This is to certify that the

dissertation entitled

DATA PLACEMENT IN SHARED-VIRTUAL-MEMORY MULTIPROCESSORS WITH NON-UNIFORM MEMORY ACCESS TIMES

presented by

JAYASHREE RAMANATHAN

has been accepted towards fulfillment
of the requirements for

PhD degree in Computer Science

Lionel M Ni
Major professor

Date July 1, 1992

Data Placement in Shared-Virtual-Memory Multiprocessors with Non-Uniform Memory Access Times

By

Jayashree Ramanathan

A DISSERTATION

Submitted to
Michigan State University
in partial fulfillment of the requirements
for the degree of

DOCTOR OF PHILOSOPHY

Department of Computer Science

1992

ABSTRACT

Data Placement in Shared-Virtual-Memory Multiprocessors with Non-Uniform Memory Access Times

By

Jayashree Ramanathan

Each processor in a multiprocessor usually has some local memory and, in addition, it may share a global memory with all the other processors. Such a memory organization is motivated by price/performance reasons, but results in a non-uniform memory access (NUMA) time due to the difference in the latency of the interconnects used to access the local and the non-local memories. Many NUMA multiprocessors allow processes of an application to interact by means of a shared virtual memory because of the advantages of virtual memory and the ease of programming using the shared-memory model. To achieve an acceptable performance in these multiprocessors, it is important to properly place the data of a parallel application in the memory hierarchy. The primary aim of this thesis is to identify the limitations of existing data placement techniques and to develop better methods of placing data when providing a shared virtual memory in NUMA multiprocessors.

Existing data placement techniques include data replication and data migration, both of which place data in terms of blocks such as pages or cache lines. Using analysis and trace-driven simulations, we study the factors affecting page-level replication, and

show that it needs to be adaptive to page reference patterns and hardware parameters such as the available physical memory and the time to transfer data between the local and the non-local memories. We also demonstrate that proper layout of data in the shared virtual memory reduces the false sharing of each page, creates simple page reference patterns, and simplifies data placement. These results apply to other block-level placement strategies as well. In the absence of help from the compiler or the applications programmer however, such strategies incur runtime overhead when being adaptive and further, are unable to optimally place blocks that are falsely-shared and those whose reference patterns change rapidly.

The conclusions from our investigation thus motivate our study of compiler-assisted data placement. Our approach to assist data placement is by using compile-time objects containing data of the same variable type and with similar reference patterns. We demonstrate how our approach can be incorporated in a compiler. We then develop a compile-time object-creation scheme that assists block-level placement; we also propose solutions to several related issues. Next, we develop a scheme that assists object-level placement which requires the applications programmer to specify when data placement operations are needed. We derive the compile-time overhead of applying our schemes. We also compare the performance of applications for these different types of placement schemes by experimental simulations. We demonstrate that there is significant performance improvement when the compiler assists data placement. Further, the best performance is obtained when data placement operations are entirely specified by the compiler. In summary, compiler-directed data placement is a promising approach to improve the performance of applications and the programmability of NUMA multiprocessors.

ACKNOWLEDGEMENTS

I owe much to my parents. My mother, who devoted her life to the upbringing of her children motivated us to be independent and well educated. It is unfortunate that she did not live to see me complete my Ph.D. But she was and still continues to be a great source of inspiration and motivation for me. My father encouraged us to pursue higher education and to excel in whatever we chose to do. I gladly dedicate this thesis to my parents.

I am fortunate to have made several good friends along the way. Some of them helped shape my interests and values, out of which I must mention Sudha, Nirmala, and Ravishankar. I also enjoyed and continue to enjoy the friendship of Chitra, CS, John, Poonam, Rajni, and Rama.

To my parents

I am thankful to our department for offering me a teaching assistantship which enabled me to come to the U.S. to pursue a Ph.D. I owe thanks to both Ms. Sasikala Reddy of the School of Osteopathic Medicine, and Dr. Dirk Goodman, the Director of the Case Center, for providing me research assistantships during my first summer in the U.S. Thanks are also due to my thesis advisor Prof. Lionel Ni and our chairperson Prof. Anthony Wojcik for hiring me as a lab manager from 1988 till 1991. The lab manager position not only provided me financial assistance, but also enriched my technical and interpersonal skills. I enjoyed working for Frank Northrup, our department's computing facilities coordinator. My thanks to our graduate secretary Lora Mae Higbee for her enormous help in administrative matters.

The courses that I took with Prof. Richard Enbody got me interested in the broad area of my dissertation. I also enjoyed participating in the meetings of the parallel processing and the distributed computing research groups organized by Prof. Richard Enbody and Prof. Matt Mutka, respectively.

I thank Krish Kannan of the IBM T.J. Watson Research Center for offering me a summer internship in 1989. My thanks are also due to Paul Castel, Pyau Allen, and Jeanne Ferrante, all of the IBM T.J. Watson Research Center for providing me with research papers and positive feedback on my dissertation work.

I thank Prof. Lionel Ni for suggesting that I look at the Mach operating system for my term paper work. He also encouraged me to apply to the IBM T.J. Watson

ACKNOWLEDGEMENTS

I owe much to my parents. My mother who devoted her life to the upbringing of her children motivated us to be independent and well-educated. It is unfortunate that she did not live to see me complete my Ph.D. But she was and still continues to be a great source of inspiration and motivation for me. My father encouraged us to pursue higher education and to excel in whatever we chose to do. I gladly dedicate this thesis to my parents.

I am fortunate to have made several good friends along the way. Some of them helped shape my interests and values, out of whom I must mention Sudha, Nirmala, and Ravishankar. I also enjoyed and continue to enjoy the friendship of Chitra, CS, John, Poonam, Rajni, and Rama Seshu.

I am thankful to our department for offering me a teaching assistantship which enabled me to come to the U.S. to pursue a Ph.D. I owe thanks to both Ms. Sasikala Reddy of the School of Osteopathic Medicine, and Dr. Erik Goodman, the Director of the Case Center, for providing me research assistantships during my first summer in the U.S. Thanks are also due to my thesis advisor Prof. Lionel Ni and our chairperson Prof. Anthony Wojick for hiring me as a lab manager from 1988 till 1991. The lab manager position not only provided me financial assistance, but also enriched my technical and interpersonal skills. I enjoyed working for Frank Northrup, our department's computing facilities coordinator. My thanks to our graduate secretary Lora Mae Higbee for her enormous help in administrative matters.

The courses that I took with Prof. Richard Enbody got me interested in the broad area of my dissertation. I also enjoyed participating in the meetings of the parallel processing and the distributed computing research groups organized by Prof. Richard Enbody and Prof. Matt Mutka, respectively.

I thank Krish Kannan of the IBM T.J. Watson Research Center for offering me a summer internship in 1989. My thanks are also due to Paul Carini, Fran Allen, and Jeanne Ferrante, all of the IBM T.J. Watson Research Center for providing me with research papers and positive feedback on my dissertation work.

I thank Prof. Lionel Ni for suggesting that I look at the Mach operating system for my term paper work. He also encouraged me to apply to the IBM T.J. Watson

Research Center for a summer internship. Both these factors eventually led to the choice of my dissertation topic. I also thank him for his understanding when I decided to complete my dissertation in Poughkeepsie, New York. I appreciate the hospitality shown by him and his wife.

Finally, I owe much to my husband Ram for his love, encouragement, and support throughout my dissertation work. He patiently bore with my long hours of work, and took care of most of the household chores which significantly reduced the time it took to write this thesis.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF TABLES

LIST OF FIGURES

CHAPTER 1

INTRODUCTION

The speed of a single-processor computer has increased by several orders of magnitude since the first electronic digital computer was introduced in 1940, primarily due to advances in hardware technology. However, many important problems remain unsolved as their solution requires computational power that are far beyond the capabilities of the fastest single-processor computers currently available. For instance, there exist problems that need computers capable of executing 10^{12} floating point operations per second (teraflops). Since hardware technology is approaching its physical limitations, such computational power can be realized only by introducing parallelism in computers. An important class of parallel computers are *multiprocessors*, which consist of multiple processors and memory modules connected together by one or more interconnects.

In this thesis, we consider multiprocessors with a *non-uniform memory access* (NUMA) time, i.e., the access time is not the same for all memory locations. Processors in such a multiprocessor concurrently execute either different applications or different processes of the same parallel application. We consider the latter case and assume that there are enough processors so that each application is allocated as many processors as the number of processes it creates and further, each process is statically assigned to a given processor. Most existing multiprocessor operating systems, such as the BBN's nX [1], allow such exclusive allocation of resources to a given application, thereby allowing its performance to be maximized, independent of other applications. We also assume that the processes of an application communicate and synchronize by

using the same virtual addresses for data they share. This method of communication is achieved by assigning addresses to the application's data and code from a set of virtual addresses shared by all processors, referred to as the *shared virtual memory* (SVM). Due to the variation in cost to access different physical memory locations in a NUMA multiprocessor, data in the SVM needs to be placed in the NUMA physical memory such that the time to access it is minimized for all processors. In this thesis, we propose and evaluate new data placement techniques that help to achieve this goal.

In this chapter, we first present a typical architecture of a NUMA multiprocessor and then discuss the advantages of a SVM and the implementation issues in providing it in a NUMA multiprocessor. Next we provide an overview of existing techniques to address the problem of data placement. We then summarize the contributions of our thesis in addressing the data placement problem. We conclude with an overview of the remaining chapters.

1.1 A NUMA Multiprocessor

Typically, each processor in a multiprocessor has some memory placed local to it and, in addition, it may share a global memory with all the other processors as shown in Figure 1.1. Such a memory organization is motivated by price/performance reasons similar to the cache and the main memory hierarchy prevalent in uniprocessors. For example, processors in small-scale multiprocessors such as the Alliant's FX/8 [2] and the Sequent's Symmetry [3] are connected by a single bus interconnect to a global main memory. Each processor has a local cache, and a snoopy hardware cache protocol keeps all the caches consistent by listening to memory accesses on the shared bus.

A single bus is limited in bandwidth for large-scale multiprocessors, and technology limitations make it too expensive to provide hardware connectivity from each processor to every other processor. Therefore, large-scale multiprocessors are built with intermediate connectivity using interconnects such as multi-stage interconnects as in the BBN's TC2000 [1] and the IBM's RP3 [4], point-to-point interconnects as

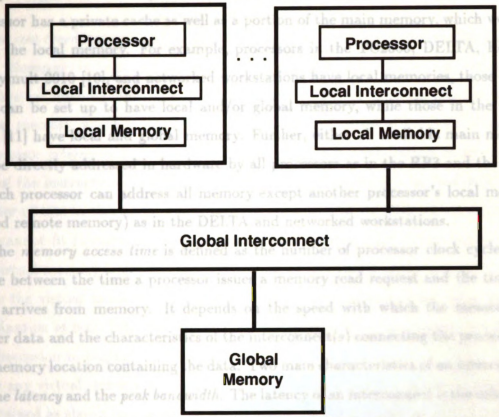

Figure 1.1. A NUMA multiprocessor.

in the Intel's DELTA [5], and hierarchical interconnects as in the Kendall Square Research's KSR1 [6] and the DASH multiprocessor [7]. Since there is no longer a single bus to snoop at, some of these multiprocessors such as the TC2000, RP3, and DELTA do not provide hardware cache consistency. Others use a directory-based cache consistency protocol to maintain information about cache lines in either centralized or distributed directories as in the DASH. Some others, such as the KSR1, use a hardware protocol that traverses the interconnect hierarchically. Details of the IEEE scalable coherent interface and related projects can be found in [8].

Main memory in large-scale multiprocessors includes modules global to all processors just as in small-scale multiprocessors. In addition, there are modules placed local to each processor. Note that in small-scale multiprocessors, the memory local to

each processor includes only its cache. However, in large-scale multiprocessors, each processor has a private cache as well as a portion of the main memory, which we refer to as the local memory. For example, processors in the TC2000, DELTA, Horizon [9], Symult 2010 [10], and networked workstations have local memories, those in the RP3 can be set up to have local and/or global memory, while those in the IBM's ACE [11] have local and global memory. Further, either all available main memory can be directly addressed in hardware by all processors as in the RP3 and the ACE, or each processor can address all memory except another processor's local memory (called remote memory) as in the DELTA and networked workstations.

The *memory access time* is defined as the number of processor clock cycles that elapse between the time a processor issues a memory read request and the time the data arrives from memory. It depends on the speed with which the memory can deliver data and the characteristics of the interconnect(s) connecting the processor to the memory location containing the data. Two main characteristics of an interconnect are the *latency* and the *peak bandwidth*. The latency of an interconnect is the minimum number of clock cycles required to transfer one word of data either from one processor to another, or from one processor to a memory module. The peak bandwidth gives the maximum rate at which data can be transferred across the interconnect and is usually measured in words/cycle [12]. The variation in the latencies of the interconnects used to access the different memory locations results in a non-uniform memory access time, and hence the name NUMA multiprocessors. For example, the difference in the times to access the cache and main memory gives rise to a NUMA time in small-scale multiprocessors. An additional NUMA time in large-scale multiprocessors is due to the difference in the times to access local, global, and remote memory. As an example, in TC2000, a remote memory access takes four times longer than a local memory access.

1.2 Shared Virtual Memory

Due to cost considerations, all computer systems use physical memories of only limited size. However, in order to allow application programmers to be not constrained by physical memory limitations, most existing operating systems provide them a virtual memory, which is much larger than the actual physical memory. Therefore, a processor references any data element by using its virtual address in the address field of the instruction. The maximum size of virtual memory is limited only by the number of bits in this address field and the *swap* area of the disk which stores data that cannot fit in main memory. Virtual memory systems use either *paging*, where the virtual memory is divided into units of equal size called *pages*, or *segmentation*, where the virtual memory is divided into units of unequal size called *segments*, or a combination of both [13]. Since most current multiprocessors use paging, we restrict our discussion to paging.

In any virtual memory system, a map from virtual memory to physical memory is maintained as shown in Figure 1.2. For example, in a paged virtual memory system, a *page table* stores the mappings from virtual pages to physical pages. The page table can be an *indexed* page table as shown in Figure 1.3, or an *inverted* page table in which case the mappings from physical to virtual pages are maintained. We restrict our discussion to the commonly prevalent indexed page table. A large virtual memory implies a large number of virtual pages and consequently, a large number of mappings to be stored in the page table. Therefore, the page table might not fit in a single physical page, in which case a multi-level page table is necessary. All physical pages belonging to a page table are referred to as *page directories*. The *page table base register* stores the pointer to the first page directory page. During the virtual-to-physical address translation, the virtual address is split into various fields that provide the index into the different physical pages of the page table (Figure 1.3). In practice, the entire page table does not reside in physical memory. Rather, pages belonging to the page table are brought into physical memory on demand.

Consulting a multi-level page table for each instruction containing a memory ref-

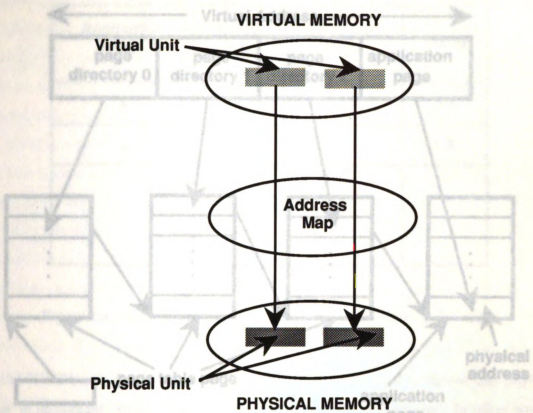

Figure 1.2. Mapping from virtual memory to physical memory.

erence results in performance degradation. To alleviate the above problem, virtual memory systems usually use a fast associative memory called the *translation lookaside buffer* (TLB) that has a subset of the mappings in the page table. During address translation, the processor first consults the TLB and if the mapping corresponding to the virtual page is not in the TLB, then it consults the page table. The TLB can be organized to allow mappings of multiple applications as shown in Figure 1.4. If it contains the mappings of only a single application, then there is no need for the field containing the application's ID. In this case, the TLB needs to be flushed every time a processor switches from executing one application to another. Each entry in the TLB and the page table also contains flags such as *dirty*, which indicates whether the page has been updated, *reference*, which indicates whether the page has been referenced,

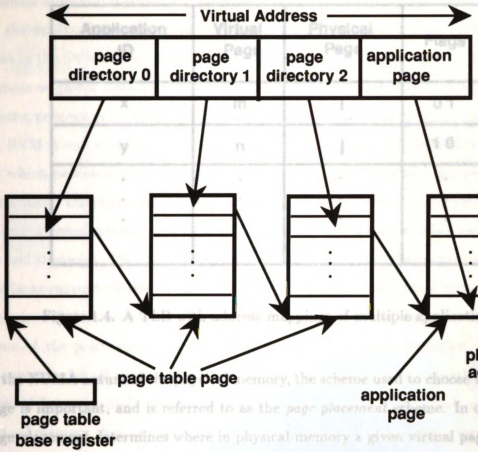

Figure 1.3. Virtual-to-physical address translation using a three-level page table.

and *protect*, which indicates the page's protection level. Due to cost considerations, the TLB has a limited number of entries, and therefore, *TLB replacement* policies are used to determine which TLB entry to replace in order to make room for a new mapping.

Usually, the number of physical pages is much smaller than the number of virtual pages. Therefore, the same physical page is allocated to different virtual pages at various times and consequently, the mappings in the TLB and the page table also change with time. A *page fault* is said to occur if a processor references a virtual address belonging to a virtual page which is not mapped to any physical page. The fault is resolved by mapping this virtual page to one of the free physical pages. Due

Application ID	Virtual Page	Physical Page	Flags
x	m	i	0 1
y	n	j	1 0
.	.	.	.
.	.	.	.
.	.	.	.

Figure 1.4. A TLB with address mappings of multiple applications.

to the NUMA nature of the physical memory, the scheme used to choose this physical page is important, and is referred to as the *page placement* scheme. In other words, page placement determines where in physical memory a given virtual page is placed. If no free pages are available, one of the already allocated physical pages is reclaimed, and the scheme used to make this choice is called the *page replacement scheme*. In other words, page replacement determines which virtual page will be replaced in memory by the currently referenced virtual page. It follows that the page replacement determines page placement when there are no free physical pages. Once the page fault is resolved, the TLB and the page table are updated to reflect the resulting change in the address map.

Specific virtual addresses are assigned to the data and code of an application executing on a uniprocessor that provides virtual memory. Extending this concept to a parallel application executing on a multiprocessor, all data and code, including those shared among processors, are assigned the same virtual address in all processors. Such an assignment allows the application to be written in the shared-variable programming model, in which processors synchronize and communicate by means of shared data mapped in a shared virtual memory or SVM. Therefore, the SVM offers the

benefits of virtual memory and the ease of programming in the shared-variable model to the applications programmer. Further, placing the operating system's code and data in the SVM allows better memory utilization by enabling pages to be mapped to remote or global memory instead of disk. It also facilitates easy distribution of work among processors by means of one or more shared queues of ready-to-run processes.

SVM is one example of a *process-interaction paradigm* which defines the method by which processes of a parallel application can synchronize and communicate with each other. Other process-interaction paradigms are the *remote procedure call* (RPC), object-oriented systems, and data-unit [14]. Each of these approaches have their merits and demerits. The RPC mechanism can be used among heterogeneous machines but lacks support for shared data and cannot be extended easily into the asynchronous domain. Object-oriented systems provide application-level features, are free from some of the problems with SVM which we discuss in a later section, but are not appropriate for all applications. The data-unit approach is a combination of features derived from the SVM and the object-oriented systems. A data unit is a region of virtual memory that can be mapped into the address map of various processors. As in object-oriented systems, application-level operations on data units are supported which enable synchronization and communication among processes. In this thesis, we consider the SVM process-interaction paradigm and develop methods to efficiently provide such a paradigm in NUMA multiprocessors.

