hit latency of CC-NUMA(rr) for Ocean and Radix. The waiting times in those case are extraordinarily long compared with the others. The reason is that write

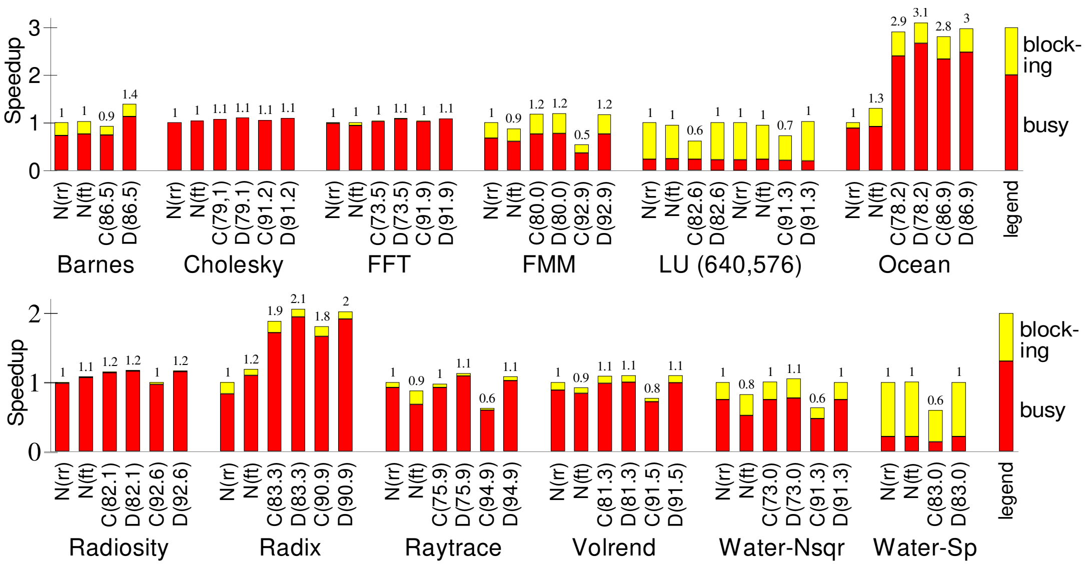


Figure 5.8: Speedup: the ratio of a program execution time to the case of CC-NUMA(rr)

hits on a clean cache copy happen frequently in those programs. In CC-NUMA(rr), the home directory of a clean cache copy is usually in a remote node. Thus, the write hits have to go to the home directory through the network to update directory information. Since the cache cannot hold all of the data in a large working set, the modified copy is easily discarded from the node and relocated to the local memory of the home node. Also, a program with a large working set has a strong likelihood that the discarded line will be immediately loaded again into the local cache with a clean state. Thus, write hits on the clean cache copy would happen recursively. Since waiting time at the network is extremely long, waiting times for requests caused by write hits becomes large. In CC-NUMA(ft), the home directory of the clean line is likely to be the same node due to CC-NUMA(ft)'s page allocation scheme. Thus, few of the write hits are sent to the network. In COMA and DYMA, the modified copy can be relocated to its local memory, and loaded again from the local memory with the private state. Thus, the next write hit on the clean cache copy does not generate network traffic.

Figure 5.8 shows the increase in speed as based on the execution time of CC-NUMA(rr), i.e., speedup $= \frac{\text{execution time of CC-NUMA(rr)}}{\text{execution time of an architecture}}$. A processor which arrives at a synchronization point (wait(), barrier(), lock(S), etc) is blocked until the blocking event disappears. Thus, a large amount of blocking time implies high synchronization overhead in a program even though the blocking time is not directly related to program execution time. The longest blocking time among the processors is presented in Figure 5.8. DYMA is faster than the

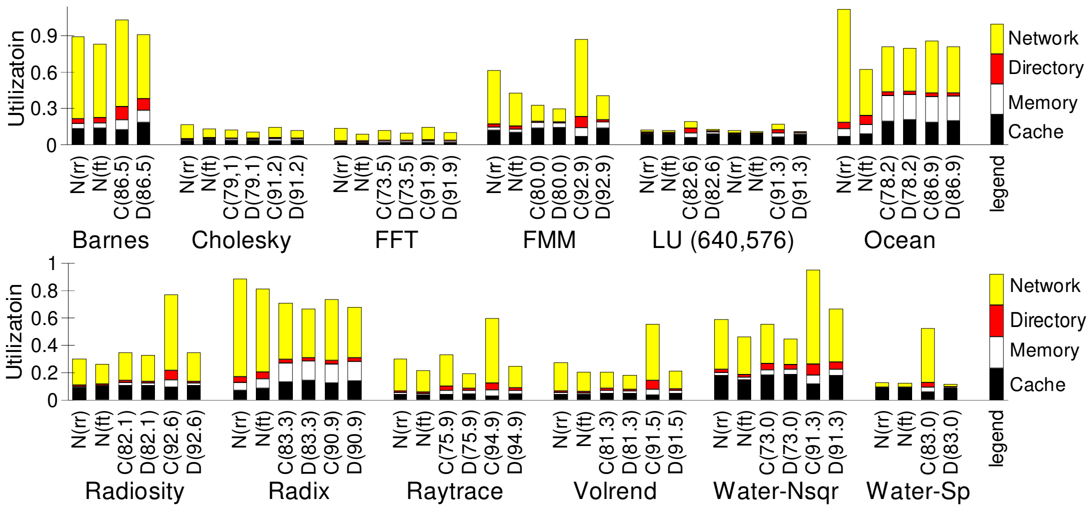


Figure 5.9: Utilization of the system resources

others except in two special cases. The performances of CC-NUMA(rr) in Water-Nsqr and CC-NUMA(ft) in Water-Sp are slightly better than that of DYMA (91.3% memory pressure in Water-Nsqr). As in Figure 5.1, there are almost no local memory hits in those cases. This means that few long remote accesses are replaced by short local accesses in DYMA. In our simulator, when a new page is loaded into main memory, in CC-NUMA the page is saved in the local memory without generating any replacements, while in DYMA the binding of a new page replaces some DM lines, if necessary due to limited amount of a local memory. The speed difference in the two special cases is very small because no page faults happen after all the data of a program are loaded into main memory. There is no winner between CC-NUMA(rr) and CC-NUMA(ft). The reason for this is that the number of local memory accesses is evenly distributed among all nodes in CC-NUMA(rr), while CC-NUMA(ft) has higher local memory hit rates than CC-NUMA(rr). COMA has the biggest variations in performance level. That is, COMA performs worse than CC-NUMA when the working set size is small. The situation is reversed when the working set size of an application is large enough to create a serious network bottleneck in CC-NUMA. Here, note that the performance degradation in DYMA at high memory pressure is small. We found that DYMA performed well even when memory pressure in some programs was greater than 100%.

In summary, CC-NUMA has the smallest number of accesses to the cache and the local memory, while COMA has the largest (see Figure 5.2 and 5.3). At the directory and at the network, CC-NUMA has the highest overhead except for some cases of COMA, while DYMA has the lowest overhead (see Figure 5.4 and 5.5). In CC-NUMA, the page-fault

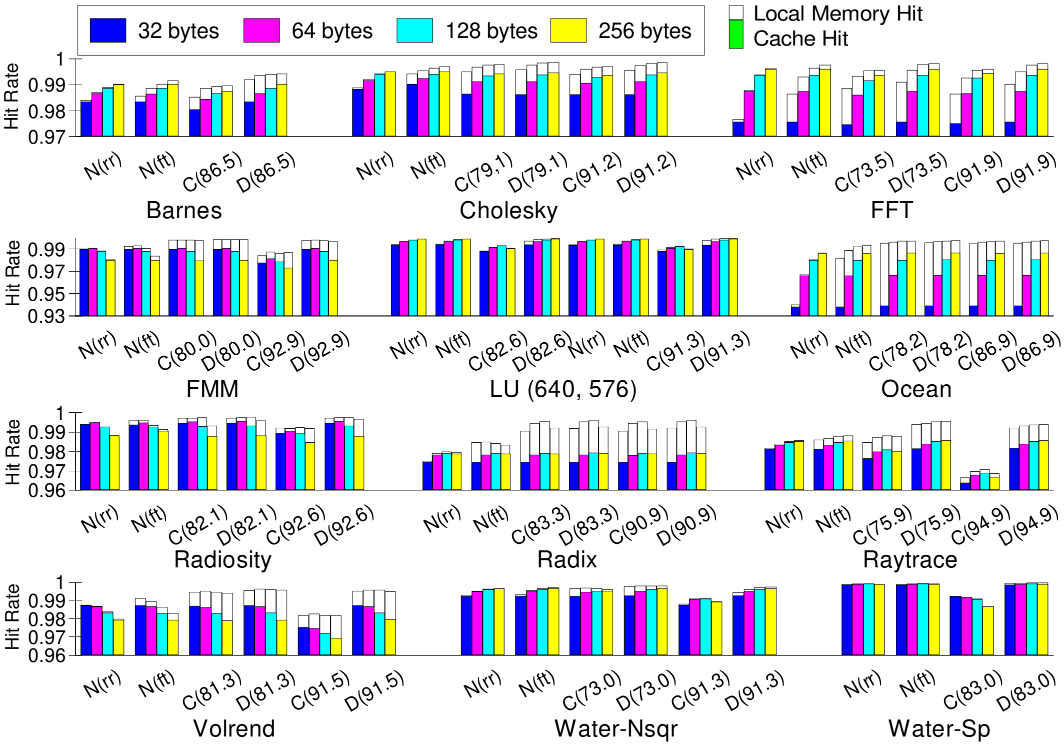


Figure 5.10: Node hit rates varying the line size from 32 bytes to 256 bytes

placement scheme reduces the networking overhead (see Figure 5.5). However, the round-robin placement scheme balances the workload equally across the system resources (see Figure 5.6). Utilization ($= \frac{\text{busy time}}{\text{execution time}}$) of the system resources is shown in Figure 5.9. For most of the programs, DYMA's utilization is as high as that of COMA although system overhead in DYMA is lower than that in COMA. This suggests that DYMA utilizes system resources more effectively. CC-NUMA has the least utilization of the local memory, and has relatively high utilization of the network. This indicates that CC-NUMA has a bias on the use of the system resources.

5.4.2 Line sizes

In order to see how different line sizes affect the node hit rate, we simulate the DSM architectures varying the line size from 32 bytes to 256 bytes. Even though larger line size requires longer data access latency and higher overhead in handling the line at the system resources, we did not change any other machine parameters. The simulation results are shown in Figure 5.10. A large line size does not always increase the cache and local memory hit rates especially when the hit rates are high. Also, the variance of node hit

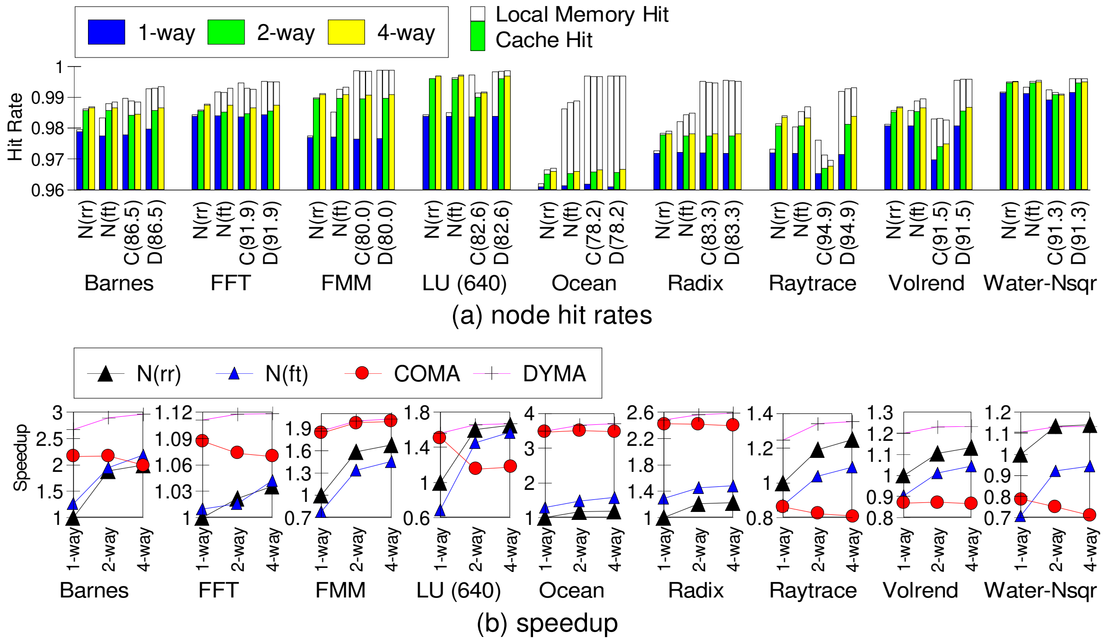


Figure 5.11: Performance varying cache set associativity from 1-way to 4-way

rates is smaller than that of cache hit rates. Generally, a large line size causes high false sharing effects and low temporal locality. Another reason for the results is that the negative effect on the locality of data caused by the replacement grows as the number of lines in the local memory decreases. From the above observation, we conclude that a large line size is preferable only when the cost of handling the large line is nearly equal to the cost of handling a small line. This rule can be applicable to all three architectures.

5.4.3 Cache set associativities

We also study performance sensitivity to the set associativity of the cache. Generally speaking, a high set associativity improves hit rates by reducing conflict misses. Figure 5.11 (a) plots node hit rates along with the cache set associativity from 1-way to 4-way. The local memories in COMA and DYMA are set to 8-way set associativity. Since the number of sets in the local memories of COMA and DYMA is restricted to a multiple of the number of cache sets in our simulator, maintaining the same memory pressure while varying associativities is not possible in some programs. Thus, the figure shows only the programs in which memory pressure is not changed with the different set associativities. As expected, cache miss rates drop by the 2:1 cache rule of thumb. However, interestingly, the node hit rates of COMA tend to tilt downward as the associativity increases. A higher cache hit rate reduces the frequency of references to the local memory. The lower access rate increases the miss rate at the local memory.

In COMA, the reduction scale of the local memory hit rate is larger than the increment scale of the cache hit rate. This is because the negative effects of relocating data in a remote memory become stronger than the positive effects of binding data to the local memory as the frequency of references to the local memory decreases. This explains why COMA degrades the speedup in most of the programs as the cache associativity increases (see Figure 5.11 (b)). In the figure, the speedup is based on the program execution time of CC-NUMA(rr) with 1-way set associativity. Since the node hit rate of CC-NUMA is strongly controlled by the cache hit rate, CC-NUMA has the highest increment rate of speedup. Although the node hit rates of DYMA look invariant, DYMA's performance improves since some long local memory accesses are replaced by short cache accesses.

5.4.4 Memory set associativities

Another factor that has a first order effect on the performance of the architectures is the set associativity of the local memory in COMA and DYMA. Higher set associativity of the local memory can reduce replacement traffic as well as node conflict misses. Figure 5.12 (a) shows the sensitivity of node hit rates to the set from 1-way associative to 8-way associative. The default 4-way set associativity in the cache remains unchanged. The figure indicates that the cache hit rates of COMA are highly sensitive to the local memory set associativity. This suggests that a large number of the replacements can be reduced in COMA with high set associativity of the local memory. Unusually, the node hit rate of COMA in LU is descending slightly as the associativity is increasing. Also, in Cholesky COMA with some high set associativities has smaller local memory hit rates than with low set associativities. This is because the relocation algorithm at the directory searches the master node (node number 0) first to find available space (i.e., a higher priority is given to a lower node number). This makes the cache and local memory hit rates smaller even at a higher set associativity.

Since there are no negative effects on locality of data in the cache of DYMA, DYMA seems to be robust on the associativity of the local memory. Further, the associativities of the cache and the local memory can act together to reduce conflict misses in DYMA. The increment of local memory hit rates (H_{lm}) in both COMA and DYMA is small for most of the programs. Figure 5.12 (b) shows the speedup as compared to the program execution time of COMA with 1-way set associativity. In the legend of the figure, COMA(low) and DYMA(low) mean lower memory pressure than COMA(high) and DYMA(high) while each pair has the same memory pressure under the different set associativities. COMA has a

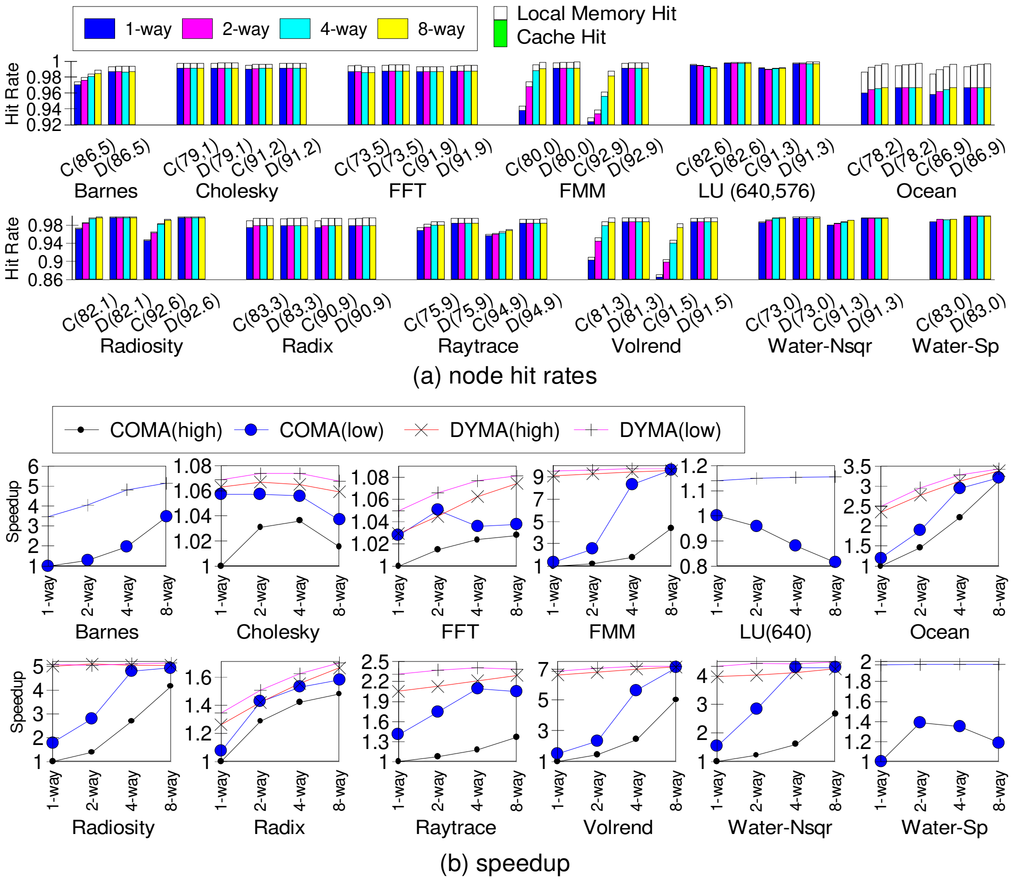


Figure 5.12: Performance varying local memory set associativity from 1-way to 8-way

faster increment rate than DYMA in several programs.

5.4.5 Node scalability

Figure 5.13 shows the simulation results when the number of nodes varies from 4 to 32. The other simulation parameters are set by the default values. At differing numbers of nodes, the architecture has differing memory pressures. Except for a few programs, locality of reference at the cache appears to be invariant with the different number of nodes. In Cholesky, the cache hit rate is increasing while it is decreasing in Radiosity. However, at the local memory, the hit rate is reduced enough to make the node hit rate low in most programs. In our simulation, the problem size of a program remains the same regardless of the number of nodes. Therefore, the size of the local memory in each node is reduced as the number of nodes increases, which explains why the local memory hit rate is decreasing.

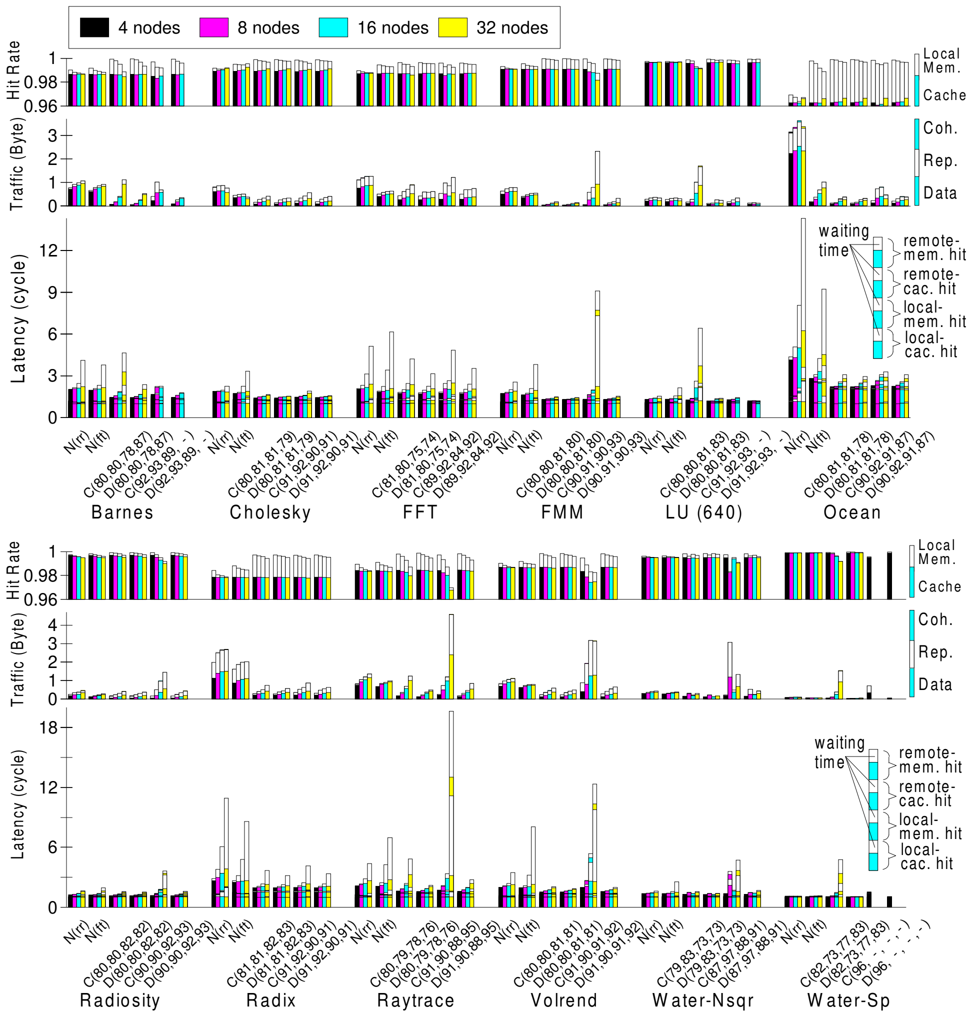


Figure 5.13: Simulation results varying the number of nodes

Although the increment of network traffic per reference in each node is not significant, the increment in the waiting time portion of data access latency is remarkable because the network bandwidth remains the same. This means that the impact of remote accesses on performance is extremely critical when the number of nodes is large. Also, our simulation results show that the maximum processor stall (or blocking) time is increasing as the number of nodes grows.

When the number of nodes is four, each node has a dedicated network bus to send out network requests. Thus, network contention caused by network arbitration among the nodes does not happen. However, there is still the possibility of contention when network requests are concentrated on a single node. Yet, even if the requests are concentrated on a single

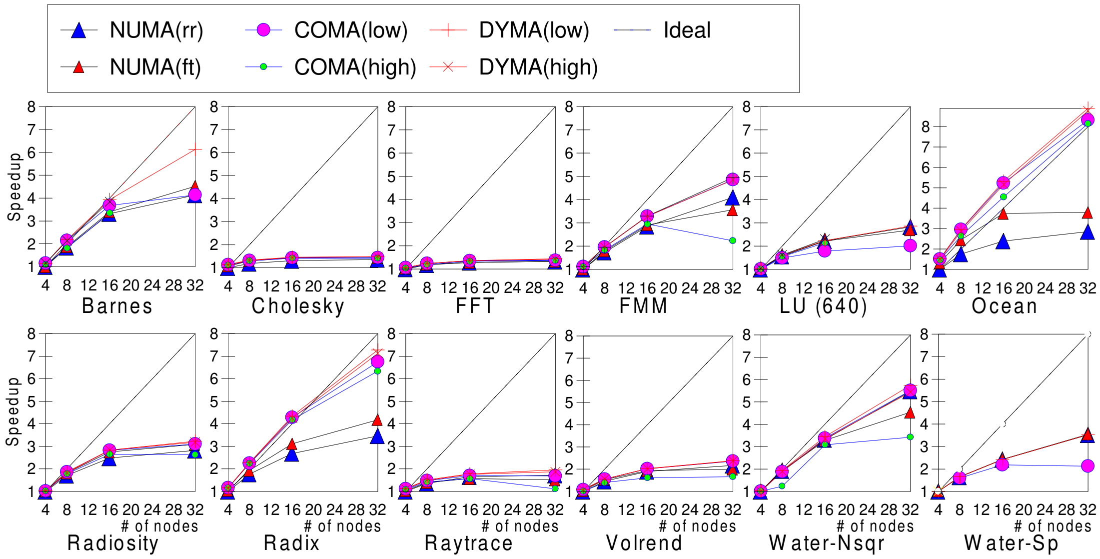


Figure 5.14: Speedup sensitivity to the number of nodes

node, the contention does not seem to be serious, given the network service time due to the small number of nodes. This is the reason the data access latency includes almost no waiting time. Thus, CC-NUMA(ft) always has shorter data access latency than CC-NUMA(rr), i.e., the reduction of waiting time resulting from the well balanced workload in CC-NUMA(rr) is less important than the reduction of remote accesses in CC-NUMA(ft) under the small number of nodes. However, if the requests were generated from a large number of nodes, the concentration of requests on a single node in CC-NUMA(ft) could make the contention serious. This is the reason why when the number of nodes is 32 CC-NUMA(ft) has longer data access latency than CC-NUMA(rr) in some programs.

Figure 5.14 depicts the speedup using the execution time of CC-NUMA(rr) with 4 nodes as a base of comparison. In the legend of the figure, COMA(low) and DYMA(low) indicate lower memory pressure than COMA(high) and DYMA(high), while each pair has the same memory pressure. There is a very small speedup in the programs (Cholesky, FFT, LU, Radiosity, Raytrace, Volrend, and Water-Sp) which have relatively long serial execution sections (see Table 5.1). However, FMM and Water-Nsqr have some speedup although the parallelism of the programs is not sufficient to get an ideal speedup. In CC-NUMA and COMA, the speedup of Barnes saturates when the number of nodes is 16 since the node hit rate of Barnes is not high enough to move the saturated point beyond 16. In fact, the node hit rate of Barnes decreases faster than those of the other programs as the number of nodes increases. For Ocean and Radix, ideal speedup is achieved in both DYMA and COMA while the speedup of CC-NUMA saturates even before 16 nodes since its low node hit rate causes a network bottleneck. In general, the performance gap among the architectures is growing as the number of nodes increases.

5.4.6 Network bandwidths

To see how different network bandwidths affect system performance, we consider two network parameters: network service time and number of buses. We can reduce the bus clock cycle time by placing the system nodes very closely. Also, a wide bus can transfer more bytes in a bus cycle than a narrow bus. A network which has a large number of buses can serve a large number of requests at the same time.

Figure 5.15 shows the simulation results when the network service time varies. In the figure, *n0, n1*, and *n2* mean that the network service times are 0/0, 30/10, and 40/20 cycles with/without data respectively. Since different network service times do not cause different data arrangements in the cache or in the local memory, there is almost no difference in the hit rates, which, in turn, makes the number of accesses to the cache, local memory, and directory unchangeable. The different network service times do affect the waiting time at the system resources, and thus affect data access latency.

As can be seen in the figure, there is considerable discrepancy in the waiting times among the different service times. In general, a long network service time increases the waiting time at the network while decreasing the waiting time at the other system resources. When the network service time is zero, the system bottleneck appears at the directory in most of the programs. Some programs, such as FFT and Water-Nsqr, develop the bottleneck at the local memory. Although the system resources, such as the cache, the local memory, and the directory, are distributed evenly across the system in all memory architectures, the workload in CC-NUMA(ft) is biased against the distributed resources. This is the reason why CC-NUMA(ft) causes such a severe bottleneck even with a small number of accesses to the remote memory. As a matter of fact, when the network service time is zero, the local memory hit rate does not affect the performance of a memory architecture since there is no difference between the accesses to the local memory and to the remote memory. This is the reason why CC-NUMA(rr) has superior performance even over DYMA.

Figure 5.15 also shows the change of location of the system bottleneck. Waiting time at the network is strongly sensitive to network service time because the network is the unique centralized resource shared by the system nodes. Thus, when the network service time rises to 40/20 cycles, the network has the longest waiting time among the system resources as shown in the figure, i.e., the system bottleneck is now observed at the network. This observation suggests that the network is the critical system resource that limits the

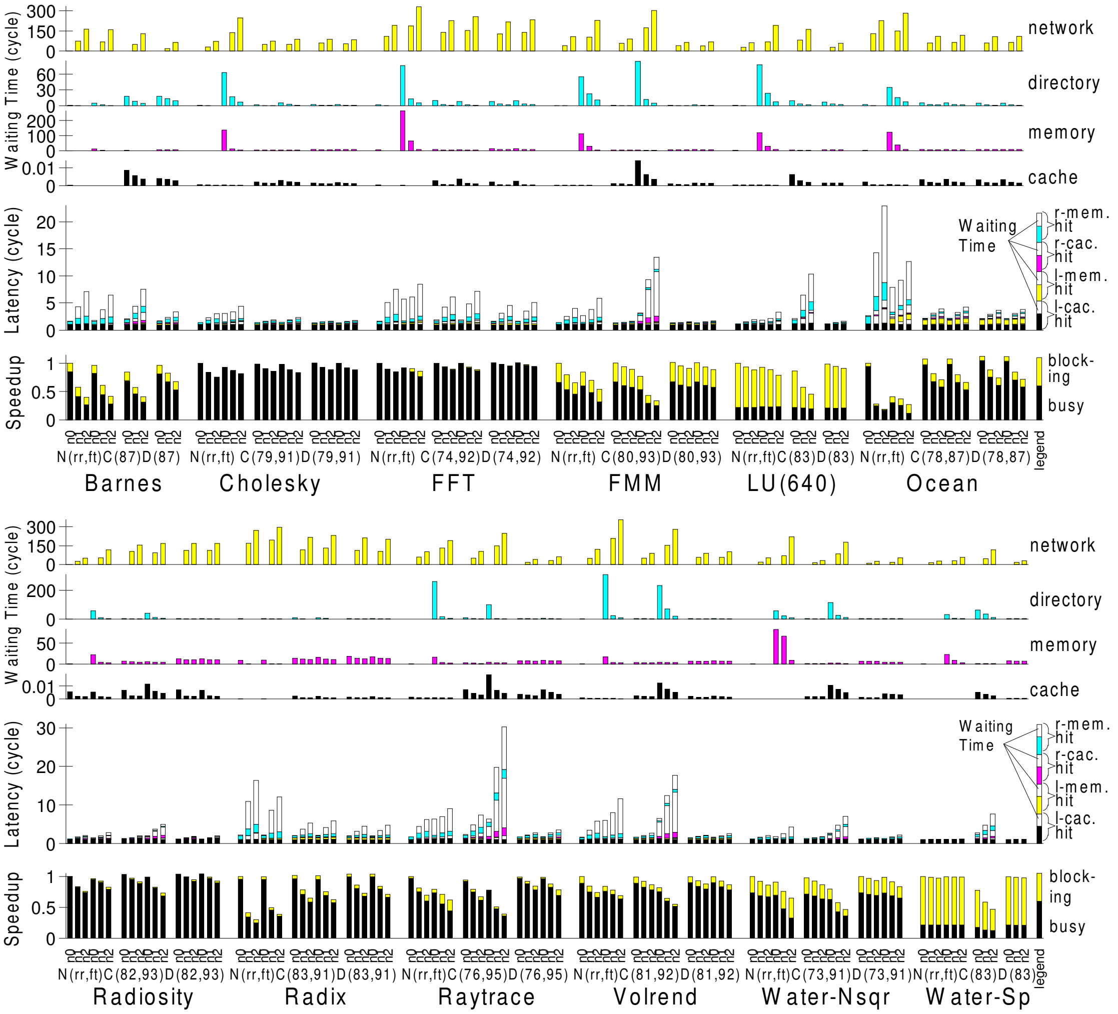


Figure 5.15: Simulation results with different network service times

scalability of the nodes, however well the network requests are distributed across the buses in the network. Since with high memory pressure both CC-NUMA and COMA generate a large number of network requests, they may suffer from the low scalability of nodes. DYMA is less sensitive to network service time than the others.

Figure 5.16 shows the simulation results when the number of buses varies from 4 to 16. The other parameter values are the same as the default values. In the figure, *b04, b08, and b16* respectively denote 4, 8, and 16 buses. Again, the different number of buses does not cause different arrangements of data. Thus, there are almost no differences in the node hit rates, in the number of accesses to the cache and to the local memory, and in the amount of network traffic. The difference among the architectures can be found in

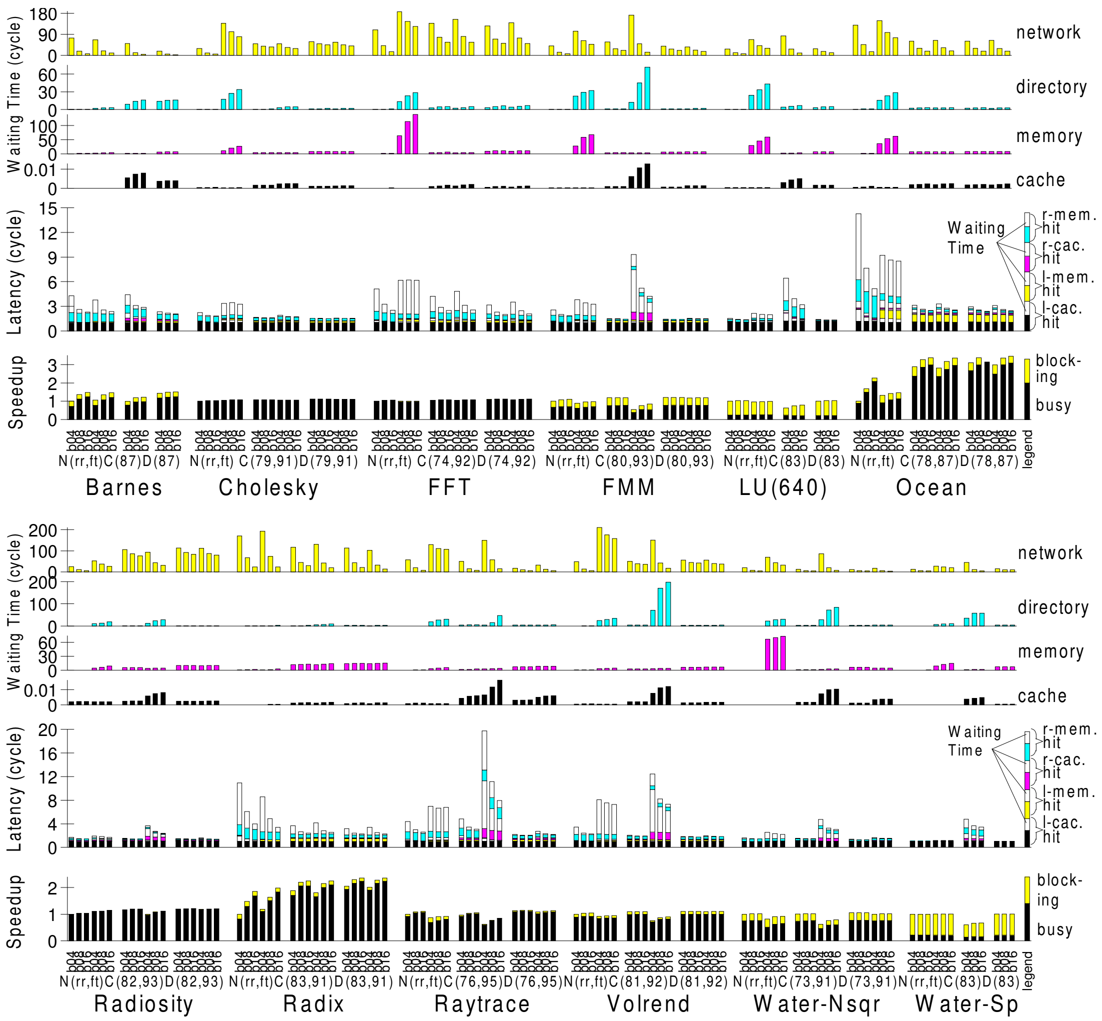


Figure 5.16: Simulation results with different number of buses

the waiting times at the system resources. Since CC-NUMA(rr) has the biggest number of network requests and distributes the requests evenly across the buses, CC-NUMA(rr) can diminish the average data access latency with a high ratio. The reduction ratio in CC-NUMA(ft) is small compared to the ratio in CC-NUMA(rr) because most network requests are consistently centralized on one bus in CC-NUMA(ft). This explains why the data access latency in CC-NUMA(ft) remains unchanged in some programs such as Cholesky and FFT. When network contention is reduced by utilizing a large number of buses, the variation in data access latency becomes small, and hence the possibility of having a large variation in execution time among nodes is low. This is the reason why the maximum processor stall time under 16 buses is smaller than under 4 or 8 buses. Also, the performance gap among