SVM implementations exist for NUMA multiprocessors in which either all processors can directly address all available memory [15, 16, 17, 18], or processors cannot directly address remote memory [19, 20, 21, 22, 23, 24]. The implementation issues in the two cases are different as we discuss below. When providing a SVM in a NUMA multiprocessor with local and/or global memory, any virtual address (either in the kernel or in the user virtual memory) can be mapped to a physical memory location in any of these memories. Let us first consider the case when each memory location has a unique physical address and all locations can be directly addressed in hardware by all processors. Each processor obtains the physical address corresponding to the virtual address in a memory reference from its TLB or page table. It directly issues a

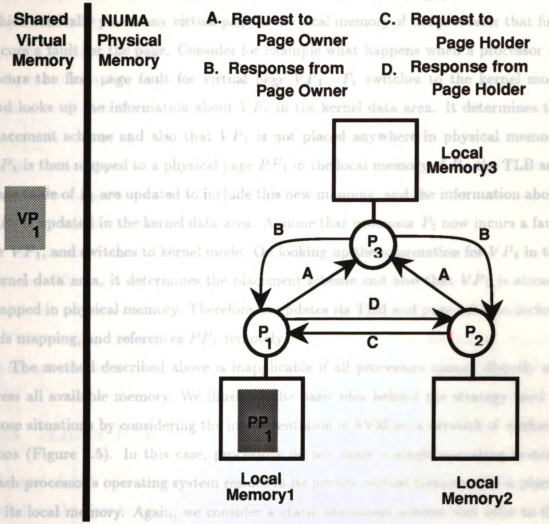

Figure 1.5. Implementation of a shared virtual memory on networked workstations.

memory request for this physical address for the read or write operation. In this case, all processors can share a single operating system which resides in the kernel portion of the shared virtual memory. The data area of this portion includes information about each virtual page such as whether it is placed in physical memory or not, and the processors sharing it. The code area of this portion includes various operating system services such as that provided by the page fault handler. We now illustrate how a page fault is handled in this case.

For simplicity, let us assume that all processors share a single physical copy of the

kernel virtual memory. For ease of explanation, we consider a page placement scheme which statically places any virtual page in the local memory of the processor that first incurs a fault for the page. Consider for example what happens when a processor P_1 incurs the first page fault for virtual page VP_1. P_1 switches to the kernel mode and looks up the information about VP_1 in the kernel data area. It determines the placement scheme and also that VP_1 is not placed anywhere in physical memory. VP_1 is then mapped to a physical page PP_1 in the local memory of P_1, the TLB and page table of P_1 are updated to include this new mapping, and the information about VP_1 is updated in the kernel data area. Assume that processor P_2 now incurs a fault for VP_1, and switches to kernel mode. On looking up the information for VP_1 in the kernel data area, it determines the placement scheme and also that VP_1 is already mapped in physical memory. Therefore, it updates its TLB and page table to include this mapping, and references PP_1 remotely.

The method described above is inapplicable if all processors cannot directly address all available memory. We illustrate the basic idea behind the strategy used in those situations by considering the implementation of SVM on a network of workstations (Figure 1.5). In this case, processors do not share a single operating system. Each processor's operating system resides in its private virtual memory and is placed in its local memory. Again, we consider a static placement scheme and refer to the processor in whose local memory a given virtual page is placed as the virtual page's *page holder*. Further, each virtual page is assigned a static *owner processor* that stores information about the page in the data area of its operating system. In order to get information about a given virtual page, processors contact the page's owner. *Page ownership* schemes deal with issues such as determination of a virtual page's owner, and efficient storage and dissemination of information about virtual pages. For ease of explanation, we assume that the information about owners of all virtual pages is either stored or can be easily computed by all processors.

Let us consider the actions in this case for the page fault sequence mentioned earlier. Assume that virtual page VP_1 is owned by processor P_3. When P_1 incurs a page fault for VP_1, it contacts P_3 to get information about VP_1. As in the previous

case, a physical page PP_1 is allocated in the local memory of P_1, and the TLB and page table of P_1 are updated. However, the virtual page information is updated by P_3, which records that P_1 is the page holder of VP_1. When P_2 incurs a fault for VP_1, it contacts P_3 and learns that P_1 is the page holder of VP_1. P_2 sends a memory request to P_1, which services the request and sends a response. The TLB and page table of P_2 are not updated and any further references to VP_1 by P_2 are handled in a similar fashion because P_2 cannot address the local memory of P_1 directly. Therefore, requests to a page in remote memory are serviced using messages after obtaining the page's holder information from its owner.

We discussed the two methods of implementing a SVM for a static page placement scheme. When this strategy is applied to a shared page, the references to the page by all but one processor require non-local memory accesses, which are expensive in a NUMA multiprocessor. This problem is addressed by data placement strategies which are covered in the next section.

1.3 Data Placement

The goal of data placement strategies is to place data in the NUMA physical memory such that the time to access it is minimized. One well-known method to achieve this goal is *data replication*, which replicates data in the local memory of the referencing processor. However, with replication, it is necessary to guarantee some form of relationship between these multiple physical copies. This relationship is referred to as the *consistency model* and is used by the applications programmer to write applications that execute correctly. To enforce the consistency model, when a processor issues a write operation, either all copies are updated, referred to as the *write-update* (WU) policy, or all other copies are removed, referred to as the *write-invalidate* (WI) policy.

Most NUMA multiprocessors provide replication at the level of a *block* which is the unit of data transfer at a given level of the memory hierarchy. Block-level replication is shown in Figure 1.6, where a virtual block containing the referenced data element is replicated in local memory. For example, NUMA multiprocessors with hardware cache

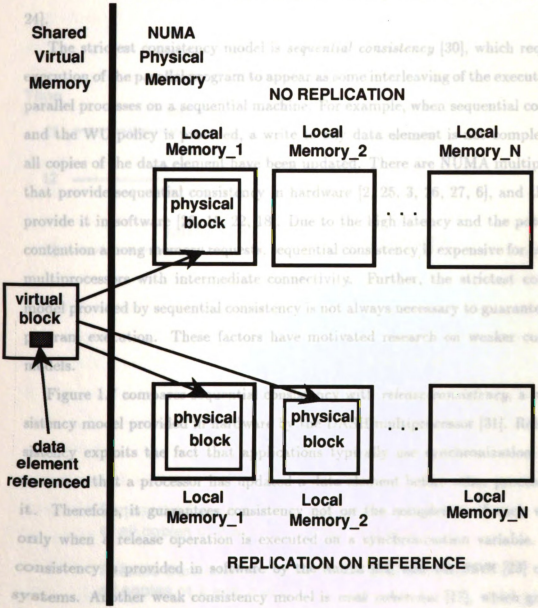

Figure 1.6. Block-level data replication.

consistency replicate the virtual cache line containing the referenced data element and keep the various physical cache lines consistent [2, 25, 3, 26, 27, 7, 17]. Others provide software mechanisms that replicate virtual pages containing the referenced data element and keep the corresponding physical pages consistent [28, 15, 22, 29, 18, 24].

The strictest consistency model is *sequential consistency* [30], which requires the execution of the parallel program to appear as some interleaving of the execution of the parallel processes on a sequential machine. For example, when sequential consistency and the WU policy is provided, a write to any data element is not complete unless all copies of the data element have been updated. There are NUMA multiprocessors that provide sequential consistency in hardware [2, 25, 3, 26, 27, 6], and those that provide it in software [28, 15, 22, 18]. Due to the high latency and the potential for contention among memory requests, sequential consistency is expensive for large-scale multiprocessors with intermediate connectivity. Further, the strictest consistency model provided by sequential consistency is not always necessary to guarantee correct program execution. These factors have motivated research on weaker consistency models.

Figure 1.7 compares sequential consistency with *release consistency*, a weak consistency model provided in hardware by the DASH multiprocessor [31]. Release consistency exploits the fact that applications typically use synchronization variables to ensure that a processor has updated a data element before other processors read it. Therefore, it guarantees consistency not on the completion of each write but only when a release operation is executed on a synchronization variable. Release consistency is provided in software by the `Munin` [24] and the `SSVM` [23] operating systems. Another weak consistency model is *weak coherence* [17], which guarantees that all copies of a data element are updated in the same order. Other weak consistency models proposed in the literature include [32, 33]. Weak consistency models offer performance improvement at the cost of either additional programmer effort in writing a parallel application or extra compiler effort in automatically parallelizing a sequential application. The approach of improving performance by transferring

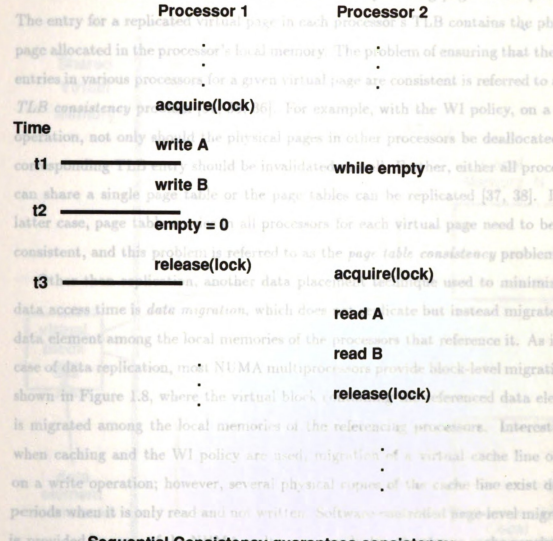

Figure 1.7. Consistency models.

data only during synchronization operations has also been used in `Clouds` [21], an object-oriented shared-virtual-memory implementation on networked workstations.

Certain other issues need to be addressed when providing page-level replication. The entry for a replicated virtual page in each processor's TLB contains the physical page allocated in the processor's local memory. The problem of ensuring that the TLB entries in various processors for a given virtual page are consistent is referred to as the *TLB consistency* problem [34, 35, 36]. For example, with the WI policy, on a write operation, not only should the physical pages in other processors be deallocated, the corresponding TLB entry should be invalidated as well. Further, either all processors can share a single page table or the page tables can be replicated [37, 38]. In the latter case, page table entries in all processors for each virtual page need to be kept consistent, and this problem is referred to as the *page table consistency* problem [15].

Other than replication, another data placement technique used to minimize the data access time is *data migration*, which does not replicate but instead migrates the data element among the local memories of the processors that reference it. As in the case of data replication, most NUMA multiprocessors provide block-level migration as shown in Figure 1.8, where the virtual block containing the referenced data element is migrated among the local memories of the referencing processors. Interestingly, when caching and the WI policy are used, migration of a virtual cache line occurs on a write operation; however, several physical copies of the cache line exist during periods when it is only read and not written. Software-controlled page-level migration is provided by some of the NUMA multiprocessors lacking hardware cache consistency [39, 16, 19, 40, 18].

Next, we discuss certain issues in implementing the data placement schemes. The information about each cache line which is required by hardware-controlled cache-line-level placement strategies is maintained in either a centralized or a distributed manner in hardware. Software-controlled page-level placement strategies also need to maintain information about each virtual page and make it efficiently available to all processors. For example, to maintain consistency of a page that is replicated, it is necessary to know which processors have its copy in their local memory. Similarly,

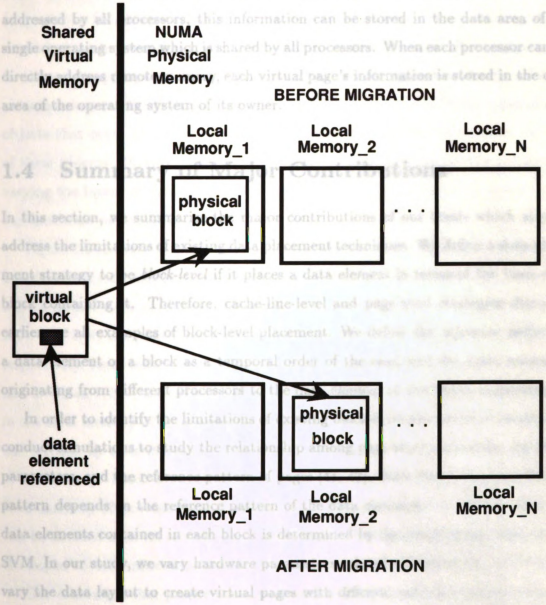

Figure 1.8. Block-level data migration.

for page migration, information about where the virtual page is currently placed in physical memory is needed. More sophisticated page placement strategies need additional information such as the history of read and write references to the various pages by different processors. When all available physical memory can be directly addressed by all processors, this information can be stored in the data area of the single operating system which is shared by all processors. When each processor cannot directly address remote memory, each virtual page's information is stored in the data area of the operating system of its owner.

1.4 Summary of Major Contributions

In this section, we summarize the major contributions of our thesis which aims to address the limitations of existing data placement techniques. We define a data placement strategy to be *block-level* if it places a data element in terms of the basic data block containing it. Therefore, cache-line-level and page-level strategies discussed earlier are all examples of block-level placement. We define the *reference pattern* of a data element or a block as a temporal order of the read and the write references originating from different processors to the data element or the block, respectively.

In order to identify the limitations of existing block-level placement strategies, we conduct simulations to study the relationship among page-level replication, hardware parameters, and the reference pattern of pages [41, 42]. Note that a block's reference pattern depends on the reference pattern of the data elements it contains. Also, the data elements contained in each block is determined by the layout of the data in the SVM. In our study, we vary hardware parameters related to replication, and we also vary the data layout to create virtual pages with different reference patterns for the same application. We conclude that the effectiveness of replicating a page depends on its reference pattern and also on hardware parameters. In addition, we found out that with proper data layout, the amount of sharing of the application's pages is reduced, and then the performance of the application with and without replication is almost the same because the two schemes are equivalent for a non-shared page. Hence, we

conclude that proper data layout can simplify data placement. These results apply to other block-level strategies as well.

In related work, Bolosky et al. [43] conclude that page-level migration strategies need to be dependent on hardware parameters related to migration. Other studies (surveyed in [44, 45]) emphasize the need for a block's placement strategy to be dependent on its reference pattern. Previous studies have also made it straightforward to choose an appropriate block-level placement strategy once the block's reference pattern is known. For example, Bennett et al. [29, 46] identify different types of data objects that occur in typical applications, and propose placement schemes for each of these objects. However, our study is unique in that we consider the influence of varying the layout of data in the SVM and also the relationship between replication and hardware parameters.

Figure 1.9 shows examples of how to choose placement strategies for virtual blocks based on their reference pattern. Here, a virtual block is denoted by $b(p_j, p_k, ...)$, where b is its ID and each p_j denotes a processor j that references it. Private blocks 1(1), 2(2), and 3(N) are placed statically in the local memory of processors 1, 2, and N, respectively. Read-only block 4(1,2) is replicated in the local memory of processors 1 and 2. Block 7(2,N) is read more often than it is written, and is therefore replicated in the local memory of processors 2 and N. Blocks 6(1,2) and 8(1,2) are actively read and written by several processors, and are not replicated due to the high overhead of consistency. Block 6 is placed in global memory, while block 8 is placed in the local memory of processor 2. Therefore, the ideal placement strategy for a block can be decided once its reference pattern is known. However, techniques need to be developed to determine each block's reference pattern.

Our conclusion that proper data layout can simplify data placement is related to the problem of *false sharing* [47], which occurs when a virtual block contains data elements with different reference patterns. Data placement strategies are limited in their ability to place a falsely-shared block. For example, assume that block 7(2,N) in Figure 1.9 contains two data elements each of which is exclusively referenced by processors 2 and N, respectively. Replicating such a block leads to unnecessary consis-

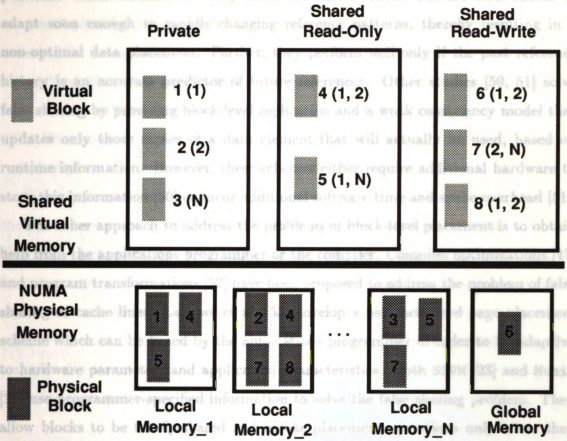

Figure 1.9. Block-level placement in a shared-virtual-memory NUMA multiprocessor.

tency overhead, migrating it leads to a ping-pong effect, and placing it statically in a given processor's local memory results in non-local memory accesses for the other processor. Since the probability of false sharing is higher for larger blocks [48, 49, 50], the resulting performance loss is severe for NUMA multiprocessors providing page-level placement.

It is clear that to improve the performance of block-level placement, false sharing needs to be eliminated and the placement strategy should be tailored to the reference pattern of blocks. One approach is to address these issues without any help from the compiler or the applications programmer. For example, certain block-level placement strategies [15, 16] use the runtime reference history to adapt to changing reference

patterns. Such schemes not only incur runtime overhead but are also unable to adapt soon enough to rapidly-changing reference patterns, thereby resulting in a non-optimal data placement. Further, they perform well only if the past reference history is an accurate predictor of future references. Other studies [50, 51] solve false sharing by providing block-level replication and a weak consistency model that updates only those copies of a data element that will actually be used, based on runtime information. However, these schemes either require additional hardware to store this information [50] or incur additional software time and space overhead [51].

The other approach to address the problems of block-level placement is to obtain help from the applications programmer or the compiler. Compiler optimizations [47] and program transformations [52] have been proposed to address the problem of false sharing of cache lines. LaRowe et al. [53] develop a parameterized page placement scheme which can be tuned by the applications programmer in order to be adaptive to hardware parameters and application characteristics. Both `SSVM` [23] and `Munin` [24] use programmer-specified information to solve the false sharing problem. They allow blocks to be falsely-shared and invoke placement operations only when they are actually required, as determined by the programmer-specified information. In addition, `SSVM` allows the programmer to place data in terms of objects rather than blocks, which we refer to as *object-level placement.*

Our approach to address the problems of block-level placement is by developing new techniques that assist data placement which can be incorporated in a compiler. These techniques reduce the runtime overhead and they also minimize the programmer effort involved in proper data placement. Our method for compiler-assisted data placement uses compile-time objects containing data with the same variable type and similar reference patterns. We design a compiler that uses our method, and we develop algorithms to determine the type and reference pattern of data. We also develop specific object-creation schemes for different types of data placement strategies viz., the block-level and object-level types of placement.

Our object-creation scheme that assists block-level placement [54] creates objects that are not falsely-shared and that have temporal locality. We propose methods to

implement these objects which ensure that blocks are not falsely-shared. Further, we use these objects to provide information required for proper placement, thereby reducing the runtime overhead of collecting this information. Our scheme that directs object-level placement creates objects based on the data exchange pattern [55, 56] and automatically generates all the programmer-specified information required by SSVM. We evaluate the cost of applying our schemes by deriving their time and space complexity.

We use experimental simulations to compare the performance of applications for the block-level placement (with and without compiler assistance) and the compiler-directed object-level placement strategies. We conclude that compiler-specified information about objects leads to a significantly better performance than that in the absence of such information. Further, the best performance is achieved when placement is directed rather than assisted by the compiler. Our work demonstrates that compiler-directed data placement is a promising approach to improve the performance of applications and the programmability of NUMA multiprocessors.