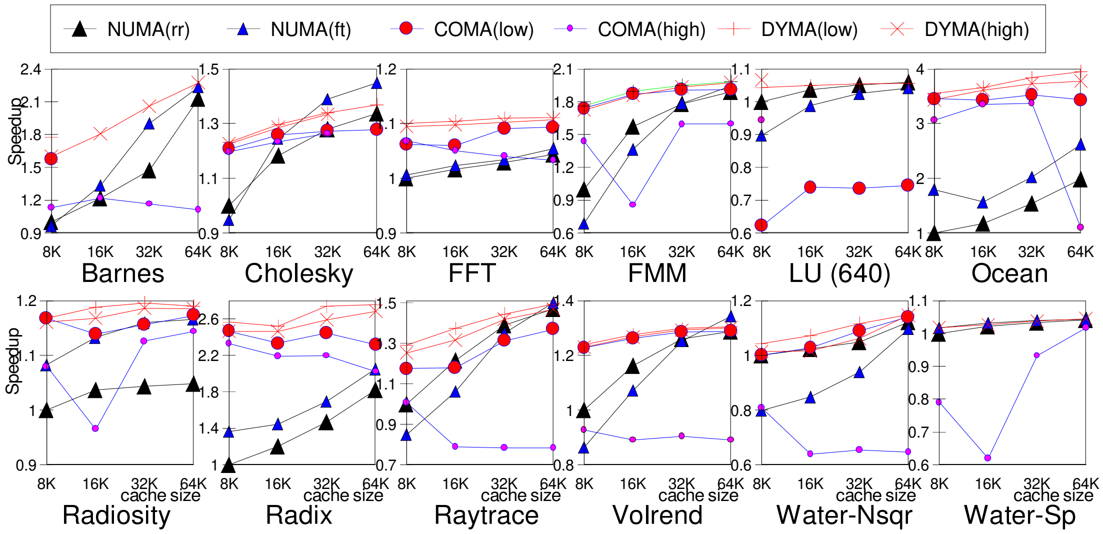
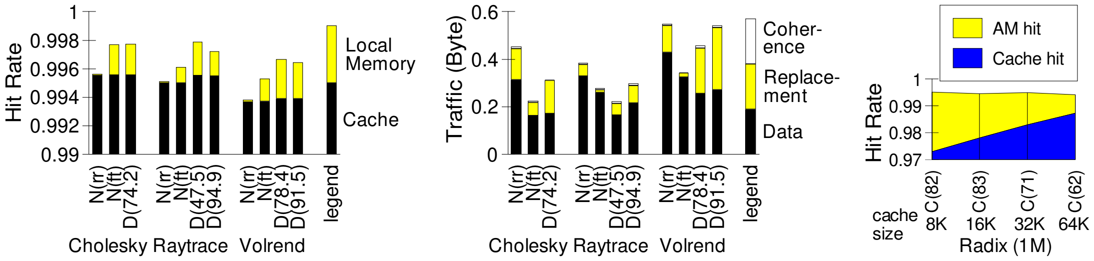


Figure 5.17: Speedup varying the cache size from 8K bytes to 64K bytes (32 nodes)

the architectures narrows as the number of buses increases.

5.4.7 Cache sizes

Figure 5.17 shows the speedup of the architectures with 32 nodes and 4 buses based on the execution time of CC-NUMA(rr) with 8K byte cache. The size of the cache increases from 8K bytes to 64K bytes. Usually, CC-NUMA(ft) has the fastest increment rate. This is because in CC-NUMA(ft) the hot spot at the master node occurs frequently when the cache size is small. However, if the cache size is large, the hot spot disappears, and the page placement in CC-NUMA(ft) increases the node hit rate. In fact, the cut-off point between CC-NUMA(ft) and DYMA can be found in Cholesky, Raytrace, and Volrend. In those programs, the slight increment in the node hit rate does not compensate DYMA for the high replacement overhead (see Figure 5.18 (a) and (b)). When a page is first loaded, the page is saved in the local memory under our page placement model for CC-NUMA without generating any replacement traffic. However, since the local memory has the same size in all system nodes in DYMA, all existing DM lines are replaced when a new page is loaded into the DM. Thus, our simulation model inherently gives higher replacement overhead to DYMA than to CC-NUMA. Our simulation results prove that the performance gap between DYMA and CC-NUMA generally decreases as the size of cache increases. On the other hand, the gap widens as the size of the working set increases. Unlike DYMA, the performance of COMA is volatile especially at high memory pressure. Further, the



(a) node hit rates (64K cache) (b) network traffic (64K cache) (c) node hit rates

Figure 5.18: Tradeoffs among the DSM architectures

performance of COMA is declining as the cache size increases in some applications. The reason for this is shown in Figure 5.18 (c). Fewer cache misses makes the decrement rate of AM hits in Radix faster than the increment rate of cache hits. In LU, the performance of both COMA and DYMA with high memory pressure is better than that with low memory pressure due to higher node hit rates. This is shown in Figure 5.17 when the cache size is 8K bytes. The reason is that under high memory pressure the relocation algorithm allows effective data migration in LU. The different cache sizes of COMA and DYMA with high memory pressure are not simulated in LU due to restriction in the sizes of the local memory in our simulation.

5.4.8 Replacement algorithms in the DM

To see the influence of the replacement algorithm in the DM on the performance of DYMA, we applied two additional replacement algorithms to the DM of DYMA. The results are shown in Figure 5.19. The random selection algorithm (*Random* in the figure) selects a copy for replacement randomly. The other selection algorithm (*Priority-B* in the figure) does not select a DM copy if all of the copies in the DM set have ownership and the copy from the cache does not have ownership. In this case, the algorithm sends the directory a replacement-hint request to update directory information for the non-ownership copy. Therefore, this algorithm can reduce relocation requests while sacrificing some node hits. The selection algorithm used for the previous simulations is represented by *Priority-A* in the figure. As can be seen in the figure, the random selection algorithm improves node hit rates and reduces data traffic rates in some programs such as Barnes and Raytrace. However, it increases the replacement overhead excessively. Thus, in most of the programs the random selection algorithm has the lowest speedup. There is almost no difference between the Priority-A and Priority-B schemes. Since only memory references to program data are used in the simulations, the results might be different if we took memory references to program

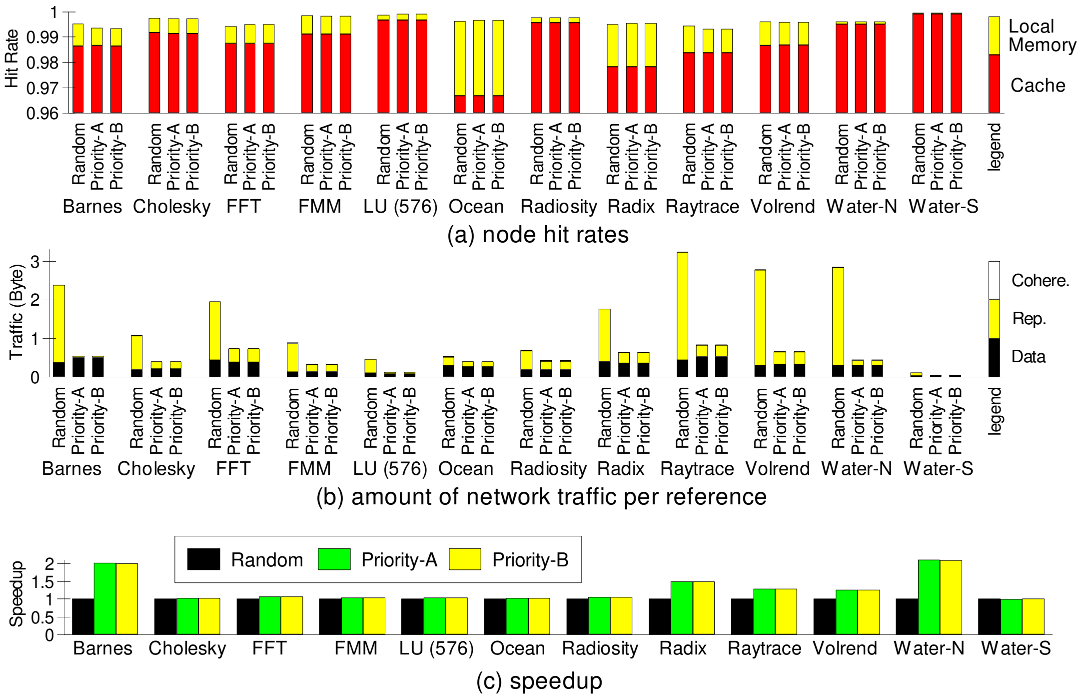


Figure 5.19: Performance of DYMA varying the replacement algorithms at the DM

instructions into account.

5.4.9 Address interleaving on directories

This section shows how different address interleaving schemes on the distributed directories affect the system performance. Even though different address interleaving schemes will require different directory sizes, and hence, different directory processing times, we assume directory processing times are the same for all of the schemes. The effect of the directory processing time on overall performance is studied in the next section.

Also, instead of studying all possible address interleaving schemes, we look at only two schemes that place the node field, I_n, at the two extreme positions in the address. The other schemes which place the I_n between these two extremes should have results somewhere between those of these two schemes. Our simulation results actually show that there is very little difference in the performance of the two schemes we studied.

The first scheme places the I_n field outside of the I_s field. This scheme interleaves 2^{I_s} consecutive line addresses as the unit of the interleaving. Since we use this scheme in CC-NUMA as the round-robin (rr) scheme, let us call this the *RR* scheme. The second scheme places the I_n field inside the I_s field. It corresponds to interleaving a line address as the

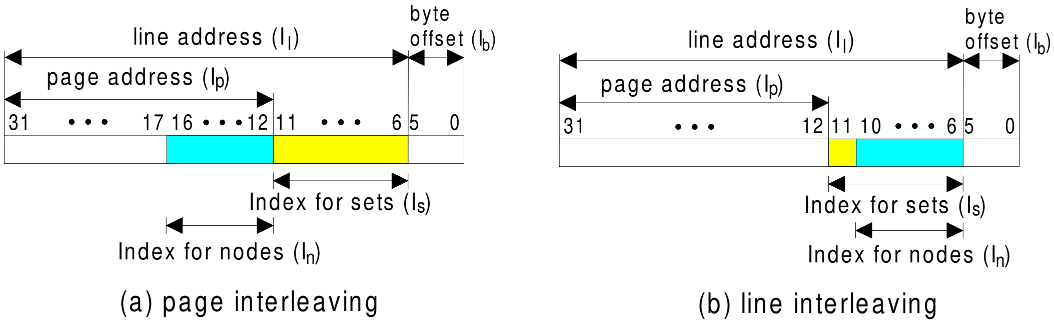


Figure 5.20: Two address interleaving schemes in CC-NUMA

unit of the interleaving, and will be called *LI* for line interleaving. The page size is assumed to be 4K (or 2^{12}) bytes and the line size is 64 (or 2^6) bytes. Also, the width of I_s is 6 bits because the size of the cache is assumed to be 16K (or 2^{14}) bytes and the set associativity is four. The number of cache lines in a cache memory, c, is $\frac{2^{14} \text{ byte cache size}}{2^6 \text{ byte line size}} = 256$. Thus, in CC-NUMA, the home node (I_n) of an address is determined from bit 12 to bit 16 in the *RR* scheme, i.e., page interleaving (see Figure 5.20 (a)), and from bit 6 to bit 10 in the *LI* scheme (see Figure 5.20 (b)). This will result in different numbers of pointers needed in the pointer pool: $c = 256$ for the *LI* schemes and $cn = 256 \times 32 = 8192$ for the *RR* scheme, assuming 32 nodes.

In the simulation for COMA and DYMA, the number of AM and DM sets is assumed to be a multiple of the number of cache sets, and it varies according to memory pressure given the program problem size. Thus, the home node field (I_n) in the *RR* scheme is changed depending on the variable AM and DM sizes. Unlike CC-NUMA(rr), COMA(rr) and DYMA(rr) put several consecutive pages in the same directory since the number of sets in the local memory is larger than the number of lines in a page. Also, in the *RR* scheme, there is no way of finding a node which has space for the relocation request at the directory. Thus, if there is no shared node which has the same replaced copy, the directory randomly selects a node for the relocation request. This might increase the number of network requests. However, in the *LI* scheme which we have used so far, the home node field (I_n) is taken from bit 6 to bit 10 as in CC-NUMA. Also, the directory in the *LI* scheme can select the node which has space for the relocation request. The simulation results of CC-NUMA(ft) are also included in the following figures.

Figure 5.21 and 5.22 show the results of the simulation. Since the address mapping to the cache set is the same in all schemes, there should be no difference in the cache hit rates. However, the side-effects caused by replacements in COMA(rr) with high memory pressure make the cache hit rate different, especially in the programs (FMM, LU, Raytrace, Volrend,

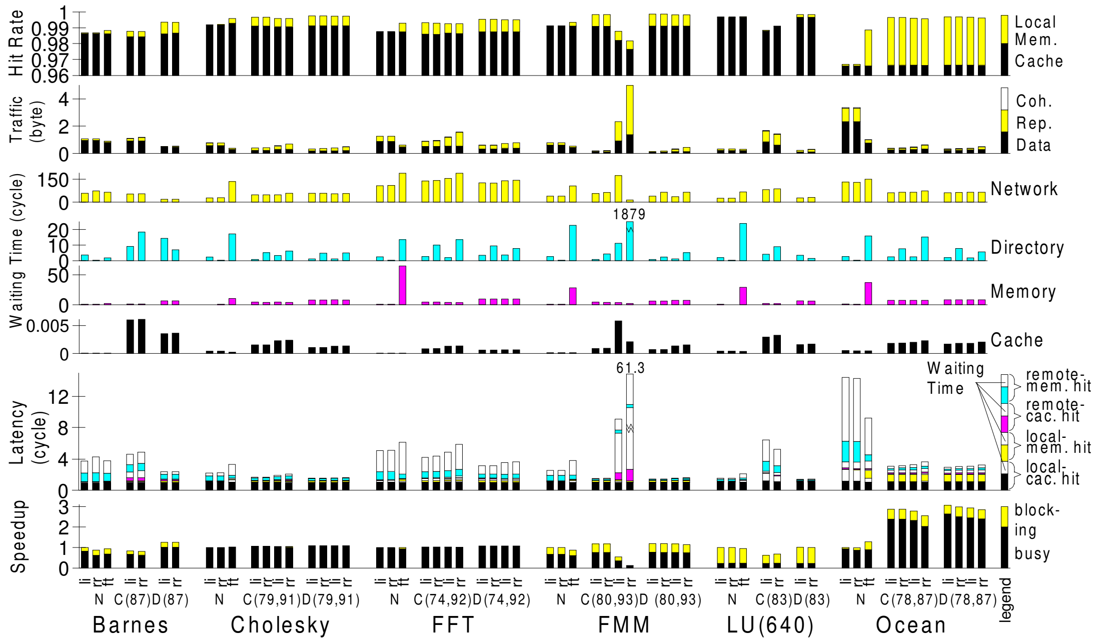


Figure 5.21: Simulation results with different address interleaving schemes

and Water-Sp) which actively share some data. Although the way of allocating data in the local memories is different between CC-NUMA(rr) and CC-NUMA(li), it does not make a difference in local memory hit rates. This is because both CC-NUMAs allocate data in the local memories without considering reference locality. In fact, we found that the possibility of finding the line requested by a cache miss is uniformly distributed among all memory modules.

We can observe that the amount of replacement traffic in COMA and DYMA with the *RR* scheme is larger than with the *LI* scheme. This is because the random selection algorithm in the *RR* scheme cannot find the exact node which can keep the relocation request, and hence a relocation request may be forwarded several times until it finds available space. In CC-NUMA, the relocation request is not forwarded since the home node can keep the replaced copy. Thus, there is no difference in the amount of replacement traffic between CC-NUMA(rr) and CC-NUMA(li). Since the relocation request is not on the critical path of a data access, the different amounts of replacement traffic only affect the system performance indirectly.

Despite the similar number of requests, the *LI* scheme usually has longer average waiting times at the directory and at the local memory of CC-NUMA than the *RR* scheme. This is because requests for a missing line and a replaced line are sent to the same home node

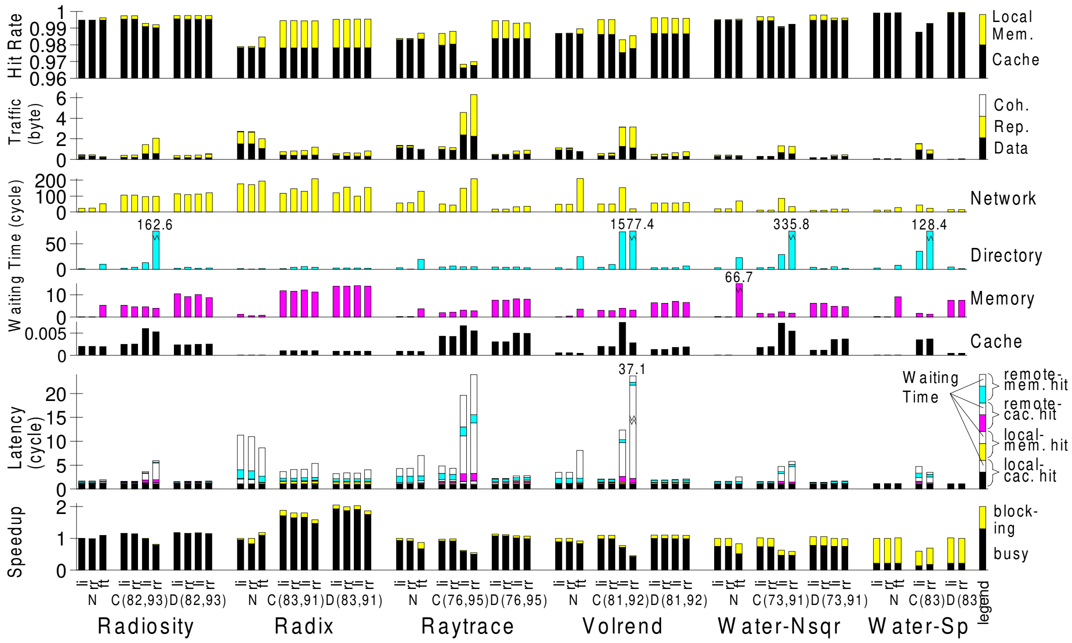


Figure 5.22: Simulation results with different address interleaving schemes (cont.)

in the *LI* scheme. The replacement request which follows the missing data request usually needs to wait for the completion of the missing data request at the same node. The home directories of a missing line and a replaced cache line in the *RR* scheme are different in most cases. Hence, conflicts at the directory and at the local memory happen less frequently. However, since the relocation request is not on the critical path of a data access, the longer waiting time does not affect overall system performance. In COMA and DYMA, since the *RR* scheme creates a large number of relocation requests, the *LI* scheme does not always have longer waiting times at the directory and at the local memory than the *RR* scheme. Further, COMA(rr) with high memory pressure suffers from severe waiting time at the directory in FMM, Radiosity, Volrend, and Water-Nsqr.