1.5 Thesis Organization

The rest of our thesis is organized as follows. In Chapter 2, we present the results of our study on factors affecting data placement schemes. In Chapter 3, we outline our approach for compiler-assisted data placement. In Chapter 4, we discuss our work on compiler-assisted block-level placement. In Chapter 5, we present our work on compiler-directed object-level placement. In Chapter 6, we outline the results of our experimental evaluation of various types of placement schemes. Finally, in Chapter 7, we summarize the contributions of our thesis and present directions for future research.

CHAPTER 2

FACTORS AFFECTING DATA PLACEMENT

The primary aim of this thesis is to develop efficient data placement techniques, which minimize the data access time, when providing a shared virtual memory in a NUMA multiprocessor. As a first step toward achieving this aim, we conduct a study [41, 42] to identify the limitations of existing data placement strategies, most of which provide block-level placement. We present the results of our study in this chapter, which is organized as follows. First, we provide a detailed overview of related work, which was done prior to our study and which motivates the goals of our work. Next, we outline our assumptions about the hardware features of the NUMA multiprocessor and the manner in which the shared virtual memory (SVM) is provided. After presenting our work on page replacement, we outline the methodology, performance metrics, and workload of our study. Next, we discuss our experiments in detail, and present our conclusions.

2.1 Related Work

The issues of providing a SVM have been studied for both NUMA multiprocessors in which all processors can directly address all available memory and those in which each processor cannot directly address another processor's local memory. Li [28] was the first to propose the idea of providing a SVM for a NUMA multiprocessor belonging to

the latter category. SVM is also provided by the Mach operating system [57], which has been ported to a wide variety of paged uniprocessors and multiprocessors. Studies about SVM which were done prior to our study considered block-level data placement using either replication or migration. Further, in the case of block-level replication, studies differed in the consistency model used and whether the write-invalidate (WI) or the write-update (WU) scheme was used to maintain consistency.

Data Replication

First, we discuss studies that considered block-level data replication and maintained sequential consistency using the WI scheme. IVY [28, 20] is a user-level SVM, which is implemented for a network of Apollo workstations. It uses several page ownership schemes such as the single-owner, statically-determined multiple-owner, and dynamically-determined multiple-owner schemes. The port of Mach to workstations interconnected by a token-ring [58] accommodates multiple page sizes and heterogeneous architectures. It also uses the single-owner and multiple-owner page ownership schemes, with fault tolerance being implemented for the former. Mirage [22] is a kernel-level segmented-page SVM, implemented for three VAX 11/750's networked by an Ethernet. It uses a lazy sequential consistency model which introduces a delay before a write request to a segment shared by multiple readers is granted. PLATINUM [15] is a SVM implemented for a BBN Butterfly Plus. It selectively replicates pages depending on the number of times the WI scheme is invoked for each page. Further, it uses a directory-based page ownership scheme, and replicates page tables and keeps them consistent by interrupting processors that are actively using the updated page table entry. DUnX [59, 18] is a SVM implementation for BBN's GP-1000 [60] and provides several page-level replication schemes. While these studies considered sequential consistency, others provided a weaker consistency model instead. For example, Bisiani et al. [61] used trace-driven simulations to study their weak coherence model maintained using the WU scheme. They concluded that full replication with a large number of processors is expensive and advocated selective replication instead. Weak coherence is provided in hardware by the PLUS multiprocessor [17] which also

supports various hardware synchronization operations that enable applications to be written for this consistency model.

Data Migration

Next, we discuss studies which considered block-level data migration instead of replication. In the port of Mach on the ACE [16], a page is migrated to the local memory of the processor that references it; after a selective number of such migrations, the page is placed statically in global memory. Further, kernel pages are neither replicated nor migrated and are placed in global memory. Black et al. [39] used trace-driven simulations to study competitive page-level migration algorithms, which require hardware reference counters for implementation. Page-level migration is also provided in the implementation of a SVM for an iPSC/2 by Li and Schaefer [19]. Mizrahi et al. [62] used trace-driven simulations to study several block-level migration strategies that extend the memory hierarchy into the interconnect. Scheurich and Dubois [40] study page-level migration using page pivoting in a point-to-point mesh interconnect. DUnX [59, 18] also provides various page-level migration schemes. Different strategies to distribute the read-only pages of an application among the local memories of various processors have been studied in [63].

Other problems that need to be solved when providing a SVM in a NUMA multiprocessor have also been studied in the literature. For example, the TLB and the page table consistency problem has been studied in [34, 35, 36, 15]. Holland [37] studied three page table management schemes using software simulation of synthetic applications on a BBN Butterfly Plus. The study concludes that a single page table is a software bottleneck and a fully replicated indexed page table uses a lot of memory. Therefore, the best choice is a partially replicated indexed page table which not only uses less memory but also performs almost as well as the fully replicated indexed page table. In a related paper [38], the most-recently-used page replacement scheme is considered, and the influence of the number of replacement daemons and their frequency of invocation, on typical applications, is studied.

Motivation and Goals of Study

Most of the previous work involves actual implementation, and performance is measured in terms of speedup or execution time. Such a performance measure applies to the hardware parameters of the specific multiprocessor in question. These studies do not compare the performance of a given placement strategy on different multiprocessors. Such a comparison is one of the goals of our study, and we realize this goal by characterizing each multiprocessor by hardware parameters that are related to the placement strategy in question. We also study page replacement schemes which have not been addressed in detail by previous studies. Further, some of the block-level placement strategies proposed in previous studies are designed to be adaptive to the block's reference pattern. For example, `PLATINUM` [15] replicates pages depending on their invalidation frequency, and the port of `Mach` on the ACE [16] migrates pages only a certain number of times. Also, Bennett et al. [29, 46] propose data placement schemes that adapt to data objects of different types. Since a block's reference pattern or a data object's type is determined by the layout of the application's data in the SVM, another goal of our study is to consider the interaction between the data layout and the data placement strategy. We realize these goals by using a combination of analytical and trace-driven simulation techniques, which allow great flexibility in parameter variation.

2.2 Assumptions

In this section, we outline our assumptions about the hardware features of the NUMA multiprocessor and also the manner in which the SVM is provided. We consider NUMA multiprocessors such as the BBN TC2000 which lack hardware cache consistency and in which main memory is organized as modules local to each processor. Further, all processors can directly address all available memory. We assume that all processors share a single operating system which is allocated in the kernel virtual memory. The code area of the operating system is replicated in local memory on reference, but there is a single copy of the data area. For example, all processors share

a single copy of the data structure that stores information about the various virtual pages. We do not replicate the operating system's data because it is frequently updated by all processors, and the overhead of enforcing consistency might nullify the benefits of replication. Instead, we distribute this data evenly across all available local memory modules.

The data placement strategy we consider is software-controlled, page-level replication with sequential consistency maintained using either the WU or the WI scheme. We refer to the WU scheme alternatively as *multicast*, as its effect is the same as that of a multicast communication. We define *full replication* (FR) as the strategy that replicates a virtual page in the local memory of each processor that references it, and *no replication* (NR) as the case when virtual pages are not replicated. Further, we refer to the first physical page allocated to a given virtual page as the *master*, and each physical page allocated thereafter as a *replica*. Note that if the master is in remote memory relative to a processor issuing a write request, then both the master and the local copy need to be updated. We refer to such an update of the master copy as a *remote master write*. We do not use explicit page-level migration, but as discussed in the next section, pages are migrated during specific cases of page replacement.

We use two schemes for initial page placement: (1) *fault processor* placement, where the virtual page is placed in the faulting processor's local memory, and (2) *modulo* placement, where the virtual page is placed in the local memory of the processor given by: (virtual page number) *mod* (number of processors). Page replacement has not been given much attention in existing studies, and it is important when using the FR scheme which needs more memory, and therefore, we study it in detail and postpone discussion on it until the next section.

To enable fast address translation, each processor has its own TLB and we use a single-hand clock scheme[64] for TLB replacement. Further, each processor has its own copy of an indexed page table, the pages of which remain allocated during the application's lifetime. In the FR case, each processor has its local copy of any virtual page, and the page's entry in the processor's TLB and page table also pertains to this local copy, and hence, TLB and page table consistency is not an issue. On the other

hand, in the NR case, all processors referencing a virtual page share its only physical page, and the page's entry in the TLB and the page table of all these processors pertain to this single physical page. The information in the TLB and page table that are modified and hence related to the consistency problem are the dirty and the reference flags. Since our page replacement scheme does not use the reference flag, we consider only the dirty flag which is used to decide whether a page, on being chosen to be replaced, needs to be written to the disk. We solve the consistency problem in the NR case by updating the dirty flag as follows. When a virtual page is written, the corresponding dirty flag is set in the TLB and the page table of only the processor issuing the write request. When a page is chosen to be replaced, it is written to the disk if the dirty flag is set in the TLB or the page table of any of the processors that referenced it.

We conclude this section by defining terms to denote some of the types of memory references that can occur during an application's execution. During the virtual-to-physical address translation, a processor might find the corresponding mapping in its TLB, referred to as the *TLB hit*, or it does not find the mapping in its TLB, but finds it in its page table, referred to as a *page table hit*, or it does not find it in both its TLB and page table, referred to as a *fault*. We refer to a fault on a virtual page which is mapped in remote memory as a *remote page fault* (RPF). We assume that any physical page returned to the list of free pages is marked as a disk cache page, and refer to a fault on such a page as a *disk cache fault* (DCF). Such a disk caching scheme allows the page to be reused and prevents it from being copied unnecessarily from the disk. We refer to a fault on a virtual page which is not mapped anywhere in physical memory as a *disk page fault* (DPF).

2.3 Page Replacement

Efficient page replacement schemes are important, when providing page-level replication, because more physical pages are used. This fact is particularly true for applications with a high degree of data sharing. In this section, we discuss our work on

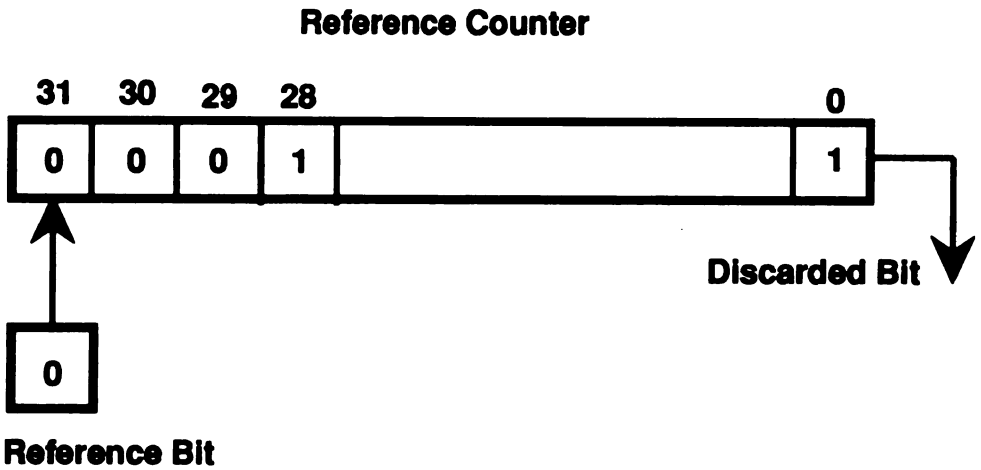

Figure 2.1. The software LRU approximation scheme.

page replacement.

2.3.1 CLOCK Scheme

The first scheme we study is the adaptation of the CLOCK algorithm used in the 4.3 BSD UNIX [64]. The CLOCK algorithm periodically resets each page's reference flag, thereby making it available for replacement. We conclude that it is preferred over the LRU algorithm [13] in the 4.3 BSD UNIX, mainly due to the hardware constraints of the VAX architecture rather than due to its superior performance. Since the CLOCK algorithm does not use the reference information for a page over a period of time, we do not study it further.

2.3.2 Software LRU Approximation Scheme

The next scheme we study is the adaptation of the software LRU approximation algorithm. The basic algorithm periodically shifts the reference bit of each physical page into the most significant bit of the page's reference counter, whose least significant bit is discarded (Figure 2.1). The page replacement scheme chooses the page with the least value of the reference counter. We adapt this algorithm for page replacement

with a NUMA physical memory as follows. We maintain separate lists for the master and the replica pages, and for each page in these lists, we periodically update its reference counter and reset its reference bit. We also maintain a set of reference counters, one for each virtual page, and update these counters periodically using either the hardware reference bits or the reference counters of the corresponding master and replica physical pages. We allow all the update and the reset intervals to be variable design parameters. Therefore, the reference information for a virtual page is a combination of its local and remote reference information. In contrast, the adaptation of software LRU approximation in [59, 18] uses only local reference information. Software LRU approximation has the following drawbacks: (1) it is expensive because the reference counters need to be implemented in hardware for performance reasons, (2) the overhead of periodically updating all the counters and bits is high, and (3) in our experience, simulating its operation takes a lot of time.

2.3.3 LRU Scheme

Next, we study the adaptation of the LRU algorithm, which is known to perform the best for applications that exhibit locality. In the basic LRU algorithm, when a page is referenced, it is placed at the beginning of the list of active pages. The page replacement scheme replaces the page at the end of the list, which is the one least recently used. Clearly, implementing this algorithm in software is too slow to be practical, while a hardware implementation is too costly. However, even software LRU approximation needs a hardware implementation for performance reasons. Further, any approximation to LRU can at best perform as good as LRU, and therefore, LRU gives a best-case performance estimate. In our experience, it takes less time to simulate its operation than that of the software LRU approximation. Based on these considerations, in our study, we develop a new page replacement scheme based on LRU called **LERN**, an acronym for **L**RU **E**xtension for **R**eplicated **N**UMA memory management.

We now describe how **LERN** works in the FR and the NR cases. The state of any physical page is defined to be *free*, when it is not allocated to any virtual page, and is

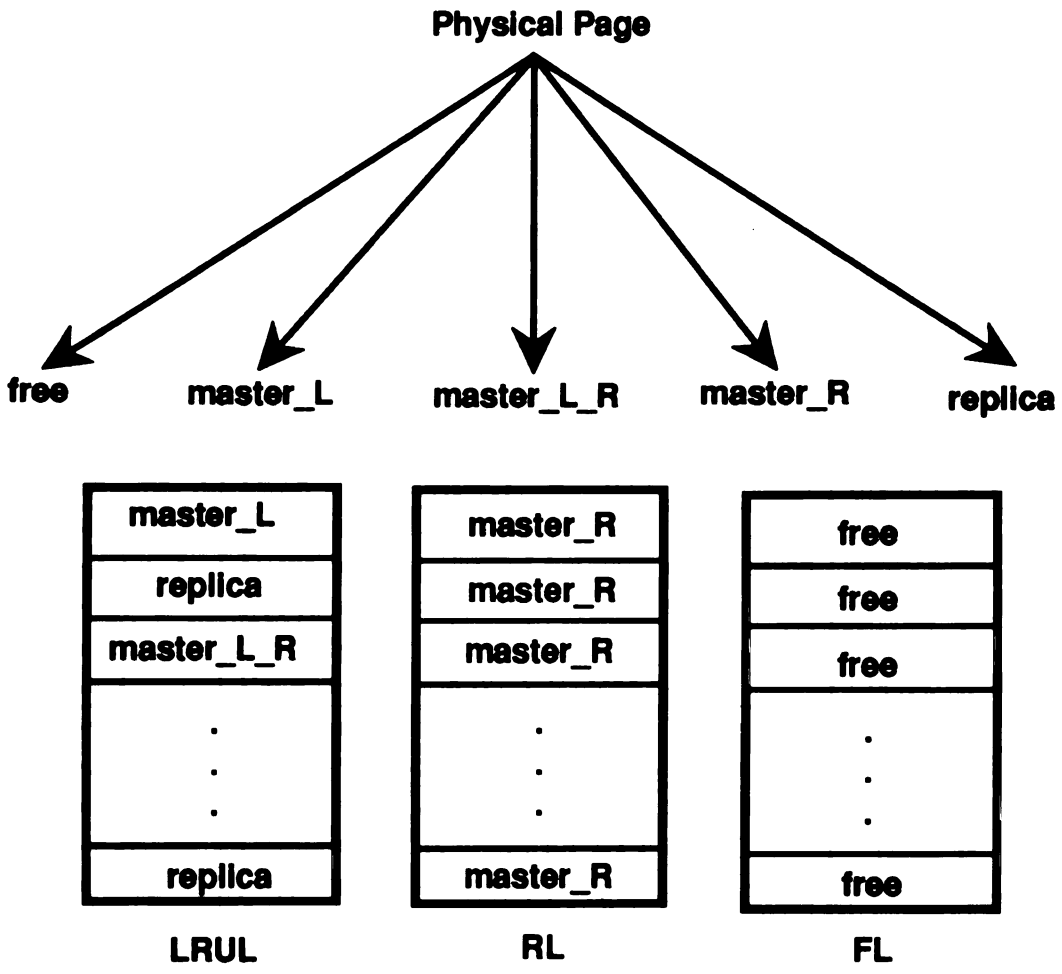

Figure 2.2. The **LERN** page replacement scheme.

defined to be *replica*, when it is allocated for page-level replication and is used locally. Further, when a physical page is the master copy, its state is defined to be *master_L*, *master_R*, or *master_L_R*, depending on whether it is used locally, remotely, or in both places, respectively. Each processor maintains a *free list* (FL) that contains free pages, a *remote list* (RL) that contains master_R pages, and a *LRU list* (LRUL) that contains master_L, master_L_R, and replica pages (Figure 2.2). Note that in the case of NR, LRUL has no replica pages. Tables 2.1 and 2.2 show the state transition diagram of a physical page when the **LERN** scheme is used for the FR and the NR cases, respectively.

Table 2.1. Page state transition diagram: **LERN** with full replication.

Current State	Event	Next State
free	Page replication	replica
free	DPF	master_L
free	DPF	master_R
replica	Periodic replacement	free
replica	Write invalidation	free
replica	Demand replacement for page replication	replica (New)
replica	Swap with master_R	master_L
replica	Demand replacement for DPF	master_L
replica	Demand replacement for DPF	master_R
replica	Swap with master_R	master_L_R
master_L	Periodic replacement	free
master_L	Demand replacement for page replication	replica
master_L	Demand replacement for DPF	master_L (New)
master_L	Demand replacement for DPF	master_R
master_L	Page replication on a remote node	master_L_R
master_R	Periodic replacement	free
master_R	No more replicas of the virtual page	free
master_R	Demand replacement for page replication	replica
master_R	Demand replacement for DPF	master_L
master_R	Demand replacement for DPF	master_R (New)
master_R	Page replication	master_R
master_R	Local page fault	master_L_R
master_L_R	No more replicas	master_L
master_L_R	No local reference	master_R
master_L_R	Page replication	master_L_R

LERN for Full Replication

In the FR case, at the end of a reference to a virtual page by a processor, a corresponding physical page is found in the processor's LRUL. Further, each processor reorders its LRUL on every reference. When **LERN** is invoked, a page can be replaced either from the LRUL or from the RL. A page is replaced from the LRUL if it contains more than a minimum number of pages, which is equivalent to the LRU stack size. Otherwise, a page is replaced from the RL. If it is decided to replace a page from the LRUL, master_L_R pages are skipped as long as the number of replica and master_L

Table 2.2. Page state transition diagram: **LERN** with no replication.

Current State	Event	Next State
free	DPF	master_L
free	DPF	master_R
master_L	Periodic replacement	free
master_L	Demand replacement for DPF	master_L (New)
master_L	Demand replacement for DPF	master_R
master_L	Remote reference	master_L_R
master_R	Periodic replacement	free
master_R	Demand replacement for DPF	master_L
master_R	Demand replacement for DPF	master_R (New)
master_R	Remote reference	master_R
master_R	Local page fault	master_L_R
master_L_R	Remote reference	master_L_R
master_L_R	No local reference	master_R

pages is above a minimum value. If a master_L_R page is chosen as the candidate from the LRUL, it is transferred to the RL after changing its state to master_R, and the replacement policy on the RL is invoked. Otherwise, a replica or a master_L page is replaced from the LRUL. The RL replacement scheme replaces the page that has the least number of replicas, and uses a first-in-first-out rule when there is a tie. The first replica of the replaced page is made the new master, and in this case, the virtual page is effectively migrated.

LERN for No Replication

In the case of NR, a virtual page referenced by any processor might be allocated either in local or in remote memory. While in the former case, the corresponding physical page is in the local LRUL, in the latter case, it is in the LRUL or the RL of another processor. To implement exact LRU, each processor, on every reference, needs to reorder the list (LRUL or RL) in which the page currently referenced is located. It is obvious that this reordering is costlier in the NR case than in the FR case. The other option is to reorder only in the case of local references and use a first-in-first-out

ordering for remote references. When **LERN** is invoked, a page can be replaced either from the LRUL or the RL. As in the FR case, a page is replaced from the LRUL, if it contains more than a minimum number of pages, and from the RL otherwise. In the former case, master_L_R pages are skipped as long as the number of master_L pages is above a minimum value. If a master_L_R page is chosen as the candidate from the LRUL, it is transferred to the RL after changing its state to master_R, and the RL replacement scheme is invoked. Otherwise, a master_L page is replaced from the LRUL. The RL replacement scheme replaces the page that is used by the least number of remote processors. The TLB and the page table entries corresponding to the replaced page are invalidated in all these processors.

2.4 Method of Study

The technique of simulation has been used extensively to study the memory performance of both uniprocessors and multiprocessors. A given run of a simulator approximates the execution of an application for a certain data placement scheme such as cache-line-level replication with sequential consistency maintained using the WI scheme. One of the key advantages of simulation is the ability to study the effect of a wide range of parameters at a cost significantly lower than actual implementation. Two well-known simulation techniques are trace-driven simulation and execution-driven simulation [65]. In the former approach, the application is modeled by a global execution order of all its memory references, referred to as its *address trace* or just trace. Techniques for collecting such traces as accurately as possible have been developed (e.g., [66]). A trace-driven simulator simulates the actions that occur in response to each reference in this global execution order, one reference at a time. It does not account for changes that can occur to this global execution order itself due to the placement strategies that are being simulated.

In the case of an execution-driven simulator, the application is modeled not as a global execution order, but as a set of execution orders, one per processor, each of which contains an event for every instruction executed by the corresponding processor.

A time counter is maintained for each processor and is initialized to zero. A given processor's time counter is incremented whenever an event from its execution order is processed. The counter is incremented by the amount of time taken for this event's execution, which depends on whether the event is a memory reference or not, and in the former case, the actions taken by the placement strategy. The simulator chooses its next event from the execution order of the processor that has the minimum value of all the time counters. When the values in the time counters of processors coincide, it makes an approximation by making a random choice. Such a simulation allows a given processor's time counter to be incremented by the time taken for actions of the placement strategy, which are in response to an event from this processor's execution order. However, other processors also participate in some of these actions such as invalidations that occur when the WI scheme is used to maintain consistency of replicated data. In order to incorporate the time for these actions in the time counters of the other processors, the time at which these actions took place, relative to each of these processors is needed, and only approximate values can be assumed for such cases. Therefore, an execution-driven simulator is only able to approximately incorporate the changes in the global execution order due to placement strategies.

It is clear that execution-driven simulation is not only time-consuming, but also is not an accurate simulation of an application's execution. Further, it requires separate execution orders for each processor, and additional parameters such as the time taken to execute each type of instruction. The additional accuracy offered by such a simulation over trace-driven simulation is at the cost of extra complexity, and is justified only if it is necessary to measure the execution time as accurately as possible. As discussed in the next section, we measure performance not by the execution time, but by a metric that needs only the number of memory references for each type of reference. Due to these considerations, we use trace-driven simulations in our study. Independently, Bolosky et al. [43] also use a trace-driven simulator to study the relationship between page-level migration and hardware parameters when providing a SVM in a NUMA multiprocessor. They measure performance by the mean cost per reference, which is similar to one of our measures of performance. Further, they compare re-

sults obtained for simulations on traces which are generated by arbitrarily perturbing a given global execution order. They conclude that ignoring the perturbation caused by the placement strategies does not influence the results obtained.

Our simulator is written in Gnu C++ and its input parameters include the TLB size, the number of processors, the number of physical pages, the page size, the number of address bits, and the parameters related to the **LERN** scheme and the various schemes for the other problems involved in providing a SVM. Its output includes various statistical data such as the number of page replacements and the number of memory references for each reference type, which can be measured at various points of the simulation run.

2.5 Performance Metrics

Since the goal of data placement is to minimize the data access time, our key measure of performance is the *effective memory access time* (EMAT), which is the mean memory access time for the application averaged over all its memory references. The access time for a given memory reference depends on the type of the reference and the placement strategy used. For example, if a reference causes a disk page fault, then its access time includes the time to transfer the page from disk to memory. If the reference causes a remote page fault and page-level replication is used, the access time includes the time to allocate a page in local memory and the time to initialize the allocated page from the master which is allocated in remote memory. The EMAT is given by:

$$\frac{\text{Data consistency time} + \sum_{\substack{\text{reference} \\ \text{type}}} \{\ \text{No. of references} \times \text{Access time}\}}{\text{Total no. of references}} \tag{2.1}$$

As mentioned in the previous section, our simulator records the number of references that occur in each reference type category. We derive the access times taken by various types of references in terms of parameters that characterize the hardware

Table 2.3. Parameters.

dfos	Disk fault kernel lookup
dmc	Disk to memory copy
inval	Write invalidation
lern	LERN scheme
local	Local memory access
multicast	Write update
ptl	Page table level
remote	Remote memory access
remwr	Remote master write
rlc	Remote to local copy
rpfos	Remote page fault kernel lookup

Table 2.4. TLB hit accesses.

Page location	Access time
Local	t_{local}
Remote	t_{remote}

and also those that represent the software overheads. These parameters are listed in Table 2.3, and for each parameter p, n_p represents the number of times it occurs and t_p represents the time it takes. The expressions for the access times are enumerated in Tables 2.4 through 2.8. Table 2.9 lists the time taken to maintain consistency of data for all the memory references, where $t_{multicast}$ is normalized to t_{remote}.

Our second performance measure is the *parameter overhead*, which is a measure of

Table 2.5. Page table hit accesses.

Page location	Access time
Local	$n_{ptl}t_{local} + t_{local}$
Remote	$n_{ptl}t_{local} + t_{remote}$

Table 2.6. Remote page fault accesses.

Replicate?	Master Locn.	Replica Locn.	LERN Replica?	Access time
Yes	Remote	Local	No	$n_{ptl}t_{local} + t_{rpfos} + t_{rlc} + t_{local}$
Yes	Remote	Local	Yes	$n_{ptl}t_{local} + t_{rpfos} + t_{rlc} + t_{local} + t_{lern}$
Yes	Local	Local	NA	$n_{ptl}t_{local} + t_{rpfos} + t_{local}$
No	Local	Local	NA	$n_{ptl}t_{local} + t_{rpfos} + t_{local}$
No	Remote	Remote	NA	$n_{ptl}t_{local} + t_{rpfos} + t_{remote}$

Table 2.7. Disk cache fault accesses.

Replicate?	Master Locn.	Replica Locn.	LERN Replica?	Access time
Yes	Remote	Local	No	$n_{ptl}t_{local} + t_{dfos} + t_{rlc} + t_{local}$
Yes	Remote	Local	Yes	$n_{ptl}t_{local} + t_{dfos} + t_{rlc} + t_{local} + t_{lern}$
Yes	Local	Local	NA	$n_{ptl}t_{local} + t_{dfos} + t_{local}$
No	Local	Local	NA	$n_{ptl}t_{local} + t_{dfos} + t_{local}$
No	Remote	Remote	NA	$n_{ptl}t_{local} + t_{dfos} + t_{remote}$

the influence of a particular parameter on the EMAT. By using Tables 2.4 through 2.9 in Eq. 2.1, the EMAT can be expressed as:

$$\frac{\sum_{\substack{\text{all} \\ \text{parameters}}} \left\{ \begin{array}{l} \text{No. of times the parameter} \\ \text{appears in the total access time} \end{array} \right\} \times \left\{ \begin{array}{l} \text{Time for} \\ \text{the parameter} \end{array} \right\}}{\text{Total no. of references}}$$

Therefore, the overhead of a particular parameter p is given by:

$$\frac{\left\{ \begin{array}{l} \text{No. of times } p \text{ appears} \\ \text{in the total access time} \end{array} \right\} \times \left\{ \text{Time for } p \right\}}{\text{EMAT} \times \text{Total no. of references}}$$

Table 2.8. Disk page fault accesses.

Repli-cate	Master Locn.	Replica Locn.	LERN Master	LERN Replica?	Access time
Yes	Remote	Local	No	No	$n_{ptl}t_{local} + t_{dfos} + t_{dmc} + t_{rlc} + t_{local}$
Yes	Remote	Local	No	Yes	$n_{ptl}t_{local} + t_{dfos} + t_{dmc} + t_{rlc} + t_{local} + t_{lern}$
Yes	Remote	Local	Yes	No	$n_{ptl}t_{local} + t_{dfos} + t_{dmc} + t_{rlc} + t_{local} + t_{lern}$
Yes	Remote	Local	Yes	Yes	$n_{ptl}t_{local} + t_{dfos} + t_{dmc} + t_{rlc} + t_{local} + 2(t_{lern})$
Yes	Local	Local	No	NA	$n_{ptl}t_{local} + t_{dfos} + t_{dmc} + t_{local}$
Yes	Local	Local	Yes	NA	$n_{ptl}t_{local} + t_{dfos} + t_{dmc} + t_{local} + t_{lern}$
No	Local	Local	No	NA	$n_{ptl}t_{local} + t_{dfos} + t_{dmc} + t_{local}$
No	Local	Local	Yes	NA	$n_{ptl}t_{local} + t_{dfos} + t_{dmc} + t_{local} + t_{lern}$
No	Remote	Remote	No	NA	$n_{ptl}t_{local} + t_{dfos} + t_{dmc} + t_{remote}$
No	Remote	Remote	Yes	NA	$n_{ptl}t_{local} + t_{dfos} + t_{dmc} + t_{remote} + t_{lern}$

Table 2.9. Time to maintain consistency of data.

Consistency scheme	Replicate?	Data consistency time
WI	Yes	$t_{inval}n_{inval} + t_{remote}n_{remwr}$
WI	No	0
WU	Yes	$t_{remote}t_{multicast}n_{multicast} + t_{remote}n_{remwr}$
WU	No	0

```
   DOSEQ S1
      Initialize A
S1 END DOSEQ
   DOSEQ S2 K = 1 TO m
      DOALL S3 I = 1 TO n
         DOALL S4 J = 1 TO n
           A(I,J) = 0.25 * (A(I-1,J) + A(I+1,J) + A(I,J-1) + A(I,J+1))
S4       END DOALL
S3    END DOALL
S2 END DOSEQ
   DOSEQ S5
      Output A
S5 END DOSEQ
```

Figure 2.3. Application **parallel iterative solver**.

2.6 Workload

The workload for our simulations consists of synthetic traces as well as address traces of actual applications. We input synthetic traces of well-known reference patterns to the simulator, and test the validity of the simulator by comparing its output to the expected output. One of the actual application address traces used in our study is collected by executing the Particle-In-Cell benchmark on an Alliant emulator for an eight-processor configuration [67], and we refer to it as application **PIC**. We also consider the class of applications which solve partial differential equations using iterative methods [68], and develop a program to generate the address trace for the case of a two-dimensional problem with a 5-point stencil, the typical code for which is shown in Figure 2.3.

Here, each outermost loop iteration (corresponding to S_2) uses the results of the previous iteration and hence must be executed sequentially. However, the iterations

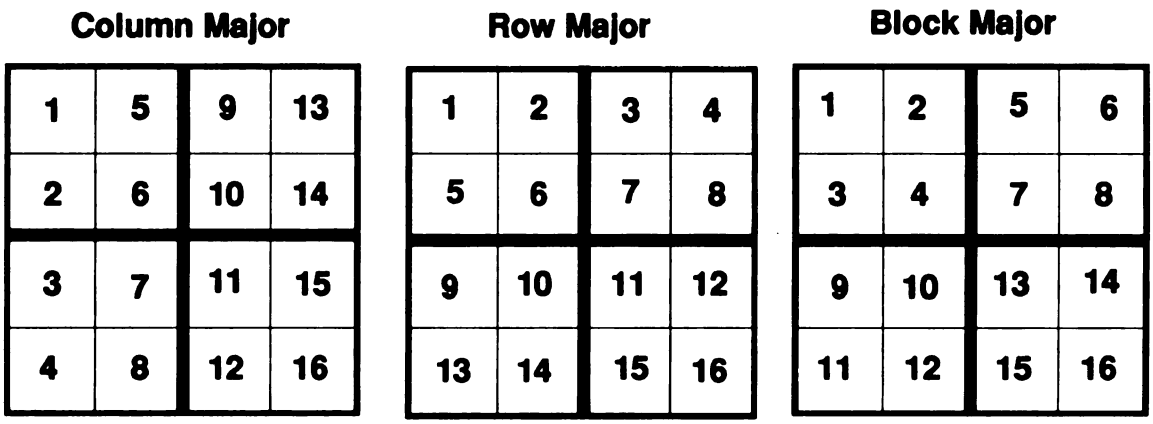

Figure 2.4. Different methods of data layout for a 4×4 array partitioned into four partitions.

of the two inner loops (corresponding to S_3 and S_4) can be executed in parallel. Since this application involves communication only with the nearest neighbors, we refer to it as application **near neighbor**.

The inputs to the program that generates the address trace for application **near neighbor** include the array size, the number of processors, the virtual page size, number of array elements per virtual page, and the layout of the array elements in the SVM. The address trace corresponds to a rectangular partitioning of the array elements among the various processors, each of which processes the elements allocated to it in a row-wise manner. We do not include the references corresponding to initialization, termination, and synchronization, when generating the address trace, because for the purposes of our study, it is sufficient to record the read and the write references corresponding to the computation. We also assume the presence of an instruction cache and therefore ignore instructions.

We study three schemes for laying out the array A in the SVM, out of which the *row major* and the *column major* schemes layout the array elements in row major and column major form, respectively. We refer to the portion of the array which is assigned to each processor as a *block*, and the *block major* scheme lays out array elements that belong to different blocks in separate regions of the SVM. These

schemes are illustrated for an example in Figure 2.4, where the number assigned to each array element is its virtual address. We refer to the three address traces generated due to these three schemes as applications **near neighbor/row major**, **near neighbor/column major**, and **near neighbor/block major**, respectively. The behavior of the first two applications are similar, except that the latter incurs more page faults because the array elements are processed row-wise and the layout of the data is in the column major form. Therefore, we do not consider application **near neighbor/column major** further in our study.

2.7 Experimental Results

Our objective is to study the interaction of page-level replication with hardware parameters and with page reference patterns created by various data layout schemes, and hence, we consider the FR and the NR placement strategies. Since replication is used to alleviate the high cost of accessing remote memory, we choose the ratio of the times to access remote and local memories (R) as a variable hardware parameter. Further, since replication needs more memory, we choose the number of pages allocated to the application as another variable hardware parameter. The number of pages is varied to cover both the *low page range region*, where there are a lot of disk page faults and the *high page range region*, where increasing the number of pages further did not lead to an improvement in performance. We use the applications **PIC**, **near neighbor/row major**, and **near neighbor/block major** to study various page reference patterns.

We now outline the fixed parameters used in our experiments. We conduct detailed experiments and conclude that for our workload, the fault processor scheme is better than the modulo scheme for initial page placement, and also, the WU scheme is better than the WI scheme to maintain consistency. Therefore, we choose the fault processor scheme and the WU scheme as fixed parameters. The values for the other fixed parameters are listed in Table 2.10. These values are typical of the TC2000 multiprocessor and are all normalized to the local memory access time t_{local}, which

Table 2.10. Fixed parameters.

Parameter p	t_p (unit t_{local})
rpfos	100
dfos	100
rlc	560
dmc	40000
lern	500
multicast	t_{remote}

is assumed to be 0.5 μs. We assume a disk access time of 20 *ms* and choose values for the software overhead to process various types of references and that for the **LERN** scheme based on the fact that the operating system's data is distributed evenly across all memory modules. The value of t_{remote} for the multicast operation that enforces the WU scheme is based on the assumption that all remote updates can take place simultaneously, which might not be true always. As we will see, our results are in fact strengthened when this assumption, which favors the FR case, does not hold.

We now discuss the results of simulations conducted with these fixed and variable parameters in terms of the performance metrics (EMAT and parameter overhead) defined earlier.

Application near neighbor/row major

Figures 2.5 through 2.8 show the parameter overhead versus the number of pages for $R = 2$ and $R = 10$, for both the FR and the NR cases. The parameter overhead graphs for the FR case can be divided into the low page range, the high page range, and the transition range regions. The overhead due to the **LERN** scheme, which is invoked in the low but not in the high page range region, causes the transition range region. The insignificance of this overhead in the NR case, because of infrequent page replacement, eliminates the transition range region in the corresponding graph. When the number of pages and R are both low ($R = 2$), disk page faults constitute the major overhead for both the FR and the NR cases. At high values of R ($R = 10$), remote

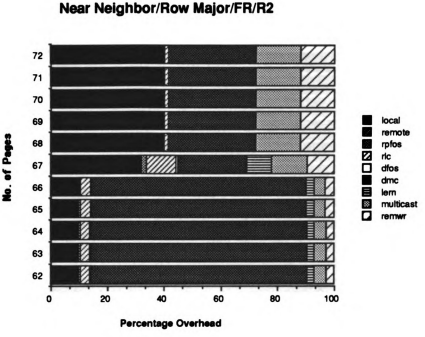

Figure 2.5. Parameter overhead (%) vs. No. of pages (FR/R=2/`near neighbor/row major`)

accesses share the overhead with disk page faults for the NR case. However, disk page faults still constitute the major overhead for the FR case, as replication reduces the number of remote accesses. When the number of pages is high, disk page faults are the minimum possible for both the FR and the NR cases. In the case of FR, local memory accesses account for a major share of the overhead, which is desirable because our goal is to make the EMAT as close to t_{local} as possible. Therefore, multicast and remote master writes constitute the major overhead and their share is more for higher values of R ($R = 10$). Remote accesses constitute the major overhead in the NR case for both high and low values of R.