Even in CC-NUMA with the *RR* scheme, we can find the unbalanced workload. If all the lines in a page are allocated at the same node, there is the possibility that the requests for the lines will be concentrated on the local memory of the node during a short period of time. Since the interleaving scheme distributes the lines across the nodes, it might give a better chance of balancing the workloads. Also, the replacement request which follows the missing data request might cause less conflict with other missing data requests. For these reasons, the waiting time portion of the data access latency of CC-NUMA(li) is slightly less than that of CC-NUMA(rr) in many programs. In COMA and DYMA, overall data access

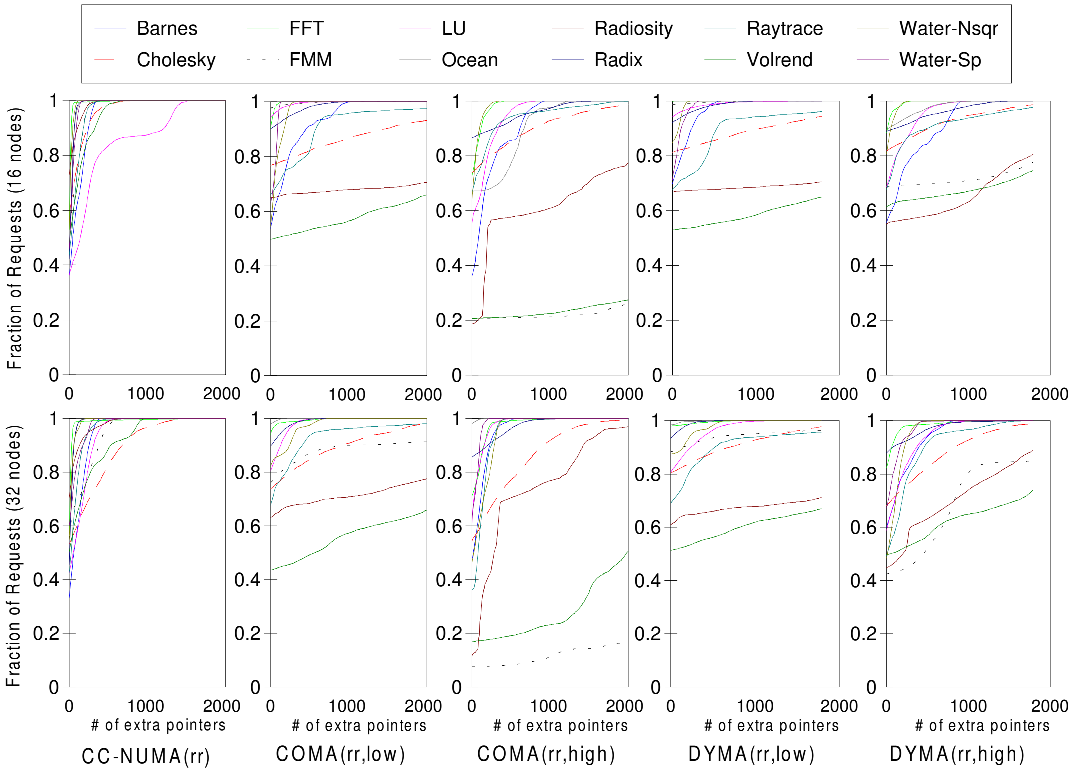


Figure 5.23: Number of pointers in use seen by data requests to the directory

latency with the RR scheme is longer than that with the LI scheme due to the different waiting times.

The speedup in Figure 5.21 and 5.22 is based on CC-NUMA(li). In CC-NUMA, no one scheme provides superior performance stability over the other schemes. However, in COMA and DYMA the RR scheme causes a big performance degradation in many programs under high memory pressure, especially when the networking overhead is high.

In the RR scheme, the maximum number of pointers needed in each pointer pool should be cn in CC-NUMA, mn in COMA, and $(m + c)n$ in DYMA, where c is the number of lines in a cache, m is the number of lines in a local memory, and n is the number of nodes. Given our default simulation parameters, c is 256, n is 32, and m is changed according to the programs (see Table 5.1). However, the number of the pointers used in the pointer pool will vary over time during program execution. The maximum number of pointers used in the pointer pool cannot reveal the actual usage of those pointers at program execution time.

In Figure 5.23, we show the number of pointers which have already been allocated when a data request arrives at the directory in the *RR* scheme. Comparing this with the *LI* scheme which needs a maximum of c pointers in CC-NUMA, m in COMA, and $m+c$ in DYMA, the x-axis in the figure starts with c $(= 256)$ in CC-NUMA, m in COMA, and $m+c$ in DYMA. The y-axis represents the fraction of the requests which see that the number of pointers additionally in use is less than x. The first and second rows of graphs in Figure 5.23 assume the number of nodes is 16 and 32 respectively. From the figure, the number of pointers actually used is much less than the maximum numbers. Also, the number of pointers used in a program, such as LU, does not always increase as the system size (number of nodes) grows. The number of pointers in use depends very much on the program's behavior. However, in general, large system size and high memory pressure increase the number of pointers. COMA(rr) with high memory pressure demands an extremely large number of pointers in FMM, Radiosity, and Volrend, which are the three programs with largest data sizes and which actively share some data. In the figure with 32 nodes, even though the maximum number of pointers which can be used by a program in CC-NUMA is $256 \times 32 = 8182$, all programs use fewer pointers than $256 \times 7 = 1792$, a number which is still much larger than the 256 needed in the *LI* scheme.

5.4.10 Directory processing times

This section shows the effects of the pointer-based directory on performance as compared with the full-map directory. All of the directories have the same directory address interleaving, i.e., interleaved data with one line as the unit of the interleaving (line interleaving). However, we consider three different choices for the directory structure: full-map directory; pointer-based directory using a head link for each line in the local memory to point to the associated linked list; pointer-based directory using the hashing of the cache line address to locate the associated linked list, designated as *ful*, *pt-a*, and *pt-b*, respectively. In our simulation, only the number of accesses to the directory is considered for the directory processing time. For the *ful* directory, we assume that every request can be served within one DRAM access time (8 cycles). That is, the bit vector saved in DRAM can be fetched and updated in 8 cycles. For the *pt-a* directory, the number of DRAM accesses per request is assumed to be two (16 cycles). In the *pt-b* directory, 5 DRAM accesses (40 cycles) is used as the processing time. Another justification for the long directory processing time is that the protocol which manages the pointer pool is implemented with software with little hardware support.

Figure 5.24 shows the simulation results when the directory processing time varies. All three directories have the same address mapping to the cache and to the local memory, hence, there is no difference in the cache and local memory hit rates. Consequently, the

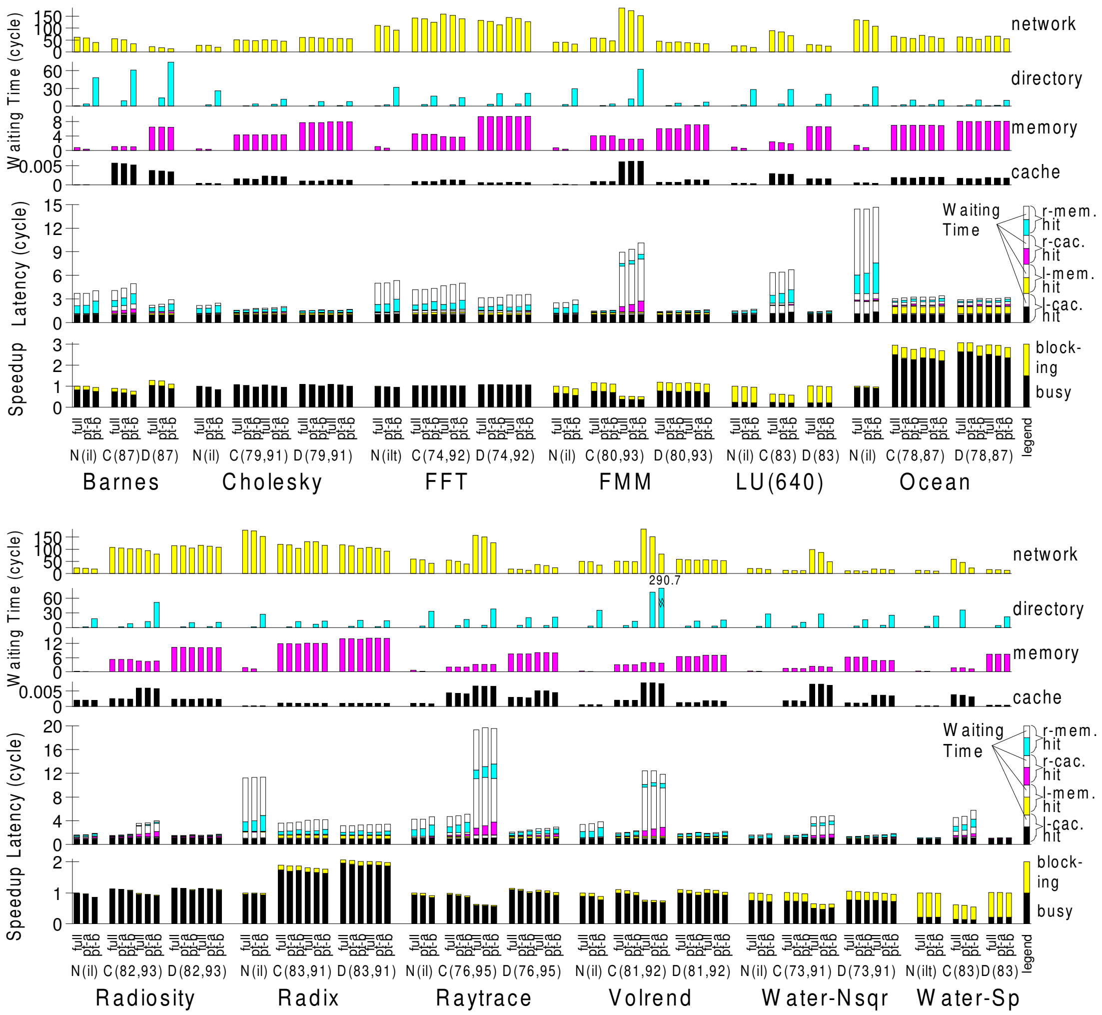


Figure 5.24: Simulation results with different directory processing times

different directory structures do not affect the number of requests to the cache, to the local memory, to the directory, or to the network. We do find a difference in waiting times at the system resources. Although we observe very little difference in the amount of network traffic among the three directory schemes, waiting times at the network and at the directory vary quite noticeably, which in turn affects the average data access latency and the speedup as shown in the figure. The speedup in the figure shows the program execution times normalized to the case of CC-NUMA with the *ful* directory. In some cases, a small directory processing time causes slower execution time than the others. This is because the processor stall time varies based on the unpredictable synchronization overhead under the different directory processing times. The results show that performance is sensitive to

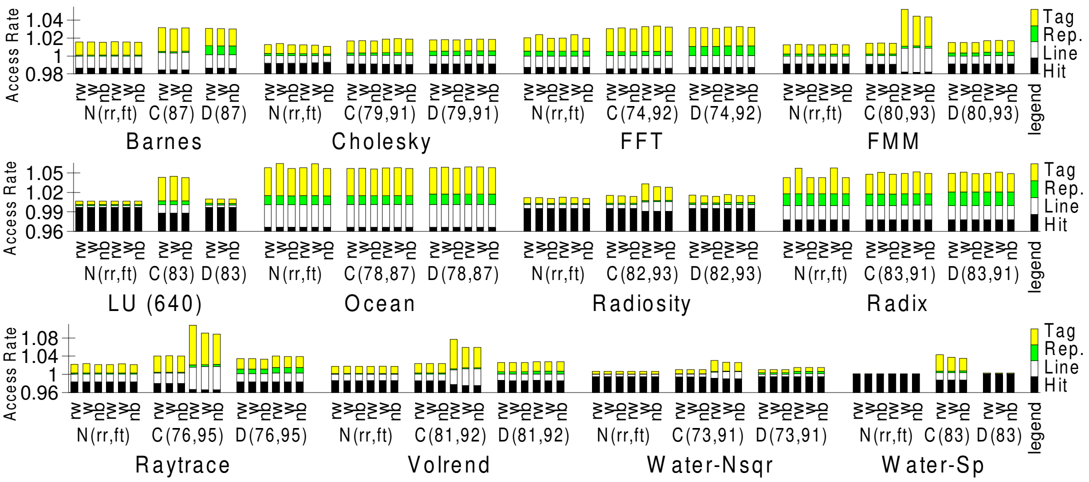


Figure 5.25: Number of accesses to the cache per reference with different protocols

directory processing time. Hence, the size of the pointer pool (a smaller pointer pool will have a faster access time) and management of the linked list should be as simple as possible. Directories similar to *pt-a* seem to be a good choice.

5.4.11 Directory protocols

The write-back from the cache to its local memory in COMA seems to be unprofitable since the write-back request cannot reduce the number of network transactions required by the next request for the line. Furthermore, since copying a line from the cache is faster than from the local memory, keeping a valid copy in memory is useless from the latency view of a remote access. Thus, the write-back scheme in COMA is not simulated. The write-back scheme in CC-NUMA can reduce the data access latency if the copy written back to the memory is continuously valid when the next request for the same cache line arrives at the directory. However, if the copy is invalidated by a sharing node, the write-back request can waste network and memory bandwidth. Although the write-back scheme may cause different consequences in performance, we did not simulate it because previous studies show that the performance advantage of the write-back schemes is marginal [AB86].

There is almost no difference in node hit rates among the protocols since the protocols hardly affect locality of the data. Differences might come from the overhead of arranging data in the memory locations. An efficient protocol should move data to the desired locations with a minimal number of requests and with a short data access latency. Thus, different protocols may create different numbers of accesses to the cache and to the local memory, different requests to the directory, and different amounts of network traffic yet still

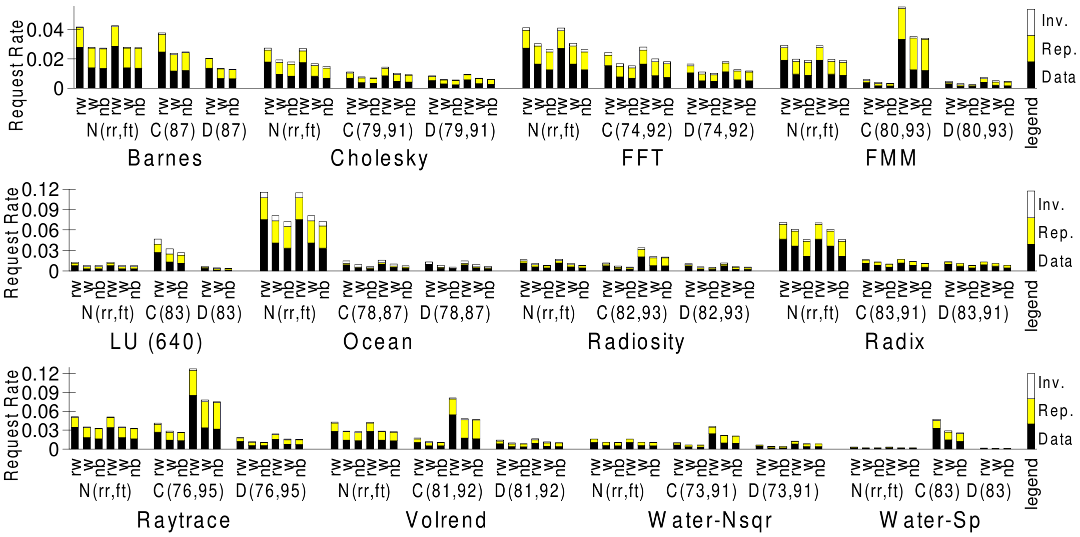


Figure 5.26: Number of requests to the directory per reference

have the similar node hit rates.

The average number of accesses to the cache per reference is shown in Figure 5.25. In the figure, *B-rw* (blocking for reads and writes) denotes the DVSM protocol without buffers at the directory, *B-w* denotes the write-only blocking protocol, and *NB* stands for the non-blocking protocol (see Section 4.3.3). There is no significant difference among the protocols except in one case. In some programs the tag access rate of the *B-w* protocol is slightly higher than those of the others. One possible reason for this is that the read requests forwarded to the owner nodes from the directory are more likely to be returned to their requesters as unsuccessful requests by invalidation requests. Two tag accesses to the local and remote caches are required for each unsuccessful request. Since the *B-rw* protocol locks the directory for every type of request, the read requests are returned to their requesters without the tag accesses to the remote caches. Since the *NB* protocol has a higher possibility that the read requests are successfully returned from the owner node, the tag access rate is smaller than that of the *B-w* protocol. Also, in many cases, the *NB* protocol has the smallest tag access rate although the flush signal to avoid deadlock in the *NB* protocol must be counted in the tag access rate. This means that the *NB* protocol generates very few flush signals.