The EMAT for the FR and the NR cases are shown in Figures 2.9 and 2.10, respectively. The steep slope corresponds to the transition from the low page range region to the high page range region. When both the number of pages and R is low ($R = 2$), the FR case incurs lots of **LERN** invocations and remote page faults, and therefore, its EMAT is worse than that for the NR case. When R is high, these factors

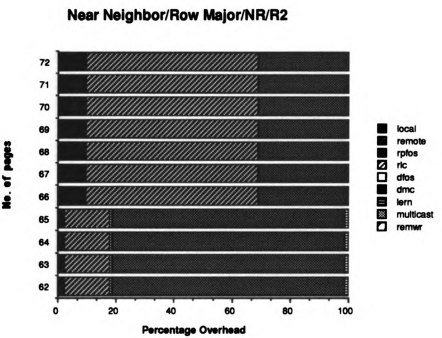

Figure 2.6. Parameter overhead (%) vs. No. of pages (NR/R=2/`near neighbor/row major`)

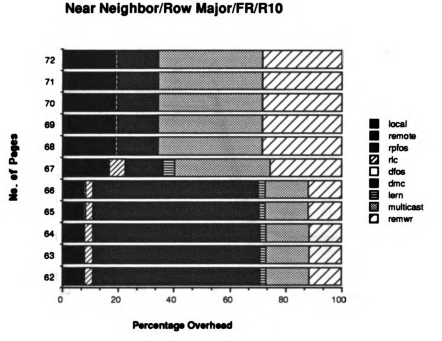

Figure 2.7. Parameter overhead (%) vs. No. of pages (FR/R=10/`near neighbor/row major`)

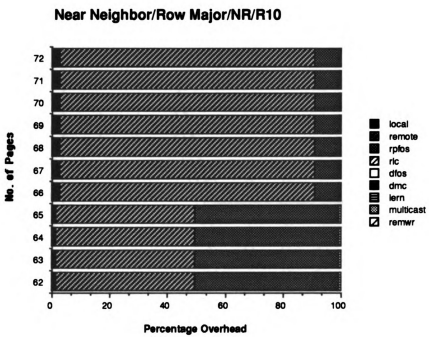

Figure 2.8. Parameter overhead (%) vs. No. of pages (NR/R=10/`near neighbor/row major`)

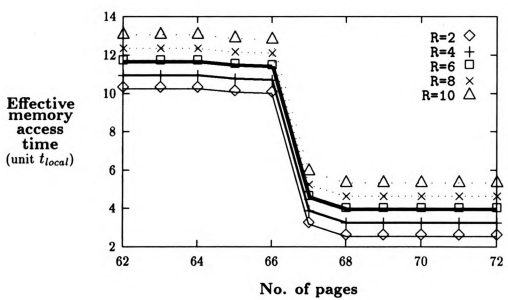

Figure 2.9. Effective memory access time vs. No. of pages (FR/`near neighbor/row major`)

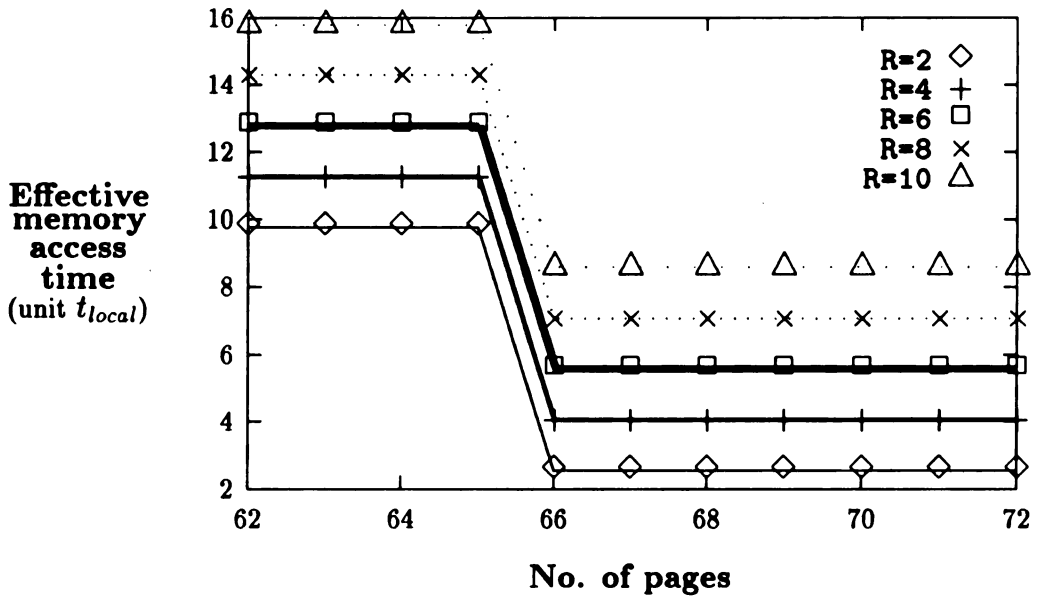

Figure 2.10. Effective memory access time vs. No. of pages (NR/**near neighbor/row major**)

are overshadowed by the remote accesses in the NR case, and hence, the EMAT for the FR case is better than that for the NR case. When the number of pages is high, the EMAT graph flattens after a certain number of pages, the actual number being smaller for the NR case than for the FR case. Again, for $R = 2$, the reduction in the EMAT for the FR case from that for the NR case is small, while it is significant when $R > 2$.

The best values of the EMAT occur in the high page range region for both the FR and the NR cases. The dominant parameters that affect the EMAT then are the remote accesses for the NR case and the remote master writes and the multicast for the FR case. Right now, pages are migrated only during certain types of page replacement, which are absent in the high page range region. Explicit page-level migration can reduce the number of remote master writes and remote accesses, and therefore, reduce the EMAT. Further, efficient multicast algorithms can reduce the multicast overhead and reduce the EMAT for the FR case. These results show that the effectiveness of replication depends on hardware parameters such as R, the available

physical memory, and the overhead of maintaining consistency.

Application near neighbor/block major

Figures 2.11 through 2.14 show the parameter overhead for $R = 2$ and $R = 10$, as the number of pages is varied, for the FR and the NR cases. As in the row major case, the overhead graphs for the FR case can be divided into the low page range, the high page range, and the transition range regions. The overhead due to page replacements is present when the number of pages is low and absent when it is high, and hence the transition range region in the FR case. The infrequent page replacements in the NR case eliminates the transition range region in the corresponding graph. When the number of pages is low, disk page faults constitute the major overhead for both $R = 2$ and $R = 10$, for both the FR and the NR cases. When the number of pages is high, for both the FR and the NR cases, disk page faults are at their minimum and it is desirable for the local access overhead to be high. Therefore, multicast and remote master write are the dominant parameters for the FR case, while remote access is the dominant factor for the NR case, for both $R = 2$ and $R = 10$.

The EMAT for the FR and the NR cases are shown in Figures 2.15 and 2.16, respectively. The steep slope corresponds to the transition from the low page range region to the high page range region. In all cases, EMAT flattens in the high page range region after a certain number of pages; this number is less for the NR case than for the FR case. The difference between this number for the FR and the NR cases is higher for the block major scheme than the row major scheme. Further, the EMAT for the NR case is better than that for the FR case, for all values of R and for the entire page range. The lower EMAT is because the block major scheme results in fewer shared pages and consequently, when the number of pages is high, the overhead for remote access in the NR case is less than that for multicast and remote master write in the FR case. Again, as in the row major case, the EMAT can be improved by explicit page-level migration and efficient multicast algorithms.

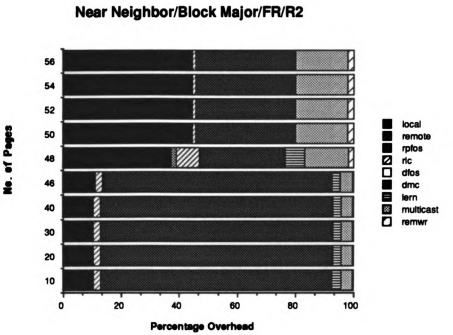

Figure 2.11. Parameter overhead (%) vs. No. of pages (FR/R=2/`near neighbor/block major`)

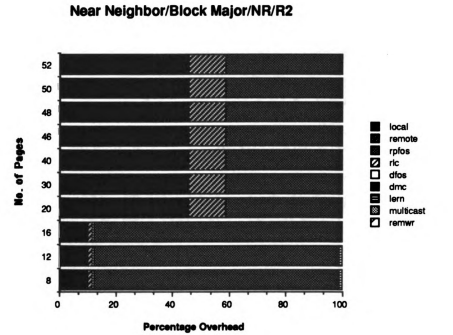

Figure 2.12. Parameter overhead (%) vs. No. of pages (NR/R=2/`near neighbor/block major`)

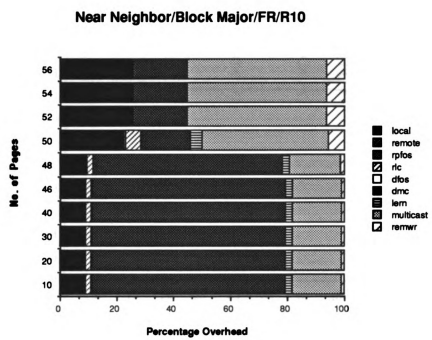

Figure 2.13. Parameter overhead (%) vs. No. of pages (FR/R=10/`near neighbor/block major`)

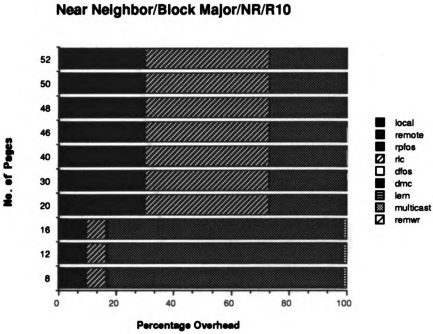

Figure 2.14. Parameter overhead (%) vs. No. of pages (NR/R=10/`near neighbor/block major`)

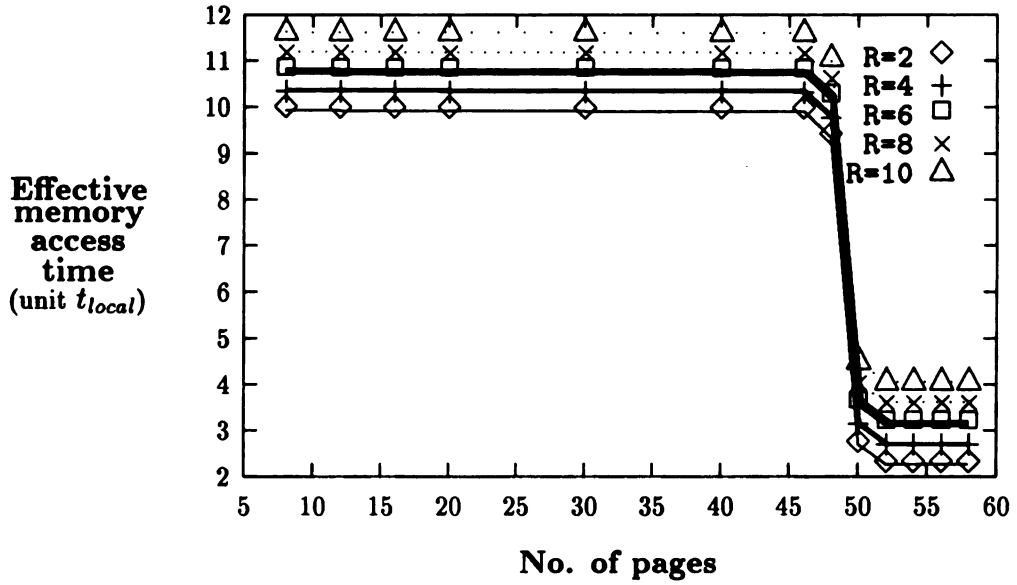

Figure 2.15. Effective memory access time vs. No. of pages (FR/**near neighbor/block major**)

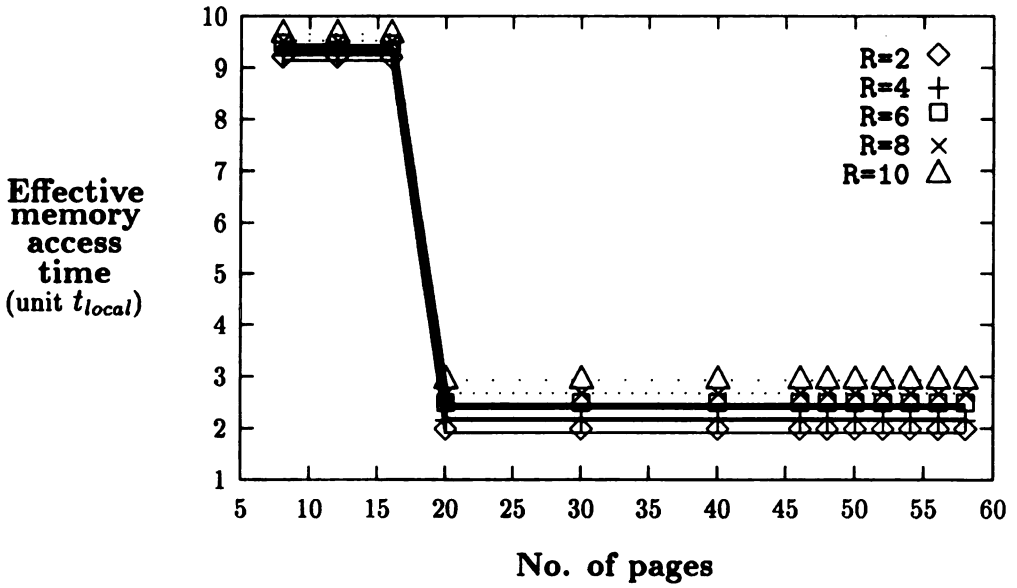

Figure 2.16. Effective memory access time vs. No. of pages (NR/**near neighbor/block major**)

Layout of Data: Row Major vs. Block Major

The row major scheme creates virtual pages with a higher degree of sharing than the block major scheme and hence, needs more memory for the FR case. Further, in the address trace for the row major scheme, the leftmost processor references a given row's virtual page first. This fact combined with the fault processor scheme leads to more pages being used by the leftmost processor. Therefore, application **near neighbor/row major** needs more memory for the NR case also. On the other hand, with the block major scheme, processors access virtual pages within their block, except when processing the border array elements, and consequently, need fewer pages for both the FR and the NR cases. Therefore, in both cases, the transition from the low page range region to the high page range region occurs at fewer pages with the block major scheme than with the row major scheme.

The best EMAT for the row major scheme occurs for the NR case when $R = 2$ and for the FR case when $R > 2$. On the other hand, for the block major scheme, the best EMAT occurs in the NR case for all values of R. When $R = 2$, the remote access overhead is lower for the block major scheme when compared to that for the row major scheme. When $R > 2$, the remote access overhead for the block major scheme is less than the remote master write and multicast overhead for the row major scheme. Therefore, the best EMAT for application **near neighbor** occurs with the block major and the NR schemes, and in addition, this combination uses the least amount of memory. These results underscore the fact that proper layout of data in the SVM can simplify data placement.

Application PIC

Figures 2.17 through 2.20 show the parameter overhead for $R = 2$ and $R = 10$, as the number of pages is varied for both the FR and the NR cases. For both high and low values of R in the FR case, disk page faults constitute the major overhead in the low page range region, and multicast and remote master writes constitute the major overhead in the high page range region. On the other hand, in the NR case,

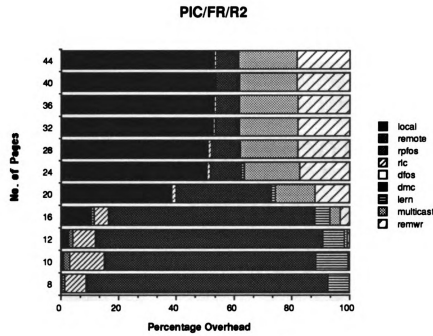

Figure 2.17. Parameter overhead (%) vs. No. of pages (FR/R=2/PIC)

remote accesses constitute the major overhead for all values of R and in the entire page range, indicating a high degree of sharing in application PIC.

The EMAT for the FR and the NR cases are shown in Figures 2.21 and 2.22, respectively. The slope corresponds to the transition from the low page range region to the high page range region. The EMAT graph flattens in all cases after a certain number of pages, the number of pages being fewer in the NR case than in the FR case. The fact that more pages are needed in the FR case concurs with the earlier statement that there is a high degree of sharing in application PIC. While the EMAT for the NR case is better than that for the FR case by a very small margin when $R = 2$, when $R > 2$, EMAT for the FR case is better than that for the NR case. Therefore, we conclude that the sharing in application PIC is mostly-read, as otherwise, the overhead of multicast would have made the EMAT for the FR case much higher. In summary, when R is high, FR can improve the EMAT for an application with a high degree of mostly-read sharing.

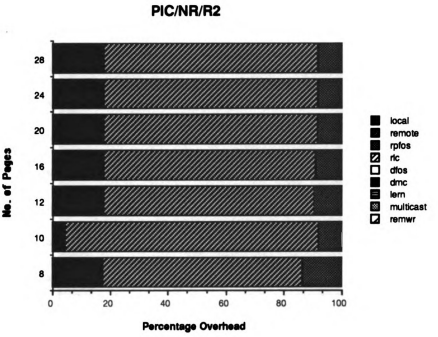

Figure 2.18. Parameter overhead (%) vs. No. of pages (NR/R=2/PIC)

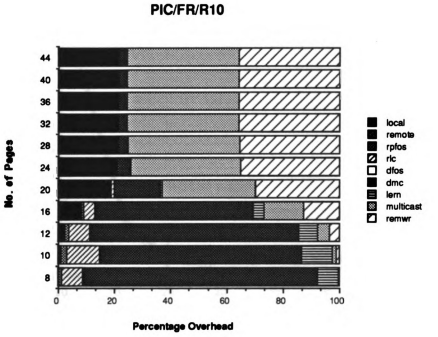

Figure 2.19. Parameter overhead (%) vs. No. of pages (FR/R=10/PIC)

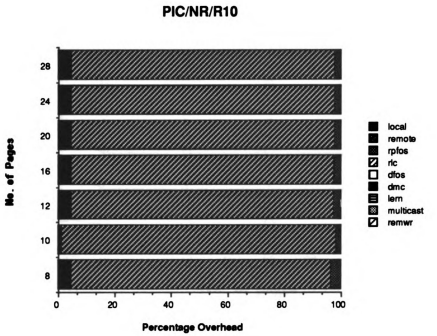

Figure 2.20. Parameter overhead (%) vs. No. of pages (NR/R=10/PIC)

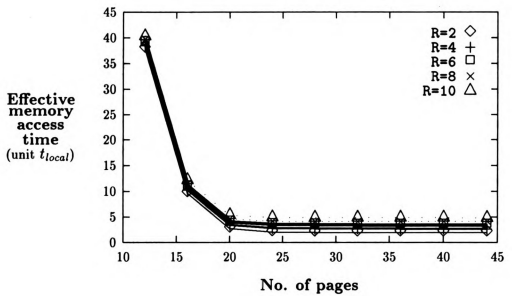

Figure 2.21. Effective memory access time vs. No. of pages (FR/PIC)

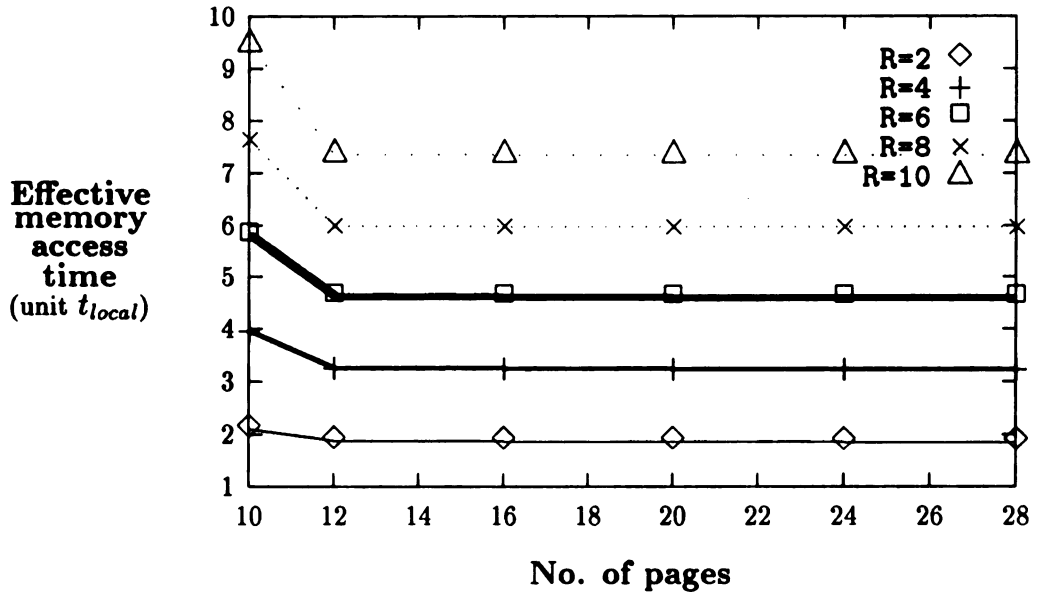

Figure 2.22. Effective memory access time vs. No. of pages (NR/PIC)

Conclusions

Our experiments show that whether to use page-level replication or not must be determined based on hardware parameters related to replication. While full replication does not provide a substantial performance improvement over no replication when remote memory accesses are relatively inexpensive, it improves performance when they are expensive. Further, since full replication needs more physical memory, when it is used for an application with a lot of shared data, insufficient physical memory can lead to performance loss due to disk page faults. When data is fully replicated and sufficient physical memory is available, the performance is limited by the time taken to keep data consistent. In essence, the choice of the data replication strategy should be adaptive to hardware parameters such as the remote memory access time, available physical memory, and the time it takes to make data consistent.