We can see a clear difference among the protocols in request rates at the directory. Figure 5.26 illustrates the difference. Since the *B-rw* protocol locks the directory entry for both read and read-exclusive requests, it needs the same number of requests to unlock the entry. Also, if subsequent data requests are blocked at the directory, the blocked requests

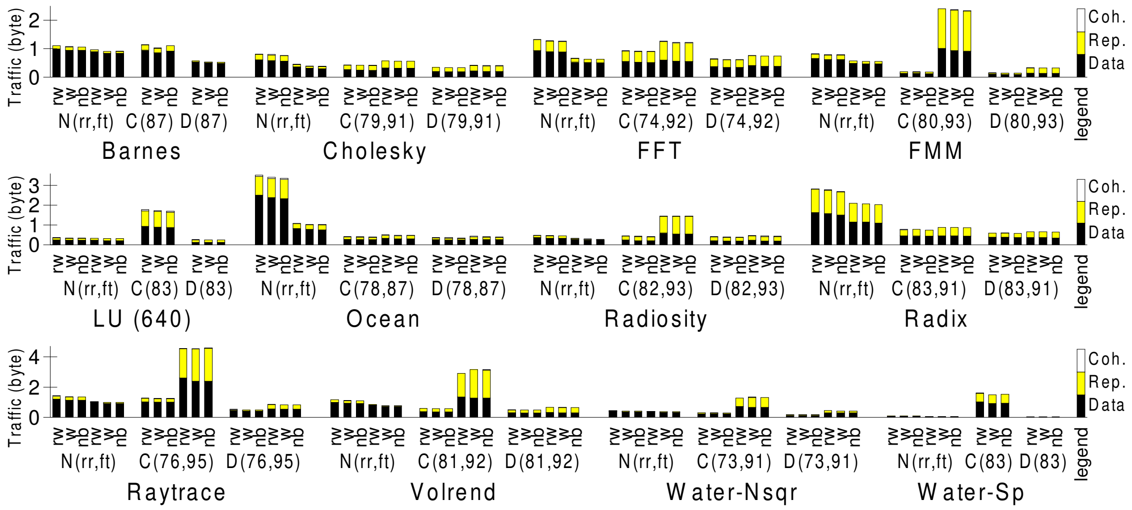


Figure 5.27: Amount of network traffic per reference

need to be reissued before the requests can finally be forwarded to the owner nodes. In the *NB* protocol, no requests are blocked at the directory, and hence no extra requests to unlock the directory entries are needed. Therefore, the data request rate to the directory in the *NB* protocol is usually less than 50% of the rate in the *B-rw* protocol. Since the *B-w* protocol locks the directory for write accesses, the data request rate of the *B-w* protocol is between those of the *B-rw* and *NB* protocols. Replacement and invalidation request rates are almost the same among the protocols because in most cases the replacement and invalidation requests are handled in the same way.

The average amount (bytes) of network traffic per reference for each directory protocol can be found in Figure 5.27. The amount of network traffic varies across the protocols even though the difference is small compared to the directory request rates. The main reason for the small difference is that a large number of the requests that make the difference in the directory request rates do not contain data. In most of the programs the *NB* protocol has the smallest amount of data traffic among the three protocols. However, for replacement traffic, it is hard to say which protocol is better than the other. In fact, the effects of the flush signals in the *NB* protocol are so marginal as to be ignored since they are generated very infrequently. The *NB* protocol has the smallest amount of coherence traffic even though the difference is negligible. Again, the primary reason for this is the method for handling simultaneous requests to shared data.

The average waiting time per request at each system resource is exhibited in Figure 5.28. Generally, there are two possible cases which could make the waiting time long,

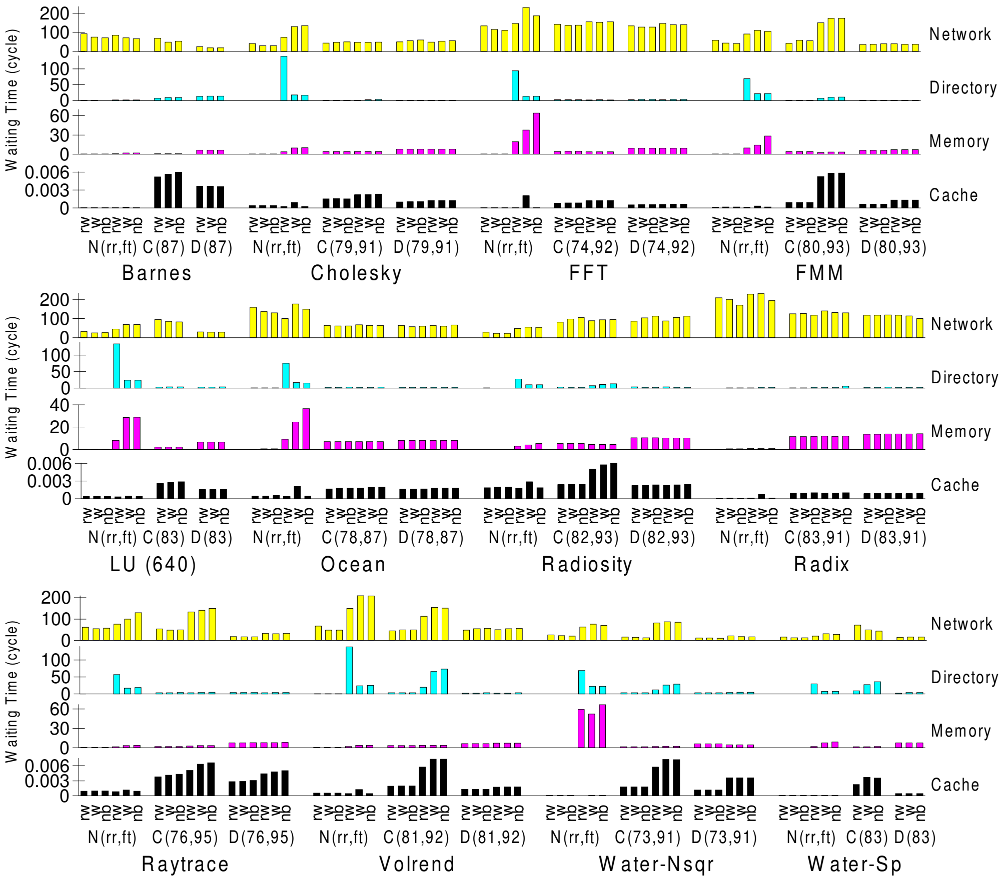


Figure 5.28: Waiting time per request at system resources

given the same number of requests to a distributed resource. The first case happens when requests are centralized to one part of the resource. The second case occurs when requests are concentrated during a short period of time. Thus, a large number of requests to the system resources need not cause long waiting times if the requests are evenly distributed over both space and time. We can find some examples of this by looking at waiting times at the system resources: cache, local memory, directory, and network. Even if there is a small difference in the rate of accesses to a resource, the difference in waiting times at a resource can be large when the resource is overloaded. CC-NUMA(ft) usually has longer waiting times than CC-NUMA(rr) due to its unbalanced workload. This explains why the variation among the protocols in CC-NUMA(ft) is big, especially at the directory. Although the *NB* protocol reduces the amount of network traffic, the *NB* protocol frequently has longer waiting times than the *B-rw* protocol, especially at the network in most of the architectures. This is because in the *NB* protocol requests are likely to be concentrated on a given resource

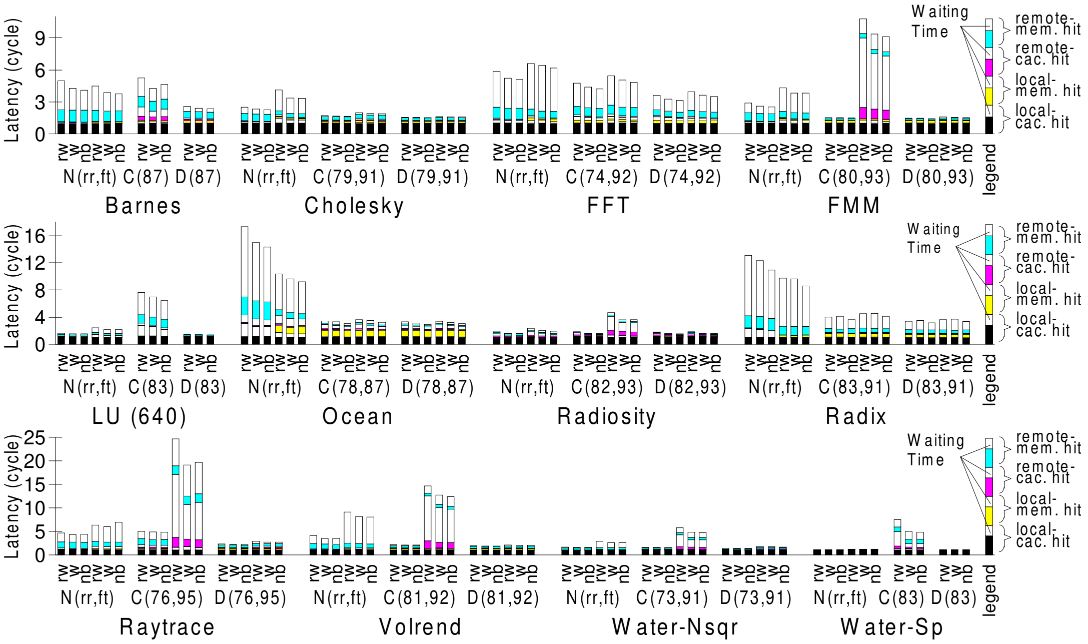


Figure 5.29: Average data access latency including the waiting time

during a short period of time. However, this does not always mean worse performance since the long waiting times are caused by the short program execution time.

Figure 5.29 presents the average data access latency per reference. If we exclude waiting time, there is almost no difference in the average latency among the protocols. This is because the critical path of a data access is the same among the protocols, and simultaneous accesses to the same cache line happen infrequently. The difference among the protocols comes mainly from the waiting time at system resources, especially for remote accesses. Extra requests which are not on the critical path in the blocking protocols increase waiting time. In the figure, the *NB* protocol tends to have a small average data access latency in most of the programs.

Figure 5.30 depicts the speedup based on CC-NUMA(rr) with the *B-rw* protocol. The *NB* protocol provides superior performance stability over the other protocols except CC-NUMA(ft) with Cholesky. Although CC-NUMA(ft) with the *NB* protocol in Cholesky has shorter average latency than the other protocols (see Figure 5.29), it has the lowest speedup among them. The reason for this is the differing processor stall times at synchronization points among the protocols. A processor which arrives at the synchronization point stalls until other processors release the stall condition. Thus, although the average latency per reference is small, the execution time of a program can be long if the execution time of one

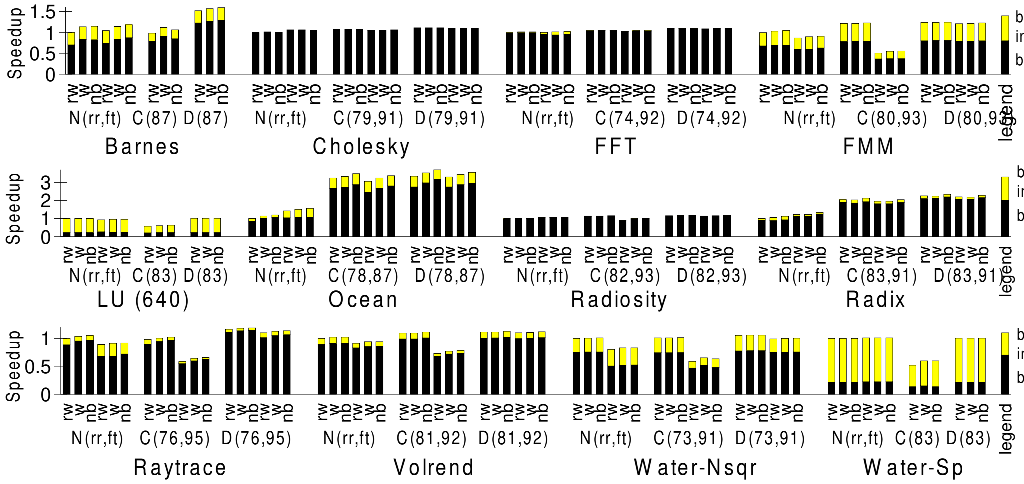


Figure 5.30: Speedup: program execution times normalized to the case of B-rw (rr)

processor is especially long. In fact, on Cholesky the accumulated stall time of the master processor (node number 0) in CC-NUMA(ft) with the *NB* protocol is longer than the others. A reverse result can also be found in CC-NUMA(ft) with Raytrace. The speedup for *NB* with (ft) is the highest even though the average data access latency is the longest (see Figure 5.29). Performance difference among the protocols is large when the cache hit rates are low and when program parallelism is high as in Ocean and Radix. This means that the reduction of network traffic rate in the *NB* scheme is more effective on performance when network contention is high.

If network service time grows, the time that the directory is being locked is increased. Thus, there is a high possibility of blocking requests at the *B-rw* directory. In fact, our simulations with a very long network service time (110/90 cycles) show that simultaneous requests are made frequently in Radiosity. However, the default network service time (30/10 cycles) does not make such frequent simultaneous requests. In fact, the performance gap among the protocols widens as network service time and directory processing time increase.

5.4.12 Two level caches

We can apply the protocol of DYMA to the two-level caches while managing the local memory as in CC-NUMA. Let us call this new two-level cache architecture exclusive CC-NUMA (E-NUMA) and call the traditional two-level cache architecture inclusive CC-NUMA (I-NUMA). Then, the binding to the L2 cache in E-NUMA happens at an L1 cache miss while it happens at an L2 cache miss in I-NUMA. In the two-level caches, no replaced line affects locality of data in a remote cache. The number of lines that can be shared

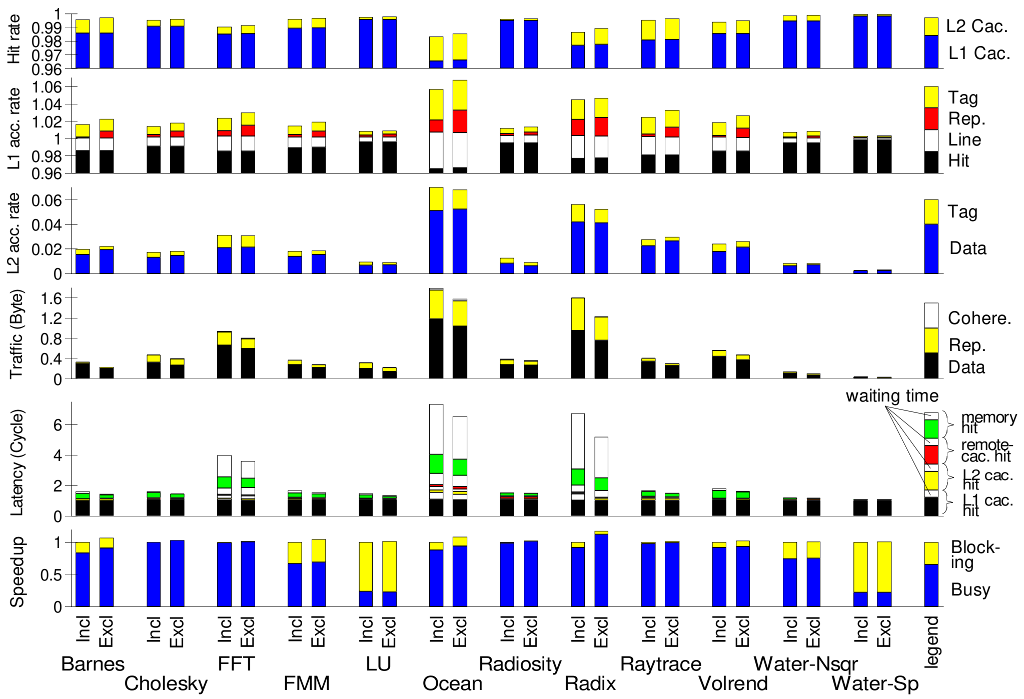


Figure 5.31: Simulation results in two level caches

by several nodes is large enough to avoid excessive competition for the shared lines. Also, the possibility of selecting an ownership copy for replacement at the cache can be low. An advantage of E-NUMA is that the L2 cache of E-NUMA can additionally possess lines which are not in the L2 cache of I-NUMA. Another advantage of E-NUMA over I-NUMA may be the small number of accesses to the L2 cache. However, the performance gap between E-NUMA and I-NUMA may not be as high as the performance gap between DYMA and COMA. In the following, we will see how much the differences affect the performance of the architectures.

Figure 5.31 shows the simulation results of the two architectures. In the figure, I-NUMA and E-NUMA are denoted by *Incl* and *Excl* respectively. Here, each architecture utilizes the round-robin fashion page allocation in the local memories. Access time to the L1 cache is the same as before. Access times for the tag and data at the L2 cache are assumed to be 2 and 4 cycles respectively. The associativities of the L1 and L2 caches are respectively 2 and 4 ways. Also, the L1 and L2 cache sizes are 16K and 64K bytes respectively. The other simulation parameters have the same default values as in Table 5.2.

The difference between the L2 cache hit rates of I-NUMA and E-NUMA is small, and there is almost no difference in the L1 cache hit rates. This suggests that the advantage of

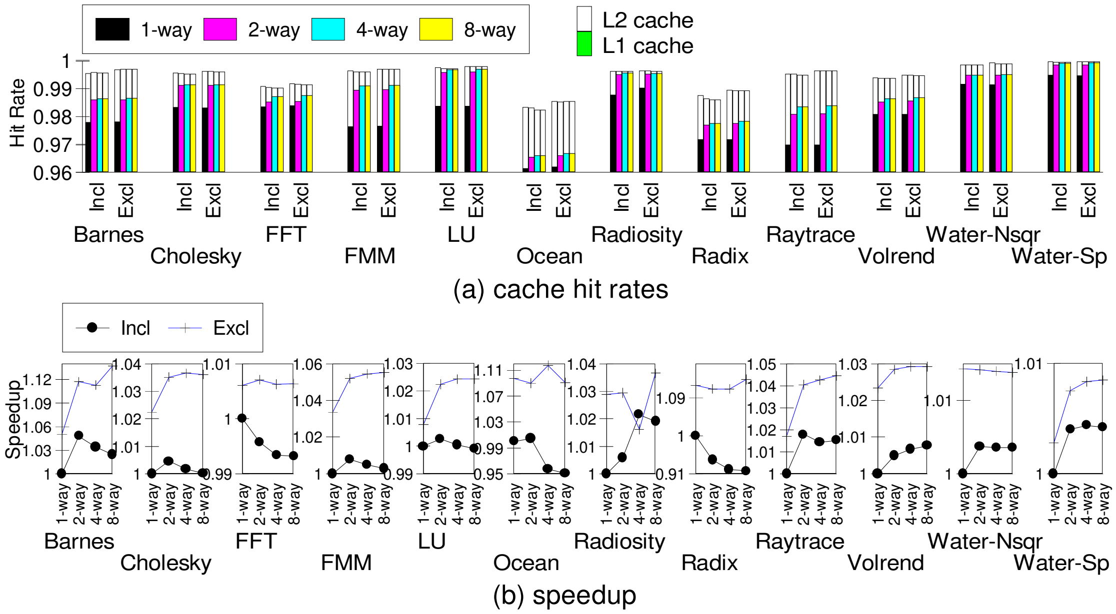


Figure 5.32: Performance sensitivity to L1 cache associativity

having additional copies in the L2 cache of E-NUMA may not result in a significant impact on performance. From this observation, we can guess that the main performance gain in DYMA over COMA does not come from the extra copies but from the small replacement overhead. Although the difference is marginal, the replacement access rates of the L1 cache in E-NUMA are higher than in I-NUMA due to the protocol for modularity between the L1 cache and the L2 cache. The high L2 hit rate in E-NUMA additionally increases the L2 data access rate. However, the lines invalidated at the L1 cache remove the L2 tag accesses enough to make the L2 (tag and data) access rates of E-NUMA lower than those of I-NUMA in most programs. The small amount of network traffic in E-NUMA makes the data access latency smaller than that of I-NUMA. Due to the small difference between the node hit rates of the two architectures, the performance gap of the two architectures is small although E-NUMA always outperforms I-NUMA.

To see the performance sensitivity to L1 cache set associativity, we simulate the two architectures while changing the L1 cache associativity from 1 way to 8 ways. The other simulation parameters remain unchanged. The simulation results are shown in Figure 5.32. The synergy effects seen in COMA are also observed in I-NUMA. The higher the L1 cache hit rate is, the lower the locality of data in the L2 cache is. This is the reason why the node hit rate is not increasing and even decreasing in some programs. This phenomenon can explain the result that, in some programs, the execution times become longer as the L1

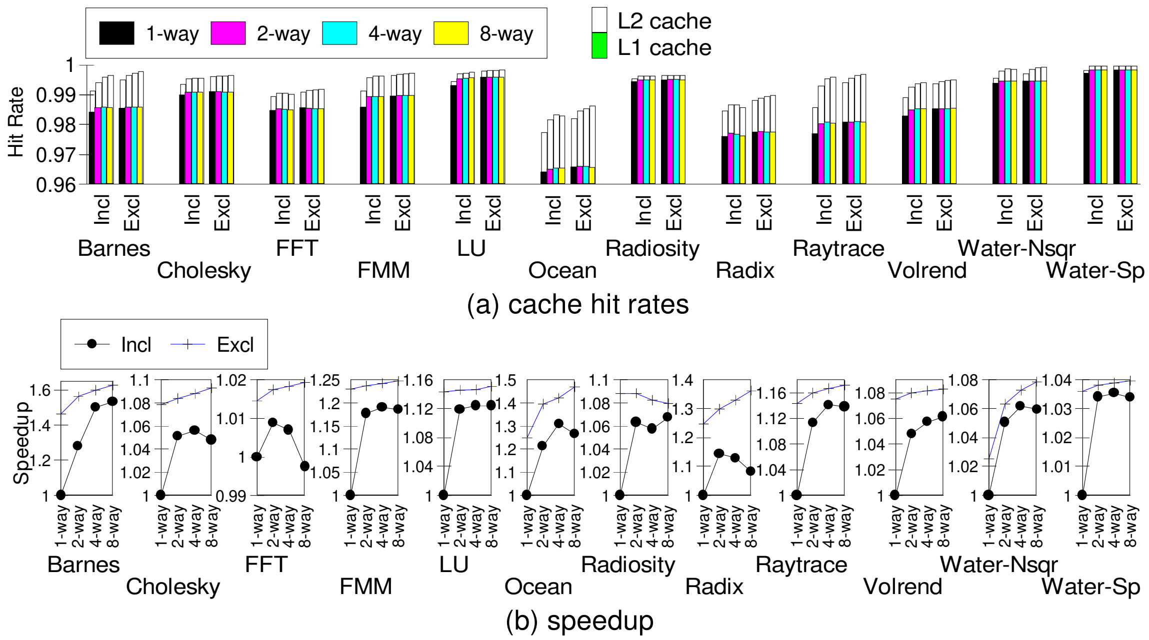


Figure 5.33: Performance sensitivity to L2 cache associativity

cache set associativity increases. However, the speedup of E-NUMA is usually increasing.

Figure 5.33 shows performance sensitivity to L2 cache set associativity. The L2 cache set varies from 1-way associativity to 8-way associativity. The L1 cache maintains 2-way set associativity. Usually, higher associativity of the L2 cache increases the node hit rate. Thus, high associativity is more desirable in the L2 cache than in the L1 cache. The speedup of I-NUMA increases faster than that of E-NUMA as the associativity becomes higher.

Next, we studied performance sensitivity to the L1 cache size. The simulation results are shown in Figure 5.34. Similar to the relationship between COMA and DYMA, the performance gap between I-NUMA and E-NUMA widens as the L1 cache size increases. This is because the redundancy of the L2 cache in I-NUMA is high when the L1 cache size is large. Also, the deduction rate of L2 cache hits is higher than the increment rate of L1 cache hits in some programs. This makes the speedup of I-NUMA even decrease in those programs. Figure 5.35 shows performance sensitivity to the L2 cache size. As can be expected, the opposite results to the previous simulation are produced. The performance gap between I-NUMA and E-NUMA narrows as the size of the L2 cache grows. This suggests that if the inclusion property is enforced the small L2 cache size should be avoided in the cache design.

In order to see the performance sensitivity to line size in the two-level caches, we simulate

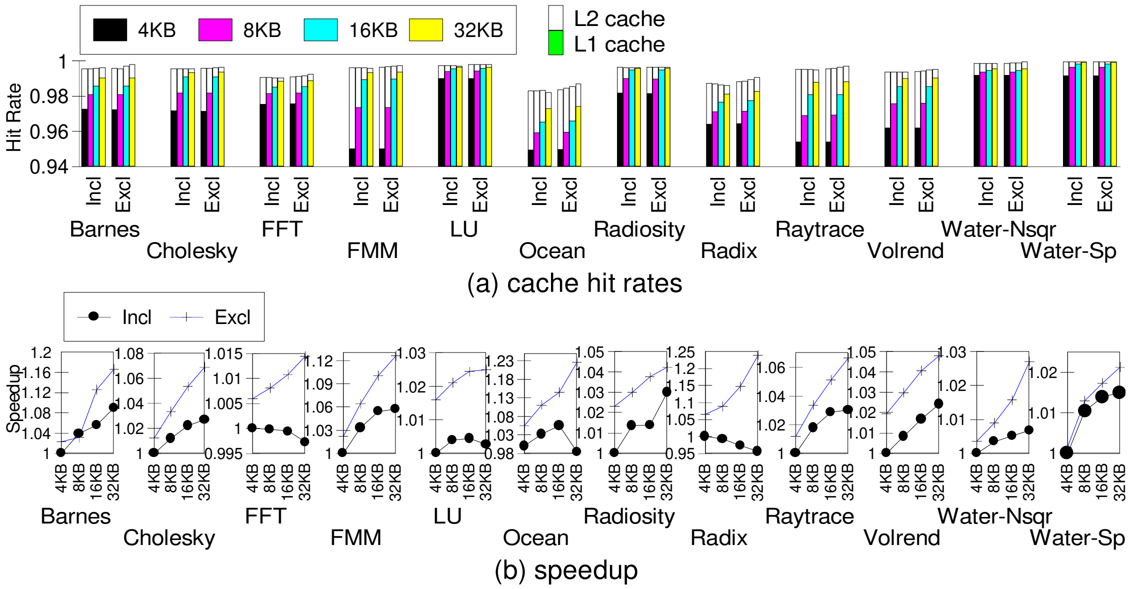


Figure 5.34: Performance sensitivity to L1 cache size

the architectures and vary the line size from 16 bytes to 128 bytes. Figure 5.36 shows the node hit rates of the architectures. As the line size becomes bigger, the node hit rates usually increase unlike the cases of COMA and DYMA. This means that, in some programs, the negative effect on the locality of data in the local memory caused by the replacement in COMA and DYMA is higher than the effect of false sharing when the line size is large. Also, this suggests that if the overhead of handling a large line is very close to the overhead of handling a small line, the large line is desirable in a system design.

To see how the L2 access time affects performance, we also simulate the two architectures while varying the L2 tag and data access times from 1 and 2 cycles to 4 and 8 cycles respectively. The simulation results are shown in Figure 5.37. Generally, the degradation of performance is linear on the access time. However, performance is not always decreasing in some programs, such as Ocean, Radix, Radiosity, and Volrend. One of the reasons for this is that the execution times of the programs are highly dependent on network communication time. Thus, the longer L2 access time has little impact on performance. Another reason is that the high synchronization overhead can produce unpredictable simulation results. Ocean and Radix belong to the first case, while Radiosity and Volrend belong to the second case.

The scalability of the two architectures is shown in Figure 5.38. The number of nodes varies from 4 to 32. The other parameters for the simulation are determined by default. Unlike in COMA and DYMA, the speedup in Ocean and Radix is not linear even though

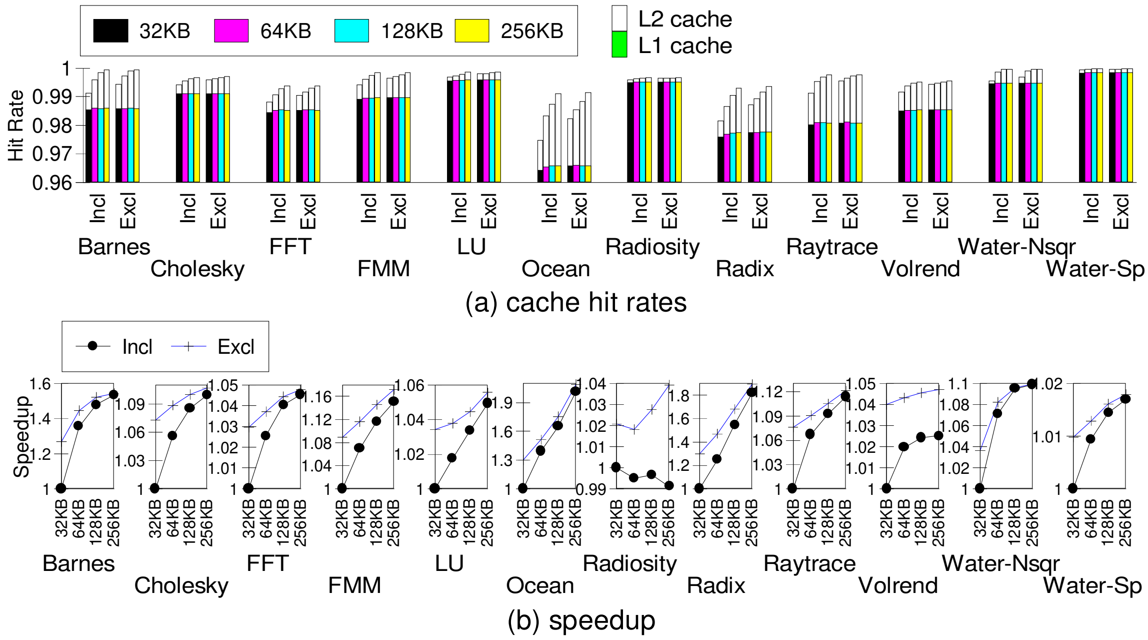


Figure 5.35: Performance sensitivity to L2 cache size

they have sufficient parallelism. The reason for this is that the size of the L2 cache is not large enough to reduce the network traffic. Thus, the network bottleneck makes their performance saturated after 16 nodes. Since Barnes and Water-Nsqr have high L1 cache hit rates, the network bottleneck does not occur even when the number of node is 32. The speedup in Barnes and Water-Nsqr is limited only by the program's parallelism. Since the other programs have very low parallelism, the speedup of the programs is very small. Similar to the relationship between COMA and DYMA, the performance gap between I-NUMA and E-NUMA widens as the number of nodes increases.

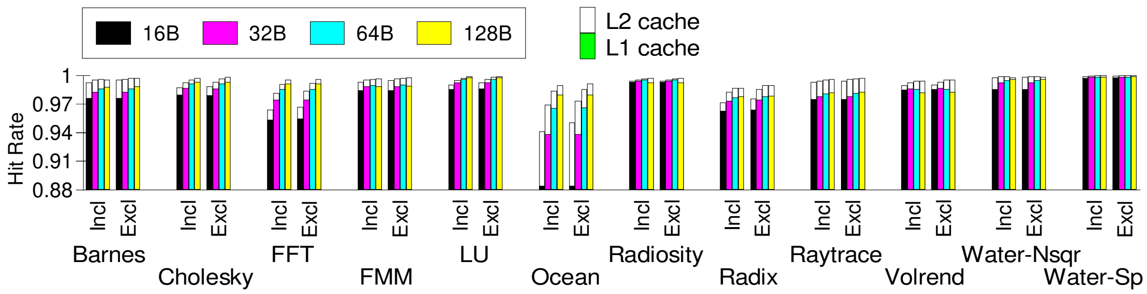


Figure 5.36: Performance sensitivity to line size

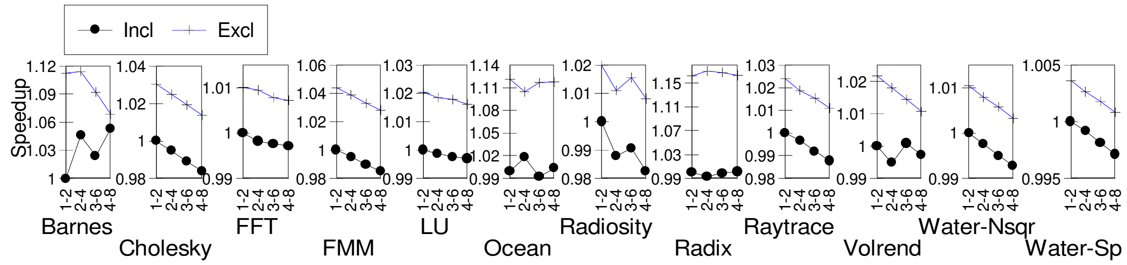


Figure 5.37: Performance sensitivity to L2 access time

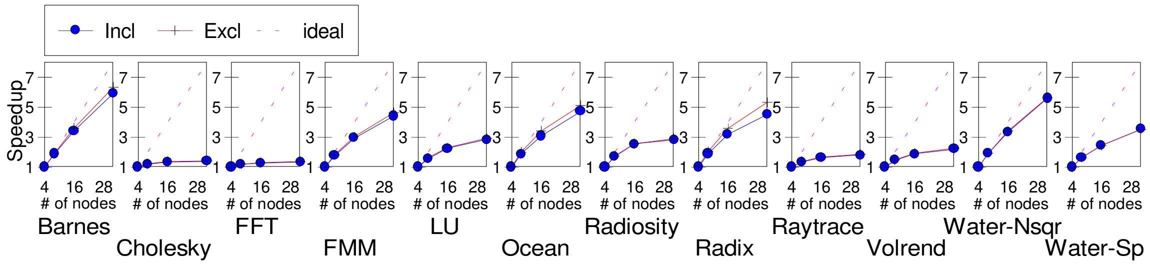


Figure 5.38: Performance sensitivity to the number of nodes

There are two ways to improve the performance of a CC-NUMA machine which has only the first-level cache. One way is to insert the L2 cache between the L1 cache and main memory. The other way is allowing binding to the local memory at a cache miss time as in DYMA. To compare these two architectures, we simulate DYMA with high memory pressure and E-NUMA varying the L2 size from 0 to 512K bytes. For these simulations, the L1 and L2 caches are 4-way associative. The tag and data access times for the L2 cache in E-NUMA is 2 and 4 cycles respectively. The results are shown in Figure 5.39. A program whose performance is highly sensitive to network traffic runs well on DYMA since the huge

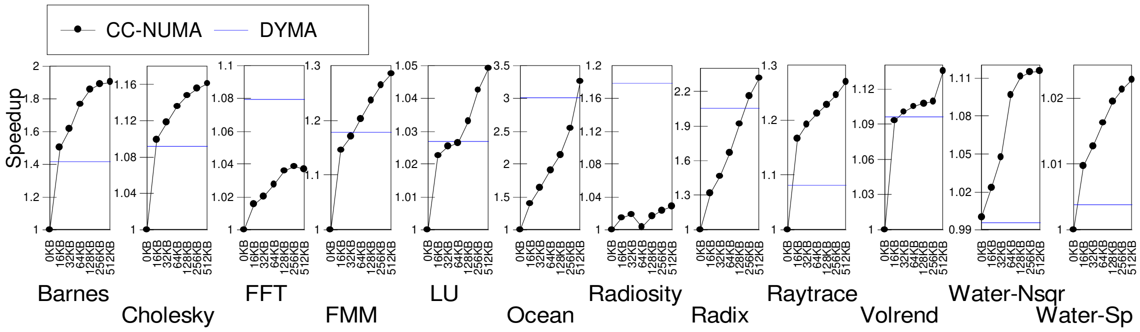


Figure 5.39: Performance comparison between E-NUMA and DYMA

size of the DM can reduce network requests. This is the reason why the speedup of DYMA in Ocean and Radix can be high without the L2 cache. However, if network traffic is low in a program, the huge size of the DM has little influence on performance. In this case, the architecture with the L2 cache is better than DYMA since the L2 cache is faster than the DM. Barnes, Cholesky, Raytrace, Water-Nsqr, and Water-Sp belong to this case. However, if the working set of a program can fit into the L1 cache, adding the L2 cache creates redundancy, i.e., it just increases initial data access time due to the L2 tag access time. Also, the page placement scheme at a page fault time in DYMA reduces network requests for compulsory L1 cache misses. These are the reasons why DYMA performs better in FFT and Radiosity than the architecture with two-level caches even though FFT and Radiosity have relatively low network traffic rates.

5.5 Limitation in simulation

In our simulation model, we find two main limitations in evaluating the performance of the DSM architectures. First, the program problem size and the cache size are not realistic. Second, we did not simulate multiprogramming workloads with I/O processing. Here, some questions arise: How many full-scale applications whose working set fits fully in the cache are there in the real world, and how do the patterns of reference in the multiprogramming environment affect the performance of the DSM architectures? We believe that there exist applications whose working sets are too large to fit fully in the cache. Even if most applications have a small working set size, DYMA may still have advantages in dynamic scheduling for load balance under a multiprogramming environment since DYMA is robust under dynamic change of locality of reference.

Chapter 6

Conclusion

New data may need to be bound to a memory location to improve performance. For the binding, existing data at the location must be unbound as well, due to limited amounts of memory. To avoid loss of data in the memory system, the replaced data may need to be relocated to some other memory location, leading to a new arrangement of data. The bound data can be invalidated for data consistency. The binding can incur extra overhead such as accesses to memory and usage of datapath bandwidth compared with non-binding. Thus, the binding should be committed only when the extra overhead can be amortized by faster data accesses under the new data arrangement in the future.