Our experiments also show that a page's reference pattern determines whether it should be replicated or not. When the sharing of virtual pages is low (application **near neighbor/block major**), performance is the same with and without replication. On

the other hand, when the sharing is high (applications `PIC` and `near neighbor/row major`), the best performance is achieved when pages are fully replicated. Further, the amount of sharing determines the physical memory needed to ensure that there are no disk page faults. In the presence of sufficient physical memory, performance of replication is determined by the overhead of maintaining consistency, which is directly related to the amount of sharing of a page.

Further, we find that the layout of data in the SVM influences which placement strategy provides a better performance. A data layout strategy that reduces the sharing of pages can eliminate the need for complex placement strategies. With proper layout of data, our experiments show that the performance of an application with and without replication is the same. We therefore conclude that proper data layout can simplify data placement.

In related work, Bolosky et al. [43] have concluded that page-level migration schemes need to be adaptive to hardware parameters that are related to migration. Also, LaRowe et al. [53] have developed a parameterized page placement scheme which can be adapted to hardware parameters and application reference characteristics by varying parameters.

2.8 Summary

In this chapter, we presented the results of our study which aims at identifying the limitations of existing block-level placement strategies. We establish that for performance reasons, page-level replication strategies must be adaptive to certain hardware parameters of the NUMA multiprocessor and also to the page's reference pattern. These results apply to other block-level placement strategies as well. It is tedious for the applications programmer to obtain by trial and error the placement scheme that performs the best for a given application and a specific NUMA multiprocessor. Further, in the absence of help from the compiler or the applications programmer, the runtime overhead of adaptive placement schemes is high. Also, the results of our study demonstrated how the compiler can play a role in simplifying data placement by

proper data layout in the SVM. These factors motivate our work on compiler-assisted data placement which is covered in detail in the following chapters.

CHAPTER 3

COMPILER-ASSISTED DATA PLACEMENT

In this chapter, we present our approach to compiler-assisted data placement when providing a shared virtual memory (SVM) in a NUMA multiprocessor. First, we motivate the need for the compiler to assist in placing data in memory, and then demonstrate that such help is necessary at all levels of the memory hierarchy. In order to assist data placement, the compiler needs information about the reference pattern of the SVM, and therefore, we next identify the various factors that determine this reference pattern. Then, we present our design of a compiler that assists data placement by using information about these factors. Next, we present our algorithms to determine this information, and finally we conclude by discussing related work.

3.1 Motivation

Proper data placement is important to reduce the performance degradation due to non-local memory accesses when providing a SVM in a NUMA multiprocessor. Data placement can be either *static* or *adaptive* depending on whether a fixed strategy is adopted or not. For example, a static page-level placement strategy replicates any referenced page while an adaptive strategy might replicate a page depending on the number of processors sharing it. Studies on block-level placement schemes have concluded that in order to achieve a high performance, such schemes must be

adaptive to related hardware parameters [43, 41, 42, 53] and also to the reference pattern of the block in question [29, 46, 41, 42, 53]. Previous studies have made it straightforward to choose an adaptive block-level placement strategy once the block's reference pattern is known. For example, Bennett et al. [29, 46] identify different types of data objects that occur in typical applications, and propose adaptive placement schemes for these objects. A read-only object is only read and never written, a mostly-read object is read more often than it is written, and a migratory object is shared by various processors in a round-robin fashion. Read-only and mostly-read objects can be replicated while migratory objects can be migrated.

Such adaptive placement schemes can be implemented only if the reference pattern is known, and this information can be gathered either at compile-time or during runtime. Examples of the latter case are page-level placement schemes reported in [15, 16] which maintain each page's reference history. In addition to space overhead, such schemes also incur time overhead because they frequently re-evaluate their decisions in order to ensure proper placement of pages whose reference patterns change rapidly. Such frequent re-evaluations, however, cause unnecessary overhead for pages whose reference pattern remains unchanged or changes infrequently. Further, these schemes can make proper placement decisions only if the past reference history is an accurate predictor of future references. LaRowe et al. [69] demonstrate that these schemes can benefit from hints in an application about changes in page reference patterns. Further, our study [41, 42] shows that simple reference patterns can simplify placement schemes. It is difficult for the applications programmer either to ensure that the reference patterns are simple or to provide hints about the reference pattern. It is therefore important to develop techniques that enable the compiler to achieve these objectives, when possible.

Another problem that limits the performance of block-level placement strategies is false sharing of blocks. We shall illustrate the problem by considering the application shown in Figure 3.1, which we use for illustration throughout this chapter. Assume that p processors are allocated to the application and that loop S_1 is parallelized by allocating consecutive sets of iterations to these processors. Further, the elements of

```
        Initialization

        DO S1 I = 1 TO n
             A(I) = B(I) + C(I)
S1      END DO

        DO S2 I = 1 TO m
             A(I) = D(I) + E(I)
S2      END DO

        DO S3 I = 1 TO q
             F(K(I)) = G(I) + H(I)
S3      END DO

        Termination
```

Figure 3.1. An example of a sequential application.

the array variables are laid out consecutively in the SVM. Then, for certain values of the block size s, it is possible for data elements which are exclusively accessed by different processors to reside in the same block (cache line or page), thus making it falsely-shared. For example, false sharing of certain blocks occurs when $n = 1000$, $p = 10$, $s = 32$ bytes, and each data element occupies four bytes, because each processor exclusively accesses 100 consecutive elements and a block contains eight elements.

Block-level placement strategies consider every processor that references at least one of the elements of a given block as one that shares the entire block. Hence, they consider all data elements of a falsely-shared block to be actually shared by processors referencing any of these data elements. Consider the situation when such a block is replicated and sequential consistency is maintained using the write-update (WU) scheme. On a write operation to a given data element, all copies of the block contain-

ing this data element are updated, even though some of the processors owning these copies will never reference this element. Though weaker consistency models such as release consistency reduce the frequency of updates, they also incur the performance loss due to unnecessary updates of falsely-shared blocks on a release operation. A write-invalidate (WI) scheme on the other hand causes unnecessary invalidations and subsequent misses. Alternatively, if this block is migrated among the local memories of interested processors, there is a ping-pong effect due to repeated invalidations and transfers. Instead, if the block is placed statically in a given processor's local memory, every other processor referencing any of its data elements incurs the cost of accessing it remotely.

Studies on caching in multiprocessors [48, 49, 50] have found that as the size of the cache line is increased, the number of cache misses decreases up to a certain line size, after which it increases. The initial reduction in cache misses is credited to prefetching facilitated by a larger block, while the subsequent increase is attributed to the higher degree of false sharing in a larger block. Since a page is much larger than a cache line, the potential of it being falsely-shared is correspondingly higher. Therefore, when NUMA multiprocessors lacking hardware cache consistency provide page-level placement schemes, it is important to eliminate the false sharing of pages, as rightly emphasized in [43, 69]. Our study [41, 42] also shows that proper layout of the data in the SVM reduces false sharing of pages and therefore simplies their placement.

One can argue that the performance loss due to false sharing can be minimized by reducing the granularity of placement operations. For example, in the KSR1 [6], each page is divided into several sub-pages, and the hardware provides replication with sequential consistency at a sub-page-level. The granularity of placement operations can also be reduced for software placement schemes. This reduction can be achieved either using hardware features for locking sub-pages such as that provided by the IBM's RISC System/6000 [70] or by maintaining software flags for each sub-page. However, the reduced block size in such an approach diminishes the benefits of prefetching. The reason large blocks provide the benefits of prefetching and do not

suffer false sharing in uniprocessors is because they contain data elements which are all referenced by a single processor. Similar results can be achieved in the case of multiprocessors if it is ensured that the blocks are not falsely-shared, irrespective of their size. Our study [41, 42] shows that false sharing of blocks can be reduced or eliminated by proper layout of data in the SVM, but it cannot be easily done by the applications programmer. Hence, it is important to develop techniques that enable the compiler to do it.

Another approach to solving the false sharing problem is to allow blocks to be falsely-shared, but invoke placement operations only when they are actually required. For example, consider one of the blocks belonging to array A when the loop S_1 of Figure 3.1 is parallelized as mentioned earlier. The data elements of this block are each exclusively written by a single processor. When such a block is replicated, it is sufficient to maintain consistency at the end of the loop, not for every write operation within the loop. In the case where data elements of a block are shared by different sets of processors, it is sufficient to maintain consistency of data elements which are actually shared, not the entire block. **SSVM** [23] and **Munin** [24] solve the false sharing problem in this manner, but they require the applications programmer to specify information about the reference behavior of data objects. It would be better if this information is provided by the compiler instead.

These factors motivate our work on compiler-assisted data placement when providing a SVM in a NUMA multiprocessor. There is some additional compile-time overhead when applying our techniques that assist data placement. This overhead, however, is not as critical as the runtime overhead of adaptive placement schemes which have no help from the compiler or the applications programmer. In addition, the overhead can be amortized over several runs of the application. It is also compensated by the ease of programming for the applications programmer who neither needs to layout the data properly nor provide information about the reference pattern. Finally, since the compiler can provide reasonably accurate information about the reference pattern in most situations, there is a high potential of improvement in runtime performance.

3.2 Data Placement and the Memory Hierarchy

The problem of data placement in NUMA multiprocessors has been addressed both by hardware cache-line-level protocols and software page-level placement strategies, which are provided at the cache and main memory levels of the memory hierarchy, respectively. In this section, we show that even when data is cached and hardware cache consistency is provided, proper page-level placement in the main memory is important, because it reduces remote page accesses that occur during cache line replacements, cache misses, and certain consistency operations. Therefore, techniques for compiler-assisted data placement should provide assistance to place data at all levels of the memory hierarchy.

We now derive an expression for the performance degradation in the absence of page placement, when data is cached. For simplicity, we ignore main memory accesses for cache consistency operations and cache line replacements, and measure performance by the EMAT, which we defined in Chapter 2. We ignore the time taken to service TLB hits and misses, and further, assume that there are no disk page faults. Therefore, a memory reference can result in a cache miss or a cache hit, and in the former case, the cache line is fetched either from remote or local memory, depending on where the corresponding page is located. We assume that the cache line is one word long and we also do not consider the times involved in page replication or migration when the page is absent in local memory.

Let t_{cache}, t_{local}, and t_{remote} be the times to access a single word in the cache, local, and remote memory, respectively, and let $L = t_{local}/t_{cache}$ and $R = t_{remote}/t_{cache}$. Let h_c and h_l be the cache and the local memory hit ratios, respectively, and p be the number of processors allocated to the application. We measure the EMAT, normalized to t_{cache}. In the presence of page placement, a cache miss is serviced from local memory with a probability of h_l, and therefore, the EMAT is given by:

$$EMAT_{with\_page\_placement} = h_c + (1 - h_c)(L \times h_l + R(1 - h_l)) \qquad (3.1)$$

In the absence of page placement, a cache miss has equal probability of being serviced

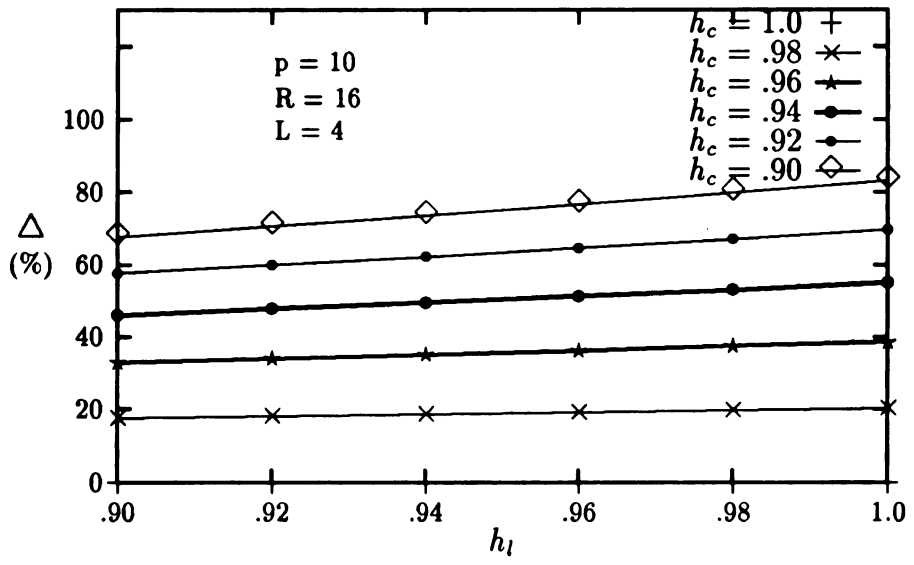

Figure 3.2. Performance degradation with caching and without page placement.

from the local memory of any of the processors. Therefore, the EMAT is given by:

$$EMAT_{without\_page\_placement} = h_c + (1 - h_c)(L/p + R(p-1)/p)) \tag{3.2}$$

The performance degradation in the absence of page placement $\Delta(h_c, h_l)$ is given by:

$$\Delta\,(h_c, h_l) = \frac{EMAT_{without\_page\_placement} - EMAT_{with\_page\_placement}}{EMAT_{with\_page\_placement}} \tag{3.3}$$

Substituting Equations 3.1 and 3.2 in Equation 3.3, we get

$$\Delta\,(h_c, h_l) = \frac{(1 - h_c)(h_l - 1/p)(R - L)}{h_c + (1 - h_c)(L \times h_l + R(1 - h_l))} \tag{3.4}$$

Equation 3.3 provides us information about the relationship of the performance degradation $\Delta(h_c, h_l)$ with various parameters. For example, when there are no cache misses ($h_c = 1$) or when the main memory access time is uniform ($R = L$), or when the local memory hit ratio equals the random placement hit ratio ($h_l = 1/p$), there is no performance degradation in the absence of page placement. Equation 3.3 gives

a value of zero for $\Delta(h_c, h_l)$ for each of these cases. For a given value of h_l, $\Delta(h_c, h_l)$ increases as h_c decreases. For a given value of h_c, $\Delta(h_c, h_l)$ increases as h_l increases. As the number of processors increases, the choice made by the random placement scheme becomes worse, and $\Delta(h_c, h_l)$ increases. The larger the values of L and R, the higher the performance degradation. Figure 3.2 shows how $\Delta(h_c, h_l)$ varies as h_c and h_l are varied, for fixed parameters of L = 4, R = 16, and p = 10. It is seen that even for high cache hit ratios (.90-.98), the performance degradation in the absence of page placement is quite high (20-80%).

Our analysis does not include the additional memory access involved in case a cache line needs to be replaced to service a cache miss. Such replacements occur for a fully utilized cache in the steady-state condition, and the benefits of proper page placement will be even higher under such conditions. We conclude that proper page placement is important even when data is cached and therefore, techniques for compiler-assisted data placement should be designed to help place data at all levels of the memory hierarchy.

3.3 Reference Pattern of the Shared Virtual Memory

Data placement strategies require information about the reference pattern of the various portions of the SVM. As a first step toward compiler-assisted data placement, we identify the various factors that determine this reference pattern. The application's code and data are allocated virtual space in the SVM. The portion of the SVM that contains the application's code is only read and never written. The area containing the application's data can be allocated statically or dynamically. It is not possible to determine at compile-time the reference pattern of the area which is allocated dynamically, and hence, we do not discuss it further. The area allocated for statically-declared variables can contain array, scalar, and synchronization variables. The reference pattern of a synchronization variable depends on the method of synchronization. The various factors that determine the reference pattern of scalar and

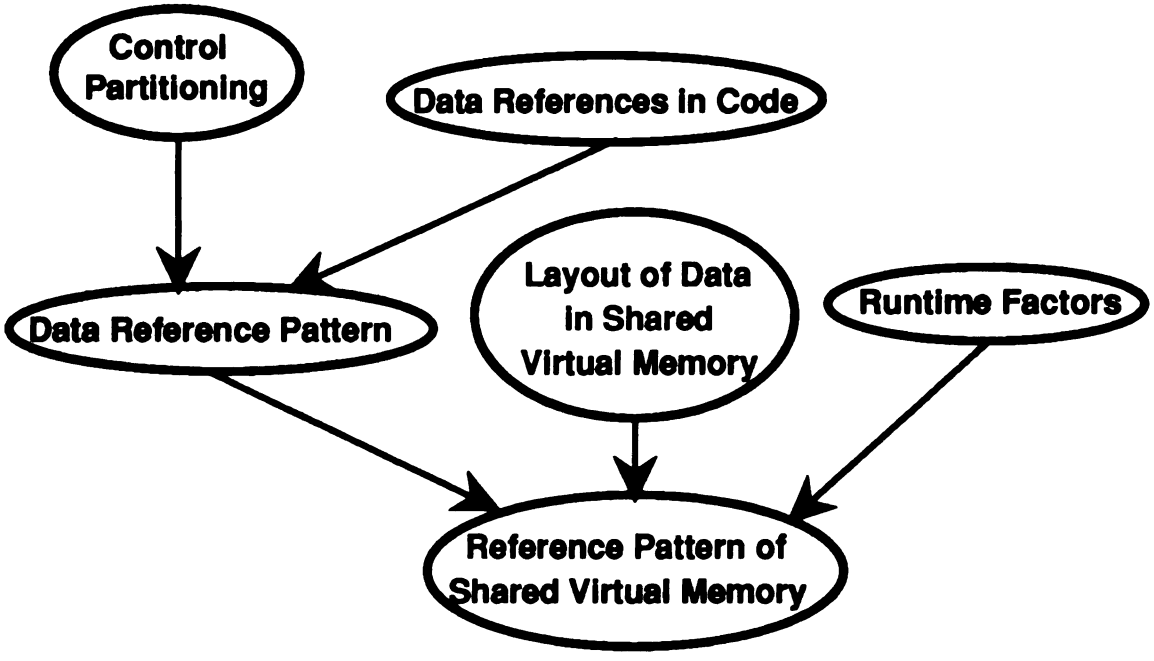

Figure 3.3. Reference pattern of the shared virtual memory.

array variables are shown in Figure 3.3.