Since patterns of reference in a program are highly localized, a big performance improvement becomes possible by binding missing data to the cache at a cache miss time. For further improvement, DSM architectures try to bind new data to memory local to a node when an event of reference happens. The typical events of references are a cache miss, a node miss, and a page fault. However, we need to be careful because the extra overhead can hardly be amortized by the faster data access latency in the DSM architectures. Patterns of reference at the local memory are not as highly localized as at the cache. In general, the higher the cache hit rate is, the lower the local memory hit rate is. Also, replaced data can have the side effect of affecting the locality of data in a remote memory. Thus, it is questionable when the proper time is to bind new data to the local memory for faster data accesses under various patterns of reference.

Although there are a large number of possible data arrangements in DSMs, this thesis focuses on the three DSM architectures which have typical binding times to the local memory. CC-NUMA has a binding for the local memory at a page fault. Also, no different data addresses are bound to the location until the page is replaced from the memory by another page fault. COMA binds new data to the local memory at a node miss time. The data address of a memory location is not changed until the data is replaced by another node miss. DYMA binds new data to the local memory when the data is replaced from the cache at the time of a cache miss. Thus, unlike in CC-NUMA and COMA, a cache miss in DYMA can change the address of data bound to the local memory.

This thesis shows that the overhead for arranging data in each architecture varies depending on reference patterns. The working sets of applications continue to grow, and there are hundreds of applications whose working sets are too large to be supported even with main memory. Accordingly, people try to increase the size of the cache. Therefore, the area of applications whose working sets can fit in the cache is growing as well. For general purposes, a memory architecture should support a wide range of working sets.

To see how the three DSM architectures perform on realistic reference patterns, we design a machine for each architecture. To minimize directory size and to provide an efficient relocation algorithm for data replaced from the local memory, each directory has information about lines which are mapped to the same cache or memory set. Also, to reduce data access latency and number of requests, a non-blocking directory protocol is proposed. In that protocol, all requests can proceed without being blocked at the directory. To support the protocol, the history of data accesses is maintained with a small history table at the directory.

Our detailed program-driven simulations not only evaluate the performance of the three DSM machines but also test the correctness of the non-blocking directory protocol. The simulation results confirm that CC-NUMA suffers from seriously long data access latency when the cache does not match well with the working set of a program. COMA tries to reduce remote data accesses by utilizing locality of reference at the local memory. However, the data bound to the local memory at a node miss can be invalidated before being referenced again by a cache miss. Therefore, if references to the local memory are not strongly localized, most likely in a small working set, it is hard to amortize the overhead incurred at the node miss. Although large unallocated memory space in COMA can reduce the overhead, it can make the caching space of main memory for the data from disk small.

If a program has a small working set, the program will terminate its execution with few replacements from the cache. Thus, by delaying the binding of new data to the local memory from the time of a node miss to the time of a cache miss, DYMA performs like CC-NUMA in the programs. With large working sets, DYMA still has the advantage of reducing data access latency by taking advantage of reference locality at the local memory, performing better even than COMA due to lower overhead and better data arrangement at binding time. Few benchmark programs have strong reference patterns which can degrade the performance of DYMA. Further, simulation results show that DYMA is robust under reference patterns which actively share some memory lines. Therefore, DYMA can improve performance under a wide range of reference patterns.

Appendix A

Pseudo-code for a non-blocking directory protocol in CC-NUMA

```
--------------------------- PROCESSOR -------------------------
struct Request {
    Type (
        Rd_RH, Rd_HR, Rd_HO, Rd_ORs, Rd_ORf, // read request
        Rx_RH, Rx_HR, Rx_HO, Rx_ORs, Rx_ORf, // read excl. request
        Inv_RH, Inv_RHo, Inv_HR, Inv_HO, Inv_HS, Inv_SR,
                                             // inv. request
        Rl_RH, Rl_HO,                        // relocation request
        Rh_RH,                               // rep. hint request
        Ack,                                 // acknowledgement
        Flush                                // flush signal
    ),
    Addr,                                    // requested address
    Sid,                                     // source proc. id
    Sync,                                    // sync. counter
    Data
};

processor() {
    // create a request and send out the request for addr
    Request = {Rd or Rx, addr, sid, 0, null or word};
    send Request to Cache
};

----------------------------- CACHE ----------------------------
struct RSR {        // Requester Status Register
    Addr,           // requested address
    Word,           // word for write request
    Wowner,         // write hit on shared-owner state
    Inv,            // invalidated by a remote request
    Rel,            // relocated from a remote node
    Sync,           // synchronization counter
```

```
    wait_req        // remote requests waiting for requested data
}

struct Cache {
    Tag,         // address tag
    State (
        D_state, // dirty state
        S_state, // shared non-owner state
        O_state, // shared owner state
        I_state  // invalid state
    ),
    Data         // cache line
};

cache() {
switch (Request.Type) {
case Rd or Rx:
   if (Cache.Tag == Tag_of(Request.addr) &&
       Cache.State != I_state)                  // cache hit
        if (Request.Type == Rd) {          // read request
            // return the requested word to the processor
            Request.data = word_of(Cache.Data);
            send_to_processor(Request);
        }
        else if (Cache.State == D_state) { // Write on D-state
            // UPDATE data
            word_of(Cache.Data) = Request.Data;
            // send an acknowledgement to the processor
            Request.Data = null;
            send_to_processor(Request);
        }
        else {                         // Write on S- or O-state
            // save request information in RSR
            RSR.addr = Request.addr;
            if (Cache.State == O_state) {
                RSR.Wowner = 1;
                Request.Type = Inv_RHo;
            }
            else {
                RSR.Wowner = 0;
                Request.Type = Inv_RH;
            }
            RSR.word = Request.Data;
            // send an invalidation request to the home Directory
            Request.Data = null;
            send_to_directory(Request);
        }
   else {                                       // cache miss
        // save request information in RSR and send out
        // a data request
        RSR.Addr = Request.Addr;
        if (Request.Type == Rd)          // read request
            Request.Type = Rd_RH;
```

```
        else {                            // write request
            RSR.word = Request.Data;
            Request.Data = null;
            Request.Type = Rx_RH;
        }
        send Request to Directory;
        if (Cache.State != I_state) {
            // generate a replacement request
            Request = {, addr_of(Cache.Tag), sid, 0,};
            if (Cache.State == S_state)
                Request.Type = Rh_RH;
            else {
                Request.Type = Rl_RH;
                Request.Data = Cache.Data;
            }
            send Request to Directory;
            Cache.State = I_state;
        }
        Cache.Tag = Tag_of(RSR.Addr};
   }
   break;
case Rd_HR:
    Cache.State = S_state;
    Cache.Data = Request.Data;
    RSR.Addr = null;
    Request.Data = word_of(Request.Data);
    send_to_processor(Request);
    // wake up suspended requests with fail
    Wake_up(0);
    break;
case Rd_ORs:
    if (RSR.inv == 1) {// invalidated while issuing request
        // retry the request
        RSR.inv = 0;
        RSR.wait = 0;
        Request.Data = null;
        Request.Type = Rd_RH;
        send_to_directory(Request);
    }
    else {          // send valid data to processor
        Cache.State = S_state;
        Cache.Data = Request.Data;
        RSR.Addr = null;
        Request.Data = word_of(Request.Data);
        send_to_processor(Request);
    }
    // wake up suspended requests with fail
    wake_up(0);
    break;
case Rd_HO:
    if (RSR.Addr == Request.Addr)
        suspend(Request);
    else if (Cache.Tag == tag_of(Request.Addr) &&
```

```
            (Cache.State == D_state ||
             Cache.State == O_state)) { // get data
        Cache.State = O_state;
        Request.Data = Cache.Data;
        Request.Type = Rd_ORs;
        send_to_Requester(Request);
    }
    else {  /* miss data */
        Request.Type = Rd_ORf;
        send_to_requester(Request);
    }
    break;
case Rd_ORf:     /* Read miss fail */
    RSR.inv = 0;    // reset invalidate bit
    Request.Type = Rd_RH;
    send_to_directory(Request);
    break;
case Rl_HO:
case Ack:
case Rx_ORf:     /* Write miss or Invalidation completion */
    /* If state is I-state, write miss and Rel_wait is set.
       If state is S- or O-state, then invalidation */
    if (RSR.Rel == 1)   // check whether relocation request
        RSR.Rel = 0;    //       arrives or not
    else
        RSR.Rel = 1;
case Rx_HR:
case Rx_ORs:
    if (Request.Type == Rl_HO || Request.Type == Rx_HR ||
        Request.Type == Rx_ORs)
        Cache.Data = Request.Data;
case Inv_HR:
case Inv_SR:
    if (RSR.Sync == 0) {
        if (Request.Type != Rl_HO && Request.Type != Ack)
            RSR.Sync = Request.Sync;
    }
    if (RSR.Sync != 1) {
        if (Request.Type != Rl_HO && Request.Type != Ack)
            RSR.Sync = RSR.Sync - 1;
    }
    else if (RSR.Sync == 1 && RSR.Rel == 0) {
        // write access is completed
        Cache.State = D_state;
        word_of(Cache.Data) = RSR.word;
        Request.Data = null;
        send_to_Processor(Request);
        // wake up suspended requests with success
        wake_up(1);
    }
    break;
case Rx_HO:      /* Remote-Write request */
    if (RSR.Addr == Request.Addr)
```

```
        // cache is waiting for the same addr
        suspend(Request);
    else if (Cache.Tag == tag_of(Request.Addr) &&
             (Cache.State == D_state ||
              Cache.State == O_state)) { // get data
        Cache.State = I_state;
        Request.Data = Cache.Data;
        Request.Type = Rx_ORs;
        send_to_Requester(Request);
    }
    else {  /* miss data */
        Request.Type = Rx_ORf;
        send_to_Requester(Request);
    }
    break;
case Inv_HS:      /* Remote-Write invalidation request */
    Request.Type = Inv_SR;
    send_to_requester(Request);
    if (Cache.Tag == tag_of(Request.Addr))
        Cache.State = I_state;
    if (RSR.Addr == Request.Addr) {
        // cache is waiting for the same addr
        RSR.inv = 1;
        // wake up suspended request with fail
        wake_up(0);
    }
    break;
case Inv_HO:     /* Remote-Owner invalidation request */
    if (Cache.Tag == tag_of(Request.Addr) &&
        (Cache.State == D_state ||
         Cache.State == O_state)) { // hit
        Cache.State = I_state;
        Request.Type = Inv_SR;
        send_to_requester(Request);
    }
    else  { /* miss data */
        Request.Type = Rx_ORf;
        send_to_requester(Request);
    }
    if (RSR.Addr == Request.Addr)
        // cache is waiting for the same addr
        // wake up suspended request with fail
        wake_up(0);
    break;
case Flush:
    if (RSR.Addr == Request.Addr)
        // cache is waiting for the same addr
        if (RSR.Wowner == 1)
            // Flush signal is created by an invalidation request
            // wake up suspended request with success
            wake_up(1);
        else
            // Flush signal is created by a relocation request
```

```
            // wake up suspended request with fail
            wake_up(0);
    break;
}

suspend(Request) { /* assume one buffer size */
    if (wait_request != null) {
        /* If full, make it empty */
        wait_request.Type = Rd_ORf;
        send_to_requester(wait_request);
    }
    // save request in the buffer
    wait_request = Request;
}

wake_up(success) {
    while (wait_request != null)
        // If not empty, wake up the request
        if (success == 1 && wait_request.Type == Rd_HO) {
            // successful completion && waiting request is a read
            Cache.State = O_state;
            wait_request.Type = Rd_ORs;
        }
        else if (success == 1) {
            // successful completion && waiting request is a write
            Cache.State = I_state;
            wait_request.Type = Rx_ORs;
        }
        else if (wait_request.Type == Rd_HO)
            // unsuccessful completion && waiting is a read
            wait_request.Type = Rd_ORf;
        else
            // unsuccessful completion && waiting is a write
            wait_request.Type = Rx_ORf;
        send_to_requester(wait_request);
        wait_request = null;
}

---------------------------- DIRECTORY ---------------------------
define N_nodes      // number of nodes
struct Directory {      // assume a full bit-map vector directory
    Owner,          // owner node id. If (Owner == Nodes),
                    //   ownership is given to the local memory copy
    Bit_map[N_nodes]  // bit-map vector for sharing nodes
};

struct History[N_nodes] { // record the history of data accesses
    node_id,        // keep previous owner node
    state           // type of the data access; read-excl. or inv.
};

directory() {
switch (Request.Type) {
```

```
case Rd_RH:      /* read request from requester */
    if (Directory.Owner == N_nodes) {
        // home is the owner, i.e., local memory has a clean copy
        Directory.Bit_map[Request.Sid] = 1;     // set bit map
        Request.Data = Memory.Data;
        Request.Type = Rd_HR;
        send_to_requester(Request);
    }
    else {   /* invalid copy: forward to the ownership cache */
        Directory.Bit_map[Request.Sid] = 1;     // set bit map
        Request.Type = Rd_HO;
        send_to(Directory.Owner, Request);
    }
    break;
case Rx_RH:      /* read-exclusive request from requester */
    sync = 0;
    for (i = 0; i < N_nodes; i++)
        if (Directory.bit_map[i] == 1) {
            sharing_node[sync] = i;     // keep sharing node info.
            sync = sync + 1;            // count sync. info.
            Directory.bit_map[i] = 0;   // reset bit
        }
    if (Directory.Owner == N_nodes) {   // home is the owner
        sync = sync + 1;
        Request.Sync = sync;
        Request.Type = Rx_HR;
        send_to_requester(Request);
    }
    for (i = 0; i < sync; i++) { // send requests to sharing nodes
        Request.Sync = sync;         // put sync. info. in request
        if (Directory.Owner == sharing_node[i])
            Request.Type = Rx_HO;
        else
            Request.Type = Inv_HS;
        send_to(sharing_node[i], Request);
    }
    // update history table
    History[Request.Sid].node_id = Directory.Owner;
    History[Request.Sid].state = 0;     // read-exclusive request
    // keep directory info. for requester
    Directory.bit_map[Request.Sid] = 1;
    Directory.Owner = Request.Sid;
    break;
case Inv_RHo: // invalidation request from ownership requester
case Inv_RH:  // invalidation request from non-ownership requester
    sync = 0;
    for (i = 0; i < N_nodes; i++)
        if (Directory.bit_map[i] == 1) {
            sharing_node[sync] = i;   // keep sharing node info.
            sync = sync + 1;          // count sync. info.
            Directory.bit_map[i] = 0; // reset bit
        }
    if (Directory.bit_map[Request.Sid] == 0) {
```

```
    // directory does not consider requester as a sharing node
    // a different inv. or read-excl. request has passed
    // invalidation request is changed to read-excl. request
    if (Directory.Owner == N_nodes) {   // home is the owner
        sync = sync + 1;
        Request.Sync = sync;
        Request.Type = Rx_HR;
        send_to_requester(Request);
    }
    else if (Request.Type == Inv_RHo) { // from ownership
        Request.Type = Flush;
        send_to_requester(Request);
    }
    for (i = 0; i < sync; i++) { // send req. to sharing nodes
        Request.Sync = sync;    // put sync. info. in request
        if (Directory.Owner == sharing_node[i])
            Request.Type = Rx_HO;
        else
            Request.Type = Inv_HS;
        send_to(sharing_node[i], Request);
    }
    // update history table
    History[Request.Sid].node_id = Directory.Owner;
    History[Request.Sid].state = 0; // read-exclusive request
    // keep directory info. for requester
    Directory.bit_map[Request.Sid] = 1;
    Directory.Owner = Request.Sid;
}
else {
    // directory considers requester as a sharing node
    // no different inv. or read-excl. requests have passed
    // invalidation request is forwarded to sharing nodes
    if ((sync == 1 && Request.Type == Inv_RHo) ||
        Directory.Owner == N_nodes) {   // home is the owner
        sync = sync + 1;
        Request.Sync = sync;
        Request.Type = Inv_HR;
        send_to_requester(Request);
    }
    for (i = 0; i < sync; i++) { // send req. to sharing nodes
        Request.Sync = sync;    // put sync. info. in request
        if (Directory.Owner == sharing_node[i])
            Request.Type = Inv_HO;
        else
            Request.Type = Inv_HS;
        send_to(sharing_node[i], Request);
    }
    if (Directory.Owner != Request.Sid) {
        // update history table
        History[Request.Sid].node_id = Directory.Owner;
        History[Request.Sid].state = 1; // read-excl. request
    }
    // keep directory info. for requester
```

```
        Directory.bit_map[Request.Sid] = 1;
        Directory.Owner = Request.Sid;
    }
    break;
case Rl_RH:      /* relocation request */
    if (Directory.Owner == Request.Sid) {
        // requester is the owner
        Memory.Data = Request.Data; // save data in local memory
        Directory.bit_map[Request.Sid] = 0; // reset bit map
    }
    else {
        // a different inv. or read-excl. request has passed
        // find the first node which took the ownership
        first_node = Directory.Owner;
        for (i = 0; i < N_nodes; i++)
            if (History[first_node].node_id == Request.Sid)
                break;
            else
                first_node = History[first_node].node_id;
        if (History[first_node].state == 0) {
            // the first node's request is a read-excl. request
            // forward relocation request to the first node
            Request.Type = Rl_HO;
            send_to(first_node, Request);
            // send a flush signal to the requester
            Request.Type = Flush;
            send_to_request(Request);
        }
        else {
            // the first node's request is an inv. request
            // send an acknowledgement to the first node
            Request.Type = Ack;
            send_to(first_node, Request);
        }
    }
    break;
case Rh_RH: /* replacement-hint request */
    Directory.bit_map[Request.Sid] = 0; // reset bit map
    break;
}
```

Bibliography

[AB84] James Archibald and Jean-Loup Baer. An economical solution to the cache coherence problem. In *Proceedings of the 11th Annual International Symposium on Computer Architecture*, pages 355–362, Ann Arbor, Michigan, June 5–7, 1984. IEEE Computer Society and ACM SIGARCH.

[AB86] James Archibald and Jean-Loup Baer. Cache coherence protocols evaluation using a multiprocessor simulation model. *ACM Transaction on Computer Systems*, 4(4):273–298, November 1986.

[ABC+95] Anant Agarwal, Ricardo Bianchini, David Chaiken, Kirk L. Johnson, David Kranz, John Kubiatowicz, Beng-Hong Lim, Kenneth Mackenzie, and Donald Yeung. The MIT Alewife machine: Architecture and performance. In *Proceedings of the 22nd Annual International Symposium on Computer Architecture*, pages 2–13. *Computer Architecture News,* 23(2), May 1994.

[AG96] Sarita V. Adve and Kourosh Gharachorloo. Shared memory consistency models: A tutorial. *IEEE Computer, special issue on shared-memory multiprocessing*, pages 66–76, December 1996.

[ASHH88] Anant Agarwal, Richard Simoni, John Hennessy, and Mark Horowitz. An evaluation of directory schemes for cache coherence. In *Proceedings of the 15th Annual International Symposium on Computer Architecture*, pages 280–289. *Computer Architecture News,* 16(2), May 1988.

[BA97] Tony Brewer and Greg Astfalk. The evolution of the HP/convex exemplar. In *Proceedings of COMPCON Spring '97: Forty-Second IEEE Computer Society International Conference*, pages 81–96, February 1997.

[BLA+94] Matthias A. Blumrich, Kai Li, Richard Alpert, Cezary Dubnicki, Edward W. Felten, and Jonathan Sandberg. Virtual memory mapped network interface for the SHRIMP multicomputer. In *Proceedings of the 21st Annual International Symposium on Computer Architecture*, pages 142–153. *Computer Architecture News,* 22(2), April 1994.

[BT98] Sujoy Basu and Josep Torrellas. Enhancing memory use in simple coma: Multiplexed simple coma. In *Proceedings of the Fourth International Symposium* *on High-Performance Computer Architecture*, Las Vegas, Nevada, January 31 – February 4, 1998.

[BW88] Jean-Loup Baer and Wen-Hann Wang. On the inclusion properties for multi-level cache hierarchies. In *Proceedings of the 15th Annual International Symposium on Computer Architecture*, pages 73–80. *Computer Architecture News,* 16(2), May 1988.

[CDV+94] Rohit Chandra, Scott Devine, Ben Verghese, Anoop Gupta, and Mendel Rosenblum. Scheduling and page migration for multiprocessor compute servers. In *Proceedings of the Sixth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 12–24. *Computer Architecture News,* 22, October 1994; *Operating Systems Review,* 28(5), December 1994; *SIGPLAN Notices,* 29(11), November 1994.

[CF78] L. Censier and P. Feautrier. A new solution to coherence problem in multicache systems. *IEEE Transactions on Computers*, (C-27)12:1112–1118, June 1978.

[CF93] Alan L. Cox and Robert J. Fowler. Adaptive cache coherency for detecting migratory shared data. In *Proceedings of the 20th Annual International Symposium on Computer Architecture*, pages 98–108. *Computer Architecture News,* 21(2), May 1993.

[CKA91] David Chaiken, John Kubiatowicz, and Anant Agarwal. LimitLESS directories: A scalable cache coherence scheme. In *Proceedings of the Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 224–234, Santa Clara, California, April 8–11, 1991. ACM SIGARCH, SIGPLAN, SIGOPS, and the IEEE Computer Society. *Computer Architecture News,* 19(2), April 1991; *Operating Systems Review,* 25, April 1991; *SIGPLAN Notices,* 26(4), April 1991.

[CKL99] Sangyeun Cho, Jinseok Kong, and Gyungho Lee. Coherence and replacement protocol of DICE - a bus-based COMA multiprocessor DICE. *Journal of Parallel and Distributed Computing*, 57(1):14–32, April 1999.

[Cor97] Data General Corporation. Aviion AV 20000 server technical overview. Technical report, 1997.

[CSG97] David Culler, Jaswinder Pal Singh, and Anoop Gupta. *Parallel Computer Architecture: A Hardware / Software Approach.* Morgan Kaufmann, San Mateo, CA, 1997.

[DCC+87] William J. Dally, Linda Chao, Andrew Chien, Soha Hassoun, Waldemar Horwat, Jon Kaplan, Paul Song, Brian Totty, and Scott Wills. Architecture of a Message-Driven Processor. In *Proceedings of the 14th Annual International Symposium on Computer Architecture*, pages 189–196. *Computer Architecture News,* 15(2), June 1987.

[Den68] P. J. Denning. Thrashing: Its causes and prevention. *AFIPS Conf. Proc.*, 11:915–922, May 1968.

[DL97] Fredrik Dahlgren and Anders Landin. Reducing the replacement overhead in bus-based COMA multiprocessors. In *Proceedings of the Third International Symposium on High-Performance Computer Architecture*, pages 14–23, San Antonio, Texas, February 1–5, 1997. IEEE Computer Society TCCA.

[FW97] Babak Falsafi and David A. Wood. Reactive NUMA: A design for unifying S-COMA and CC-NUMA. In *Proceedings of the 24th Annual International Symposium on Computer Architecture*, pages 229–240, Denver, Colorado, June 2–4, 1997.

[Goo83] James R. Goodman. Using cache memory to reduce processor-memory traffic. In *Proceedings of the 10th Annual International Symposium on Computer Architecture*, pages 124–131. *Computer Architecture News,* 11(3), June 1983.

[HHL92] Erik Hagersten, Seif Haridi, and Anders Landin. DDM - a cache-only memory architecture. *IEEE Computer Magazine*, pages 44–54, September 1992.

[IDFL96] Liviu Iftode, Cesary Dubnicki, Edward W. Felten, and Kai Li. Improving release-consistent shared virtual memory using automatic update. In *Proceedings of the Second International Symposium on High-Performance Computer Architecture*, pages 14–25, San Jose, California, February 3–7, 1996.

[IFKR92] H. Burkhardt III, S. Frank, B. Knobe, and J. Rothnie. Overview of the KSR1 computer system. *Technical Report KSR-TR-9202001*, February 1992.

[JH94] Truman Joe and John L. Hennessy. Evaluating the memory overhead required for COMA architectures. In *Proceedings of the 21st Annual International Symposium on Computer Architecture*, pages 82–93. *Computer Architecture News,* 22(2), April 1994.

[JLGS90] D. V. James, A. T. Laundrie, S. Gjessing, and G. S. Sohi. New directories in scalable shared memory multiprocessor architectures: Scalable coherent interface. *IEEE Computer Magazine*, (23)6:74–77, June 1990.

[Joe95] Truman Joe. *COMA-F: A Non-hierarchical Cache Only Memory Architecture.* Ph.D. thesis, Computer Systems Laboratory, Stanford University, March 1995.

[Jor83] Harry F. Jordan. Performance measurements on HEP — a pipelined MIMD computer. In *Proceedings of the 10th Annual International Symposium on Computer Architecture*, pages 207–212. *Computer Architecture News,* 11(3), June 1983.

[Jou90] Norman P. Jouppi. Improving direct-mapped cache performance by the addition of a small fully-associative cache and prefetch buffers. In *Proceedings of the 17th*

Annual International Symposium on Computer Architecture, pages 364–373. *Computer Architecture News,* 18(2), June 1990.

[JW94] Norman P. Jouppi and Steven J. E. Wilton. Tradeoffs in two-level on-chip caching. In *Proceedings of the 21st Annual International Symposium on Computer Architecture*, pages 34–45. *Computer Architecture News,* 22(2), April 1994.

[KCDZ94] P. Keleher, A. L. Cox, S. Dwarkadas, and W. Zwaenepoel. Treadmarks: Distributed shared memory on standard workstations operating systems. In *Proceedings of Winter USENIX Conference*, pages 15–132, January 1994.

[KCKS98] Chen-Chi Kuo, John Carter, Ravindra Kuramkote, and Mark Swanson. AS-COMA: An adaptive hybrid shared memory architecture. In *Proceedings of the International Conference on Parallel Processing*, pages 207–216, August 1998.

[KEW+85] R. H. Katz, S. J. Eggers, D. A. Wood, C. L. Perkins, and R. G. Sheldon. Implementing a cache consistency protocol. In *Proceedings of the 12th Annual International Symposium on Computer Architecture*, pages 276–283, Boston, Massachusetts, June 17–19, 1985. IEEE Computer Society TCA and ACM SIGARCH. *Computer Architecture News,* 13(3), June 1985.

[KGGK94] Vipin Kumar, Ananth Grama, Anshul Gupta, and George Karypis. *Introduction to Parallel Computing.* The Benjamin/cummings Publishing Company, 1994.

[KL95] Jinseok Kong and Gyungho Lee. DYMA: A novel memory architecture for distributed shared memory multiprocessors. DICE Project Technical Report No. 14, Department of Electrical Engineering, University of Minnesota, May 1995.

[KL96] Jinseok Kong and Gyungho Lee. Relaxing the inclusion property in cache only memory architecture. In *Proceedings in the Euro-Par'96 Parallel Processing*, volume II, pages 435–444, Lyon, France, August 26–29 1996.

[KOH+94] Jeffrey Kuskin, David Ofelt, Mark Heinrich, John Heinlein, Richard Simoni, Kourosh Gharachorloo, John Chapin, David Nakahira, Joel Baxter, Mark Horowitz, Anoop Gupta, Mendel Rosenblum, and John Hennessy. The Stanford FLASH multiprocessor. In *Proceedings of the 21st Annual International Symposium on Computer Architecture*, pages 302–313. *Computer Architecture News,* 22(2), April 1994.

[Lam79] L. Lamport. How to make a multiprocessor computer that correctly executes multiprocess programs. *IEEE Transactions on Computers*, (C-28)9:690–691, September 1979.

[LC96] Tom Lovett and Russell Clapp. STiNG: A CC-NUMA computer system for the commercial marketplace. In *Proceedings of the 23rd Annual International Symposium on Computer Architecture*, pages 308–317, Philadelphia, Pennsylvania, May 22–24, 1996.

[LCS98] T. D. Lovett, R. M. Clapp, and R. J. Safranek. NUMA-Q: An SCI-based enterprise server. Technical report, Sequent Computer Systems, Inc., 1998.

[LD96] Anders Landin and Fredrik Dahlgren. Bus-based COMA — reducing traffic in shared-bus multiprocessors. In *Proceedings of the Second International Symposium on High-Performance Computer Architecture*, pages 95–105, San Jose, California, February 3–7, 1996.

[LH89] Kai Li and P. Hudak. Memory coherence in shared virtual memory systems. *ACM Trans. on Computer Systems*, 7(4):321–359, November 1989.

[LK95] Gyungho Lee and Jinseok Kong. Prospects of distributed shared memory for reducing global traffic in shared-bus multiprocessors. In *Proceedings of the 7th IASTED-ISMM International Conference on Parallel and Distributed Computing and Systems*, pages 63–67, Washington D.C., October 1995.

[LL97] James Laudon and Daniel Lenoski. The SGI origin: A ccNUMA highly scalable server. In *Proceedings of the 24th Annual International Symposium on Computer Architecture*, pages 241–251, Denver, Colorado, June 2–4, 1997.

[LLG+90] Daniel Lenoski, James Laudon, Kourosh Gharachorloo, Anoop Gupta, and John Hennessy. The directory-based cache coherence protocol for the DASH multiprocessor. In *Proceedings of the 17th Annual International Symposium on Computer Architecture*, pages 148–159. *Computer Architecture News,* 18(2), June 1990.

[LQK94] Gyungho Lee, B. Quattlebaum, and L. Kinney. Protocol mapping in bus-based COMA multiprocessors. DICE Project Technical Report No. 10, Department of Electrical Engineering, University of Minnesota, March 1994.

[LYL87] Roland L. Lee, Pen-Chung Yew, and Duncan H. Lawrie. Multiprocessor cache design considerations. In *Proceedings of the 14th Annual International Symposium on Computer Architecture*, pages 253–262. *Computer Architecture News,* 15(2), June 1987.

[MD98] Adrian Moga and Michel Dubois. The effectiveness of SRAM network caches in clustered DSMs. In *Proceedings of the Fourth International Symposium on High-Performance Computer Architecture*, Las Vegas, Nevada, January 31 – February 4, 1998.

[NAB+95] A. Nowatzyk, G. Aybay, M. Browne, E. Kelly, M. Parkin, B. Radke, and S. Vishi. The s3.mp scalable shared memory multiprocessor. In *Proceedings* *of the 1995 International Conference on Parallel Processing*, pages I1–I10, August 1995.

[PC90] Gregory M. Papadopoulos and David E. Culler. Monsoon: An explicit token-store architecture. In *Proceedings of the 17th Annual International Symposium on Computer Architecture*, pages 82–91. *Computer Architecture News,* 18(2), June 1990.

[PD95] Fong Pong and Michel Dubois. A new approach for the verification of cache coherence protocols. *IEEE Transactions on Parallel and Distributed Systems*, 6(8), August 1995.

[PH96] David A. Patterson and John L. Hennessy. *Computer Architecture: A Quantitative Approach.* Morgan Kaufmann, San Mateo, CA, second edition, 1996.

[RLW94] Steven K. Reinhardt, James R. Larus, and David A. Wood. Tempest and Typhoon: User-level shared memory. In *Proceedings of the 21st Annual International Symposium on Computer Architecture*, pages 325–336. *Computer Architecture News,* 22(2), April 1994.

[RPW96] Steven K. Reinhardt, Robert W. Pfile, and David A. Wood. Decoupled hardware support for distributed shared memory. In *Proceedings of the 23rd Annual International Symposium on Computer Architecture*, pages 34–43, Philadelphia, Pennsylvania, May 22–24, 1996.

[SBS93] Per Stenström, Mats Brorsson, and Lars Sandberg. An adaptive cache coherence protocol optimized for migratory sharing. In *Proceedings of the 20th Annual International Symposium on Computer Architecture*, pages 109–118. *Computer Architecture News,* 21(2), May 1993.

[SFL+94] Ioannis Schoinas, Babak Falsafi, Alvin R. Lebeck, Steven K. Reinhardt, James R. Larus, and David A. Wood. Fine-grain access control for distributed shared memory. In *Proceedings of the Sixth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 297–306. *Computer Architecture News,* 22, October 1994; *Operating Systems Review,* 28(5), December 1994; *SIGPLAN Notices,* 29(11), November 1994.

[SGT96] Daniel J. Scales, Kourosh Gharachorloo, and Chandramohan A. Thekkath. Shasta: A low overhead, software-only approach for supporting fine-grain shared memory. In *Proceedings of the Seventh International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 174–185, Cambridge, Massachusetts, October 1–5, 1996. ACM SIGARCH, SIGOPS, SIGPLAN, and the IEEE Computer Society. *Computer Architecture News,* 24, October 1996; *Operating Systems Review,* 30(5), December 1996; *SIGPLAN Notices,* 31(9), September 1996.

[SH91] R. Simoni and M. Horowitz. Dynamic pointer allocation for scalable cache coherence directories. In *Proceedings of the International Symposium on Shared Memory Multiprocessing*, pages 72–81, April 1991.

[Sim95] Richard Simoni. *Cache Coherence Directory for Scalable Multiprocessors*. Ph.D. thesis, Computer Systems Laboratory, Stanford University, March 1995.

[SJG92] Per Stenström, Truman Joe, and Anoop Gupta. Comparative performance evaluation of cache-coherent NUMA and COMA architectures. In *Proceedings of the 19th Annual International Symposium on Computer Architecture*, pages 80–91, Gold Coast, Australia, May 19–21, 1992. ACM SIGARCH and IEEE Computer Society TCCA. *Computer Architecture News,* 20(2), May 1992.

[Smi82] Alan Jay Smith. Cache memories. *ACM Computing Surveys*, 14(3):473–530, September 1982.

[SS86] Paul Sweazey and Alan Jay Smith. A class of compatible cache consistency protocols and their support by the IEEE Futurebus. In *Proceedings of the 13th Annual International Symposium on Computer Architecture*, pages 414–423, Tokyo, Japan, June 2–5, 1986. IEEE Computer Society TCCA, ACM SIGARCH, and the Information Processing Society of Japan. *Computer Architecture News,* 14(2), June 1986.

[SS88] Dennis Shasha and Marc Snir. Efficient and correct execution of parallel programs that share memory. *ACM Transactions on Programming Languages and Systems*, 10(2):282–312, April 1988.

[SWCL95] Ashley Saulsbury, Tim Wilkinson, John Carter, and Anders Landin. An argument for simple COMA. In *Proceedings of the First International Symposium on High-Performance Computer Architecture*, pages 276–285, Raleigh, North Carolina, January 22–25, 1995. IEEE Computer Society TCCA.

[Tan76] C.K. Tang. Cache system design in the tightly coupled multiprocessor system. In *Proceedings of the National Computer Conference*, pages 749–753. Association for Computing Machinery, 1976.

[TLH94] Josep Torrellas, Monica S. Lam, and John L. Hennessy. False sharing and spatial locality in multiprocessor caches. *IEEE Transactions on Computers*, 43(6):651–663, June 1994.

[VF94] Jack E. Veenstra and Robert J. Fowler. Mint: A front-end for efficient simulation of shared-memory multiprocessors. In *Proceedings of the Second MASCOTS*, Jan-Feb 1994.

[WGH+97] Wolf-Dietrich Weber, Stephen Gold, Pat Helland, Takeshi Shimizu, Thomas Wicki, and Winfried Wilcke. The mercury interconnect architecture: A cost-effective infrastructure for high-performance servers. In *Proceedings of the*

24th Annual International Symposium on Computer Architecture, pages 98–107, Denver, Colorado, June 2–4, 1997.

[Wil87] Andrew W. Wilson, Jr. Hierarchical cache/bus architecture for shared memory multiprocessors. In *Proceedings of the 14th Annual International Symposium on Computer Architecture*, pages 244–252. *Computer Architecture News,* 15(2), June 1987.

[WOT+95] Steven Cameron Woo, Moriyoshi Ohara, Evan Torrie, Jaswinder Pal Shingh, and Anoop Gupta. The SPLASH-2 programs: Characterization and methodological considerations. In *Proceedings of the 22nd Annual International Symposium on Computer Architecture*, pages 24–36. *Computer Architecture News,* 23(2), May 1994.

[WSH94] Steven Cameron Woo, Jaswinder Pal Singh, and John L. Hennessy. The performance advantages of integrating block data transfer in cache-coherent multiprocessors. In *Proceedings of the Sixth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 219–229. *Computer Architecture News,* 22, October 1994; *Operating Systems Review,* 28(5), December 1994; *SIGPLAN Notices,* 29(11), November 1994.

[ZCT99] Zheng Zhang, Marcelo Cintra, and Josep Torrellas. Excel-NUMA: Toward programmability, simplicity, and high performance. *IEEE Transactions on Computers, Special Issue on Cache Memory*, February 1999.

[ZWR+80] Pitro Zafiropulo, Colin H. West, Harry Rudin, D.D. Cowan, and Daniel Brand. Towards analyzing and synthesizing protocols. *IEEE Transactions on Communications*, 28(4), April 1980.