Here, *control partitioning* refers to the distribution of the work of a parallel application among the various allocated processors. For example, loop S_1 in Figure 3.1 can be control-partitioned by assigning to each processor a distinct set of the values taken by the loop index I. A *data reference* refers to any reference to a data element in the program. A data reference can be a scalar or an array reference and an example of the latter is $A(I)$ which appears in loop S_1. In this array reference, the value of I determines which element of array A is referenced, and since $A(I)$ appears on the left hand side of the program statement, the corresponding element is written. On the other hand, array reference $B(I)$ in the same loop appears on the right hand side and hence the element referenced by it is read. When the iterations of loop S_1 are partitioned as mentioned above, each processor executes instances of the array references $A(I)$, $B(I)$, and $C(I)$ corresponding to its assigned values of I. The data reference pattern of array variables are influenced by these factors and a similar discussion applies to scalar variables as well. Hence, control partitioning and the nature of data references

in the code together determine each data element's reference pattern and hence, that of its virtual address.

The layout of data determines which data elements reside in a given block of the SVM such as a page or a cache line. Therefore, the reference pattern of each block is influenced by that of the data elements it contains. In addition, it is influenced by various runtime factors which include replacement and placement schemes, the synchronization needs of the application, and contention for hardware resources such as the interconnect, memory module, and the memory port. These factors do not change the number of read and write references in the user state by the various processors to each block in the SVM, but change only the order in which these references occur. It is difficult at compile-time to accurately predict and incorporate the perturbation to the reference pattern due to these runtime factors. Also, we do not think that this perturbation is significant enough to warrant a change to the placement strategy of a block chosen based on its reference pattern in the absence of this perturbation. Hence, we ignore these runtime factors when developing techniques that assist data placement.

In general, the nature of references to data elements is different in various parts of the program. Further, control partitioning is different for various loops of the application. Hence, the data reference pattern changes with time during the execution of a parallel application and so does the reference pattern of the SVM. For example, the reference pattern of the elements of array A can be different in loops S_1 and S_2 in Figure 3.1, for some combination of the values for n and m and certain methods of control-partitioning the iterations of these loops.

3.4 Design of the Compiler

In this section, we outline our design of a compiler that assists data placement. We present our design by considering the various factors that influence the reference pattern of the SVM (Figure 3.3), and also other factors.

```
        DOSEQ
             Initialization
        END DOSEQ

        DOALL S1 I = 1 TO n
             A(I) = B(I) + C(I)
S1      END DOALL

        DOALL S2 I = 1 TO m
             A(I) = D(I) + E(I)
S2      END DOALL

        DOSEQ S3 I = 1 TO q
             F(K(I)) = G(I) + H(I)
S3      END DOSEQ

        DOSEQ
             Termination
        END DOSEQ
```

Figure 3.4. An example of a parallel application.

Application Programmer Interface

The applications programmer writes the application either in a sequential language or in a parallel language. In the former case, the application is converted into a parallel application as done by parallelizing compilers [71, 72, 73, 74, 75]. In both cases, the parallel application consists of several code segments. We define a code segment to be *sequential* if it is executed by a single processor, and *parallel* if it is executed by more than one processor. A parallel code segment contains DOALL loops [73] in which all iterations are independent and can be executed in parallel and in

any order without any synchronization, or DOACROSS loops [73] in which there are dependencies among different loop iterations. Barrier synchronization operations are inserted after each parallel code segment. For example, the sequential application in Figure 3.1 can be converted into a parallel application shown in Figure 3.4, where DOSEQ refers to a sequential code segment.

Control-Partitioning

As shown in Figure 3.3, in order to determine the reference pattern of the SVM and assist data placement, the control-partitioning information needs to be available at compile-time. In general, loop iterations can be partitioned either at compile-time (static scheduling) or during runtime (dynamic scheduling). Though dynamic scheduling policies such as self-scheduling [76], guided self-scheduling [77], trapezoidal self-scheduling [78], and factoring [79] may reduce processing time, they do not allow the data reference pattern to be determined at compile-time. With processors getting faster relative to memory, the time spent on memory accesses constitute the major fraction of the execution time and must be reduced for good performance, particularly for NUMA multiprocessors. Also, Abraham and Hudak [80] control-partition iterative parallel loops at compile-time in order to reduce interprocessor communication, and show that the performance is better than guided self-scheduling. It is precisely for the same reason that compile-time data partitioning techniques e.g., [81, 82], have been studied for message-passing NUMA multiprocessors. Therefore, we consider compile-time partitioning of the loop iterations.

In our compiler that assists data placement, for the general case, the applications programmer specifies the control-partitioning of iterations for each parallel code segment. This requirement is similar to the need for the applications programmer to specify the data partitioning in languages such as FORTRAN-D [83] for message-passing NUMA multiprocessors. For specific cases, our compiler uses heuristics, which are derived from studies in the area of either parallel algorithms [84, 68] or compile-time control-partitioning of iterations [80].

Data References

In order to assist data placement, the compiler should also be able to resolve data references. We develop new algorithms as well as extend existing dependence analysis techniques (surveyed in [85, 86]) to resolve data references, as discussed in detail in the next section. It is not possible, however, to resolve data references that are functions of values which are unknown at compile-time. For example, the data references in the loops in Figure 3.1 cannot be resolved when the loop upper bounds n, m, and q are unknown at compile-time. Even if the loop upper bound q is known at compile-time, it is not possible to resolve the array reference $F(K(I))$ in loop S_3. Another condition under which the data reference pattern cannot be determined at compile-time is when processors execute code conditionally and the condition depends on the input data. An example is the QR factorization application in which processors choose a column of an array that has the maximum sum of the square of its elements. Note however that the applications programmer also cannot resolve these references which are dependent on values known only during runtime. We propose that the compiler try to resolve such references using profiling as in IMPACT [87]. If profiling does not help either, then the corresponding data reference pattern cannot be determined and consequently, the compiler cannot assist in placing data for related portions of the SVM.

Temporal Variation of Reference Pattern

We represent the reference pattern of a data element in a given code segment by the processors that read and those that write the data element in this code segment. We take care of the temporal variation of the reference pattern by using this representation individually for each code segment.

Assistance in Data Placement

The algorithms we discuss in the next section can be used to determine the type and the reference pattern of the data in each code segment. We use this information to assist data placement by means of *compile-time objects* which contain data of the

program → {*code segment*}$^+$
code segment → *sequential code segment* | *parallel code segment*
sequential code segment → *sequential code* | *placement directives*
parallel code segment → *Nested DOALL* | *Nested DOACROSS*

Figure 3.5. A model of a parallel application with placement directives.

same variable type and similar reference patterns. These objects assist data placement at all levels of the memory hierarchy, and the manner in which they are created depends on the type of placement. In our thesis, we consider block-level placement and also object-level placement which is similar to that provided by **SSVM** [23]. Between each pair of consecutive code segments, our object-creation schemes insert a set of *placement directives* which assist data placement and which are to be executed sequentially. The parallel application is thus transformed as shown in Figure 3.5. In the next two chapters, we discuss the object-creation schemes, their compile-time complexity, the runtime performance they offer, and other implementation issues, for the block-level and object-level types of placement, respectively.

If the compiler is unable to resolve some data references, it cannot determine the reference pattern of corresponding data elements. In this case, it specifies that it cannot assist in placing data for related portions of the SVM. The influence of these unresolved references depends on the type of data placement, and we discuss it in the context of the block-level and object-level types of placement in the next two chapters. The data elements which cannot be placed by the compiler need to be placed using the runtime reference history. The runtime overhead is still less than that in the absence of any form of compiler assistance. Therefore, as shown in Figure 3.6, our approach is for the compiler to assist data placement whenever possible, and leave the rest of the cases for the runtime mechanisms. The compilation process in our compiler that assists data placement is shown in Figure 3.7.

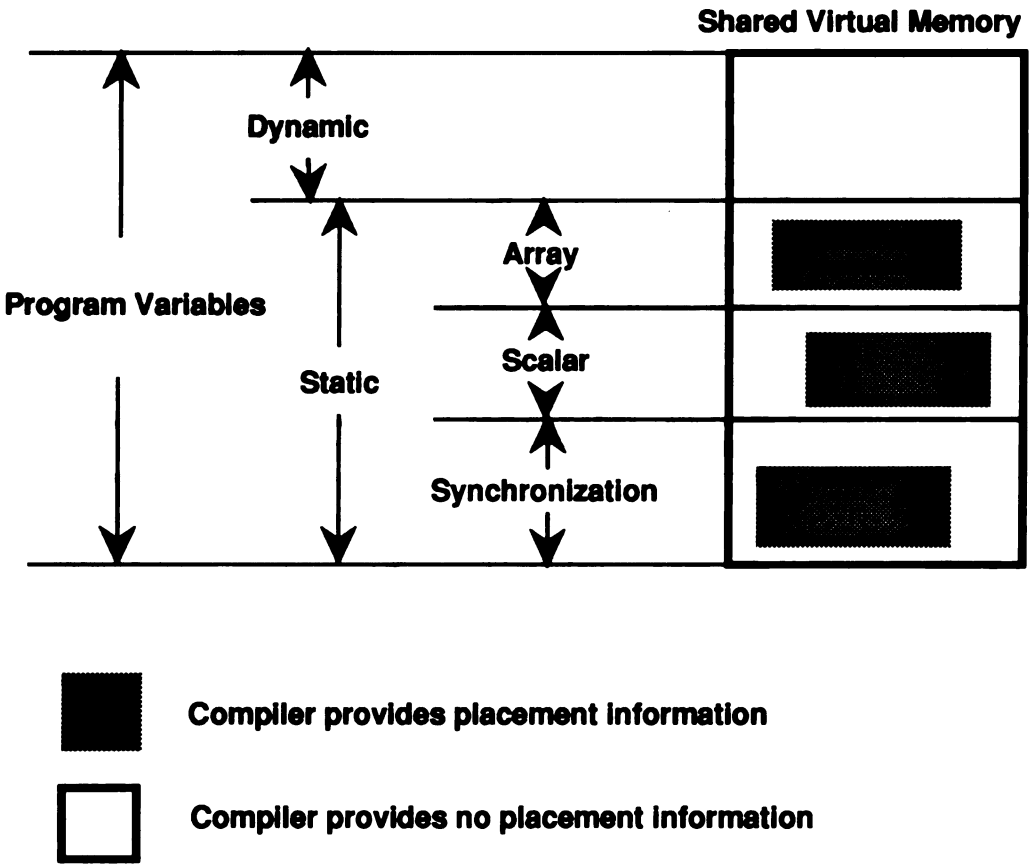

Figure 3.6. Placement information about the shared virtual memory.

3.5 Type and Reference Pattern of Variables

Our object-creation schemes which assist the block-level and object-level types of placement need the type and the reference pattern of data elements. It is easy to determine by inspection whether a variable is of the array or the scalar type. The type of each synchronization variable is specified by the applications programmer for applications written in a parallel language. It is provided by the compiler if the parallel application is derived from a sequential version using parallelizing techniques.

We consider the barrier, wait/signal and the lock types of synchronization. We determine the reference pattern of synchronization variables as follows. In general,

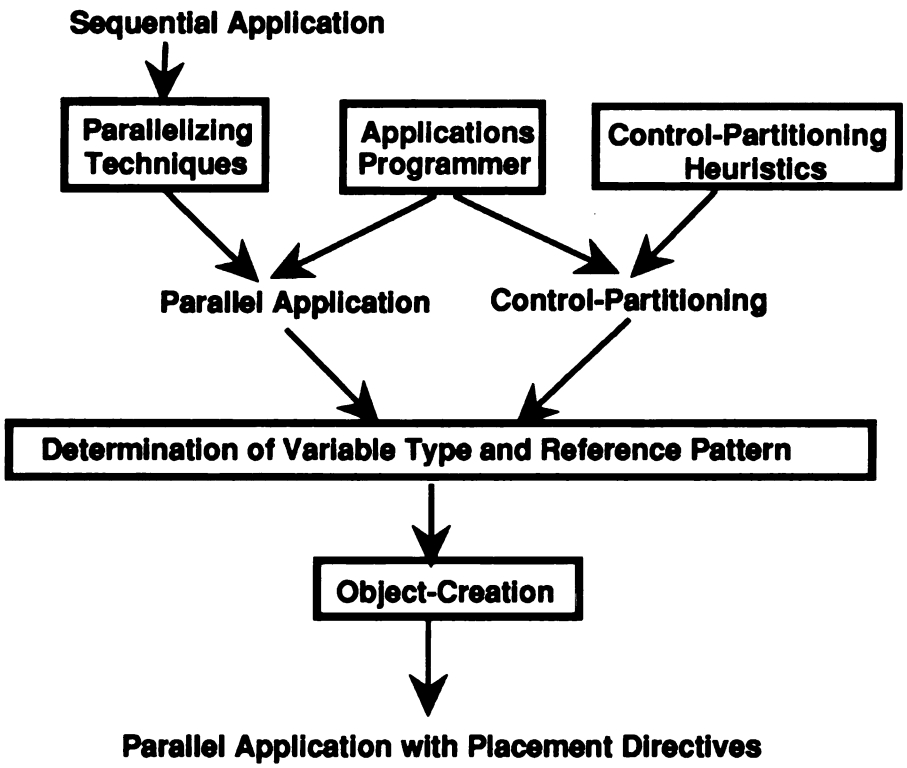

Figure 3.7. Steps in compiling an application to assist data placement.

barrier synchronization is used to ensure that the execution of the code before the barrier is completed before the start of the execution of the code after the barrier. For example, it is can be used at the end of a parallel code segment, in which case the corresponding synchronization variable is referenced equally by all processors. Wait/signal synchronization is used either to wait for an event to occur or to signal that an event has occurred. For example, it can be used to enforce dependencies in a DOACROSS loop, in which case the corresponding synchronization variable is referenced by the processors involved in the dependency in question. The lock synchronization variable is used to ensure mutual exclusion in accessing shared data, and is referenced by the processors that share the data elements it protects. Its reference pattern is the same as that of these data elements.

We now present our algorithms to determine the reference pattern of statically-declared array and scalar variables in a given code segment. Recall that the reference pattern of a data element is represented by the processors which read and those that write it. We record the information about the read references for each data element either by a flag or by a counter per processor. The information about write references is recorded in a similar fashion. A flag records whether there is a reference or not while a counter records the actual number of references. It is straightforward to determine the reference pattern for scalar variables and that for array variables with constant subscripts. When such references occur within the scope of the induction variable whose iterations are control-partitioned among the processors, the corresponding element is referenced by all processors. Further, the element is read or written depending on whether the reference appears on the right or the left hand side of the program statement. We now present algorithms to determine the reference pattern for other types of references.

Algorithm A

The first algorithm, which we refer to as **Algorithm A**, is shown in Figure 3.8. For our purposes, a scalar variable is a single-element array variable. **Algorithm A** computes the data reference pattern by determining the elements referenced for each point in the iteration space for each array reference. Though we present the algorithm for a perfectly nested loop, it can be easily generalized to loops that are not perfectly nested. Since it scans the entire iteration space, **Algorithm A** provides the reference pattern of each data element in terms of the actual number of read and write references to it by each processor.

We now illustrate how **Algorithm A** determines the data reference pattern of a given parallel code segment. We consider the loop S_1 shown in Figure 3.4. Assume that $n = 20$ and that four consecutive iterations of the loop index I are each allocated to five processors. Each point in the iteration space corresponds to a given value of I. For the array references in loop S_1, each such point writes an element of A, and reads an element of both B and C. The processor that executes this point of the

```
FOR each array reference
  FOR each loop induction variable
    FOR all iterations

/* The element referenced is determined by the subscripts and the
current values of the loop induction variables. The processor that
references the element is the one that executes the iteration
corresponding to the current values of the loop induction variables.
The reference is a read (write) if the array reference appears on the
right (left) hand side. */

        update reference pattern of appropriate element
    END FOR
  END FOR
END FOR
```

Figure 3.8. Determination of reference pattern: **Algorithm A**.

iteration space is also the one that executes these write and read operations. Using these facts, **Algorithm A** determines that the read counters for elements 0-3 of arrays A, B, and C are (0 0 0 0 0), (1 0 0 0 0), and (1 0 0 0 0), respectively. Similarly, the write counters for elements 0-3 of arrays A, B, and C are (1 0 0 0 0), (0 0 0 0 0), (0 0 0 0 0), respectively. It determines the counters for the other elements in a similar manner.

We now derive the time and the space complexity of **Algorithm A**. Let l be the depth of the loop, i be the maximum out of the number of iterations for each loop induction variable, r be the number of data references, d be the maximum out of the number of dimensions for each array variable, and p be the number of processors allocated to the application. We ignore the time spent in parsing, and assume that the data references have already been obtained. Therefore, the worst-case time complexity to determine the reference pattern of all data elements referenced in a given code

segment is $O(l \times i \times r \times d)$. The space complexity for each data element is $(2 \times p)$ units, where the unit is a bit if we only need to store whether a processor reads or writes a data element or not. If we need to store the actual number of reads and writes, the unit is the storage space required for an integer. It is possible to reduce the space requirements by allocating space for only one variable and processing variables one at a time.

Algorithm B

Since the loop depth and the number of loop iterations appear in the time complexity of **Algorithm A**, it is computationally intensive for large values of these parameters. Our next algorithm, referred to as **Algorithm B**, has a lower time complexity than **Algorithm A**, and is motivated by the work done by Jeremiassen and Eggers [86]. In this work, they propose an algorithm to determine the per-process side effect information, which represents the variables read and those written by each process, and is equivalent to our representation of the data reference pattern for each code segment. Their algorithm uses existing dependence analysis techniques [71, 72, 73, 74, 75] that are used to extract the control and data dependences of a sequential application, which is then parallelized such that these dependences are satisfied.

We now provide the traditional definition of data dependence [73]. Consider two statements S_1 and S_2 in the *control flow graph* (CFG) of a sequential application. A data dependence exists between statements S_1 and S_2, with respect to a variable X if and only if

1. There exists a path in the CFG from S_1 to S_2 with no intervening write to X, and
2. at least one of the following is true:
   - **(a)** *flow*, X is written by S_1, and later read by S_2, or
   - **(b)** *anti*, X is read by S_1, and later written by S_2, or
   - **(c)** *output*, X is written by S_1, and later written by S_2, or

(d) *input*, X is read by S_1, and later read by S_2.

In general, in order to determine the data dependences in a sequential application, dependence analysis techniques compute the side-effect information for each statement S_i, which is represented by the *USE(S_i)* and the *MOD(S_i)* sets, that contain the variables it uses and those that it modifies, respectively. Input dependences represent reads to the same variable and can be performed in any order. Therefore, they can be safely ignored for automatic parallelization. The dependences between any two pair of statements are determined by applying the intersection operation to each of the other three pairs of sets which can be derived from their USE and MOD sets. For example, if S_1 and S_2 are two statements in a sequential application, there is no data dependence between them if the sets MOD(S_1) $\cap$ USE(S_2), USE(S_1) $\cap$ MOD(S_2), and MOD(S_1) $\cap$ MOD(S_2), are all empty. On the other hand, if one or more of these sets is non-empty, then a data dependence exists between S_1 and S_2. As surveyed in [86, 85], dependence analysis techniques can also analyze statements containing procedure calls and in addition, provide the side-effect information of an array variable for either individual elements or a set of its elements, referred to as an *array section.*

Jeremiassen and Eggers [86] develop an algorithm to identify the per-process control flow graphs, given the control flow graph of the sequential application. They suggest that the per-process side-effect information can then be determined by applying existing dependence analysis techniques to the control flow graphs of the various processes. Their motivation in determining the side-effect information is to assist cache-line-level placement, but they do not mention how they plan to provide such assistance. Their work motivates us to use dependence analysis techniques in **Algorithm B**, our next algorithm to determine the data reference pattern. Unlike their approach, our algorithm determines the data reference pattern by incorporating control-partitioning in existing techniques that compute interprocedural side-effect information. Further, we demonstrate how the data reference pattern can be used to assist data placement by developing techniques [54, 55, 56] to assist both block-level and object-level placement. Our techniques can also be applied when the data

```
FOR each data reference
  IF scalar reference
    IF any applicable loop induction variable is control-partitioned
      set write (read) flag of all processors if reference occurs on
        left (right) hand side
    END IF
  ELSEIF array reference
    replace each subscript that contains a loop induction variable by
      the corresponding (lower_bound, upper_bound, step) information
    IF no loop induction variable is control-partitioned
      set each processor's array section to (lower_bound, upper_bound, step)
    ELSEIF
      compute array section for each processor based on partitioning
    END IF
    add each processor's section to its write (read) list
      if reference occurs on left (right) hand side
  END IF
END FOR
/* convert read and write list information for each array variable
into element reference pattern */
  FOR each array variable
    FOR each element
      FOR each processor
        IF element exists in read list
          set read flag
        END IF
        IF element exists in write list
          set write flag
        END IF
      END FOR
    END FOR
  END FOR
```

Figure 3.9. Determination of reference pattern: **Algorithm B.**

reference pattern is determined using other algorithms such as theirs.

Since dependence analysis techniques have been traditionally used to extract parallelism from sequential applications, they are conservative and do not always provide the exact side-effect information. For example, in the case of a conditional IF statement, these techniques assume that both branches are taken. Further, they sacrifice exactness in order to reduce the space required to store the side-effect information. For example, the side-effect information for an array variable is stored for its sections, not for its individual elements, thereby saving space at the cost of losing exactness. It is important to determine the reference pattern accurately in order to assist data placement. Therefore, in extending these techniques to determine the data reference pattern, we preserve the exactness of the data reference pattern at the cost of additional space and time complexity. For example, in the case of conditional statements, we compute the data reference pattern separately for each possible condition.

We derive **Algorithm B** by incorporating control partitioning in the algorithm for interprocedural side-effect analysis discussed in [85]. **Algorithm B** is shown in Figure 3.9; it is applicable irrespective of whether the loop in the code segment is perfectly nested or not, and it handles linear subscripts. Unlike **Algorithm A**, it does not scan all the points in the iteration space. It maintains a read and a write list per variable for each processor which contain elements of the variable that are read and written by this processor, respectively. It adds entries to these lists by processing the various data references in the code segment.

Any scalar variable within the scope of a loop induction variable which is control-partitioned is assumed to be referenced by all processors. This variable is added to the read or the write lists of all processors depending on whether it appears on the right or the left hand side of a program statement, respectively. In the case of each array reference, appropriate sections of the corresponding array are added to the read and write lists of the various processors based on the control-partitioning of the loop induction variable in each of the subscripts. Once all the data references in the code segment are processed in this manner, each data element's reference pattern is determined by examining the read and write lists of the corresponding variable. In

this manner, **Algorithm B** provides the reference pattern for each data element in terms of whether each processor reads or writes it or not. It does not provide the actual number of read and write references.

We now illustrate how **Algorithm B** determines the reference pattern for an example parallel code segment. Again, we consider the loop S_1 shown in Figure 3.4. We assume that $n = 20$ and that four consecutive iterations of the loop index I are allocated to each of the five processors. **Algorithm B** adds $A(1:5)$ to the write list for variable A of processor 0, $A(6:10)$ to the write list for variable A of processor 1, and so on. Similarly, it adds $B(1:5)$ to the read list of variable B of processor 0, $B(6:10)$ to the read list for variable B of processor 1, and so on. The array reference $C(I)$ is handled in a similar fashion. **Algorithm B** determines the reference pattern for each data element by looking up the entries in the read and the write lists of the corresponding variable. An optimization is to stop the lookup as soon as an entry which contains an element is found. In this example, it is determined that $A(1)$ is written by processor 0 because the entry $A(1:5)$ is found in the write list for variable A of processor 0. Similarly, it is determined that $A(1)$ is not written by the other processors because no entry in their write lists contain it. The reference pattern of other data elements can be determined in a similar manner.

We now derive the time and space complexity for **Algorithm B**. Let r be the number of data references, d be the maximum out of the number of dimensions for each array variable, e be the maximum out of the number of elements of an array variable, and p be the number of processors allocated to the application. The time complexity of **Algorithm B** is $O(r \times d \times p + e \times p \times r \times d)$. It requires the same space as that needed by **Algorithm A** to store the data reference pattern. Some additional space with a worst-case complexity of $O(p \times r \times 2)$ is required to store the read and the write lists for the various processors. Note that **Algorithm B** executes faster but requires more space than **Algorithm A**.

3.6 Related Work

Related work includes those in the areas of parallelizing compilers, partitioning, locality enhancement, prefetching, software-assisted cache coherence, and false sharing. We discuss each of these below.

Parallelizing Compilers: In order for our compiler to assist data placement, the parallelism in the application needs to be expressed using the DOALL and the DOACROSS constructs. When the application is written in a sequential language, we use dependence analysis techniques in existing parallelizing compilers [71, 72, 73, 74, 75] to identify and express the application's parallelism using these constructs. In addition, we extend these techniques by incorporating control-partitioning to develop algorithms which determine the data reference pattern. Further, our compile-time object-creation scheme which directs object-level placement uses these techniques to determine when it is necessary to make copies of each data element consistent.

Partitioning: We control-partition the iterations of each parallel code segment and develop techniques to place data in the NUMA physical memory. Independently, Hudak and Abraham [88, 89] have used control-partitioning and studied data placement for a single loop construct, viz., stencil-based iterative data parallel loops with near-neighbor communication. An example of an application with such a loop construct is the application `near neighbor` which we considered in Chapter 2. Their algorithm to determine the reference pattern, the sharing characteristics they consider, and their solution to the false sharing problem are all specific to this loop construct. They develop compile-time techniques that classify the data in this loop construct into various sets depending on the near-neighbor sharing characteristics. They demonstrate the performance improvement due to these techniques by measuring execution times for different methods of placing these sets at various levels of the memory hierarchy of the BBN TC2000. Their techniques for control-partitioning [80] in order to minimize interprocessor communication perform better than guided self-scheduling, and are applicable to our work. However, as we discuss below, our method of compiler-assisted

data placement [54] differs from their work in several important aspects.

Firstly, our approach is more general because we consider an entire application containing multiple code segments instead of just a single loop construct. Our algorithms to determine the data reference pattern are applicable to any DOALL and DOACROSS loops, whether perfectly nested or not. Also, our definition of the data reference pattern covers all possible sharing characteristics and not those for a specific loop construct. Our idea of creating compile-time objects using both the data reference pattern and the variable type is new. Further, these objects are created differently for various types of placement schemes. Our solution to the false sharing problem which uses these objects is not specific to a given application. Our approach can assist in placing the data of the entire application at all levels of the memory hierarchy in a shared-virtual-memory NUMA multiprocessor. It is applicable whether direct remote memory access is possible or not and also for both the block-level and the object-level types of placement. We also develop algorithms for addressing issues such as code generation, thereby enabling easy incorporation of our schemes in a compiler.

Other studies on control-partitioning are also of interest to us. Control-partitioning of two-dimensional iterative spaces has been studied in the literature. Vrsalovic et al. [84] develop a model to measure the speedup of several partitioning methods for an architecture in which each processor has both local and global memory. By modeling both message-passing and shared-memory architectures, Reed et al. [68] conclude that the overall performance depends on a complex interaction of three factors, viz., the stencil, the partition type, and the architecture. This result is related to the *mapping* problem [90] which involves mapping a parallel application to a parallel machine. The problem of obtaining an optimal mapping of a general parallel application to a general parallel machine is NP-complete [91]. In practice, an applications programmer obtains the best partitioning and mapping of an application for a given NUMA multiprocessor by trial and error. An alternate approach is to use heuristic techniques such as simulated annealing [92] which we used in an earlier work [93, 94] on the mapping problem. Brochard and Freau [95] have studied the influence

of various partitioning and data placement strategies for the matrix multiplication and finite element applications for the IBM RP3. Unlike our work, these studies do not consider compile-time techniques that assist data placement.

The approach of control-partitioning followed by data placement is different from that used by compilers for message-passing NUMA multiprocessors also referred to as *distributed-memory message-passing* (DMMP) multiprocessors. These approaches differ in the process-interaction paradigm (shared-memory versus message-passing) but aim at achieving the same objective of minimizing the performance degradation due to non-local accesses. In compilers for DMMP multiprocessors e.g., [96, 97, 98, 99, 74, 83, 100, 81, 82], data is partitioned among the local memories of the various processors, and each processor owns the data in its local memory. All processors execute the same program and the work done by each processor is determined by certain rules. For example, one such rule specifies that the owner of each data element computes its value. A processor uses messages to get a copy of a data element which it does not own. Most of these compilers require the applications programmer to specify the data partitioning. They then automatically partition the work among the processors and generate each processor's code as well as the inter-processor communication. Communication primitives are generated during runtime [98] for data references which cannot be resolved during compilation. Static performance estimators [100] and automatic data partitioning techniques [81, 82] have been developed. The issues involved in translating control-partitioned programs into data-partitioned programs are discussed in [101].

Locality Enhancement: Another approach to improve memory performance is to maximize the locality and reusability of data. Abu-Sufah et al. [102] study program analysis and transformations to improve reusability of data in uniprocessor paging systems. Gannon et al. [103] define a reference window for a dependence as the variables referenced by both the source and the sink of the dependence. After executing the source of the dependence, the cache hit ratio can be improved by saving the associated reference window in the cache until after the sink has been executed. They estimate sizes of reference windows and suggest that program transformations can

be used to enable these windows to fit in the highest level of the memory hierarchy. Other studies [104] also address storage size limitations. Wolfe [105] uses a technique called *tiling*, which divides the iteration space of a loop into blocks or tiles with a fixed maximum size. Performance is improved when the tile fits into the highest level of the memory hierarchy. Wolfe and Lam [106] also describe program transformations to improve locality. These techniques can be used in conjunction with our approach.

Prefetching: The performance loss due to the latency of remote memory accesses has also been addressed using data prefetching which takes place during computation. For example, Gornish et al. [107] use compile-time analysis to determine the earliest point in the program when data can be prefetched, and evaluate the scheme using simulations. Hudak and Abraham [88, 89] also use compile-time techniques to overlap computation with memory accesses for iterative parallel loops and demonstrate performance improvement by execution time measurements on the BBN TC2000.

Software-assisted Cache Coherence: Studies on software-controlled caches are of interest to us because object-level placement requires the compiler or the applications programmer to specify data placement operations. Cheong and Veidenbaum [108, 109, 110, 111] and Cytron et al. [112] have proposed software schemes which keep caches consistent by using data dependence analysis to determine when cache lines need to be invalidated or written back to global memory. Our compile-time object-creation scheme which directs object-level placement also uses data dependence analysis to determine when it is necessary to make copies of each data element consistent. Unlike these schemes however, our method uses the data reference pattern in order to create and transfer objects between the local memories of the processors with the goal of maintaining data consistency as efficiently as possible.

False Sharing: Finally, we summarize related work in addressing the false sharing problem. Torellas et al. [47] discuss several solutions to solve the problem for cache lines including scalar expansion and record expansion. These techniques are specific to cache lines, depend on the cache line size, and might fragment the cache lines. Lam et al. [113] study the problem in the context of block algorithms and suggest

copying non-contiguous data to consecutive locations, but they do not discuss how this process can be automated by a compiler. Eggers and Jeremiassen [52] apply program transformations to eliminate false sharing of cache lines for certain types of shared data. But, in the absence of coherent caches, their indirection scheme leads to an extra memory access. Also, their work does not address false sharing of cache lines containing data shared by different sets of processors.

Another approach to solve the false sharing problem is to provide a weak consistency model which updates only those copies of a data element that will actually be used by using runtime information and programmer-specified synchronization operations. Examples are the delayed consistency for cache-line-level replication [50] and the lazy release consistency for page-level replication [51]. While the former requires additional hardware, the latter incurs extra runtime overhead. False sharing has been also been addressed by using programmer-specified information about the sharing characteristics of data objects in **SSVM** [23] and **Munin** [24]. While **SSVM** requires the programmer to specify the exact reference pattern of all objects, **Munin** requires the programmer to specify the nature of references to an object and determines the exact reference pattern during runtime. Our approach is to reduce the runtime overhead and eliminate the programmer effort by determining the data reference pattern at compile-time when possible. As discussed in Chapter 5, we develop object-creation methods which provide the information required by systems such as **SSVM**.

3.7 Summary

In this chapter, we motivated the need for the compiler to assist in placing data at all levels of the NUMA physical memory hierarchy. The compiler can do so by providing information about the reference pattern of the SVM. After identifying the factors that determine this reference pattern, we designed a compiler that provides this information by means of compile-time objects. We also developed algorithms to determine the information needed by our object-creation schemes that assist data placement. In the next two chapters, we discuss in detail our compile-time object-

creation schemes for the block-level and object-level types of placement, respectively.

CHAPTER 4

COMPILER-ASSISTED BLOCK-LEVEL PLACEMENT

In this chapter, we present our work on compiler-assisted block-level placement when providing a shared virtual memory (SVM) in a NUMA multiprocessor. First, we outline our object-creation scheme that creates compile-time objects containing the application's data. We then discuss how these objects can be used to assist block-level placement. Next, we present solutions to the other issues that arise when assisting block-level placement using objects created by our scheme. We then derive the compile-time overhead of applying our scheme. We conclude by discussing the performance improvement offered by our scheme.

4.1 Compile-Time Object-Creation

In order to assist block-level placement schemes, false sharing needs to be eliminated, which can be done by ensuring that all the data elements contained in a given block have the same reference pattern. Further, it is necessary to reduce the runtime overhead of re-evaluating placement decisions in order to adapt them to changes in the reference patterns. The overhead can be reduced by simplifying and when possible, specifying the reference pattern of blocks. These considerations motivate our object-creation scheme which assists block-level placement by creating compile-time objects using the data reference pattern, and we call it `OCRP`, an acronym for

Object **C**reation based on **R**eference **P**attern. In contrast, the **S**tandard method of **O**bject **C**reation lays out the application's entire data contiguously in the SVM without any regard to the data reference pattern, and we call it `OCS`.

Since we include the temporal variation in the reference pattern by determining each data element's reference pattern separately for various code segments, we also create objects individually for each code segment by applying `OCRP`. We assume that runtime requests for virtual memory allocation are serviced from distinct heaps for different processors, as proposed in [47] to address the false sharing problem. We also assume that private and temporary variables are placed in the local memory of the only processor that references them, and concentrate on object-creation for code and statically-declared variables. The type of a variable can be scalar, array, or synchronization. The type of a synchronization variable can be barrier, wait/signal or lock.

Heuristics for Object-Creation: `OCRP` creates objects using each variable's type and its reference pattern which are determined as outlined in Chapter 3. It allocates each processor's code to a distinct object, and uses the following heuristics to create objects for statically-declared data.

H1 Do not allocate different types of variables to the same object.

H2 Do not allocate different types of synchronization variables to the same object.

H3 Allocate scalar variables with identical reference pattern to a single object.

H4 Allocate elements of the same array variable with identical reference pattern to a single object.

H5 Allocate elements of the same array variable which are referenced by only one processor to a single object, irrespective of whether they are read or written by this processor.

H6 Allocate barrier synchronization variables of different parallel code segments to the same object.

H7 Allocate wait/signal synchronization variables with the same reference pattern to the same object.

H8 Allocate lock synchronization variables that protect data elements with the same reference pattern to the same object.

4.2 Assistance in Block-Level Placement

In this section, we discuss the manner in which block-level placement can be assisted by using the objects created for each code segment by applying **OCRP**.

False Sharing

When the data reference pattern for each code segment is exact, the objects created by **OCRP** are not falsely shared. Therefore, for each code segment, false sharing can be eliminated by laying out the elements of each object in a contiguous portion of the SVM and ensuring that objects do not share virtual pages. In addition to eliminating false sharing of pages as well as cache lines, such a scheme also facilitates prefetching because the objects created by **OCRP** have temporal locality. One method of implementing this scheme is *page padding*, in which the virtual memory allocated to each object is padded to a page boundary. This method requires that the page size be known at compile-time, and necessitates recompilation of the application when it is ported to a NUMA multiprocessor with a different page size. The other method is for the runtime mechanism to provide primitives to allocate virtual memory for objects of various sizes. Such primitives can easily be provided by operating systems such as **Mach** [57]. Later in this chapter, we present solutions to issues that arise due to mapping objects to the SVM in this manner. These issues include internal fragmentation and code generation for references to array variables whose elements are non-contiguous in the SVM.

Since objects are mapped to the SVM for each code segment individually, variables that are referenced in more than one code segment need special consideration. Typi-

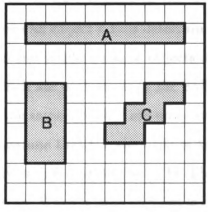

Figure 4.1. Examples of objects for elements of a two-dimensional array.

cally, synchronization variables are independent in each code segment and therefore, their values need not be propagated from one code segment to another. Most of the scalar variables are shared by all processors and have the same reference pattern in all the code segments. We propose that for both methods of eliminating false sharing, the remaining scalar variables be remapped by copying them from the objects of the previous code segment to those of the next code segment. We also use copy operations to remap array variables when page padding is used to eliminate false sharing. On the other hand, when the virtual memory for objects is allocated using runtime primitives, we remap array variables by requiring that the following additional primitives be provided: (1) *split*, which splits a single object into several objects, and (2) *combine*, which combines several objects into a single object. In general, after applying `OCRP`, operations to remap the various variables (copy, combine and split) are inserted between each pair of consecutive code segments.

Both the combine and the split operations need to know how the elements in the objects are to be combined or split, respectively. Our first approach to specifying this information is by a parameter [54]. For example, *interleave* in the primitive *split(input_object_id, interleave, interleave_factor, output_object_ids)*, specifies that the elements of the input object need to be allocated to the output objects in an interleaved fashion. Our second approach is to specify for each input and output object,

information about the layout of its elements in the SVM. We derive this second method from SSVM [23], where an object's *kind* determines the layout of its elements in the SVM, and sequential and stride kinds of objects are supported. A sequential object contains elements which are contiguous in the SVM, while a stride object is a set of sequential objects of the same size that are non-contiguous, but equally spaced in the SVM. For example, assume that the elements of the two-dimensional array in Figure 4.1 are mapped in row major form to the SVM. Then, A is a sequential object, and B and C are stride objects. When a data element is assigned to an object during object-creation, OCRP updates the kind of the object using the algorithm shown in Figure 4.2.

The objective of the combine operation is to create a consistent copy of the array variable in the local memory of the processor which executes the sequential code segment. Hence, its effect is equivalent to that of flushing the cache of all the processors at the end of a code segment in software-assisted cache coherence schemes such as [108]. Its implementation is more involved however because unlike the flush-cache operation which writes to consecutive locations in the global memory, it needs specification of the kind of all the objects that need to be combined. We propose several optimizations in using the combine operation. For example, the combine operation for an array is inserted only after a code segment in which it is modified and further, only if there is a change in the array's reference pattern in the following code segments. Moreover, only those elements of the array that are written are transferred during the combine operation.

Adaptive Placement Schemes

The manner in which OCRP creates objects allows specification of the placement information for each object, which includes the variable type as well as the nature of the reference pattern of its elements. If false sharing is eliminated using page padding, OCRP provides placement information for the virtual pages of each object. On the other hand, if false sharing is eliminated using runtime primitives, additional primitives need to be provided that allow OCRP to specify placement information for

```
/* An element's ID is its position in the SVM relative to its variable. */
  IF number of elements in object is zero /* newly-created object */
    obj_kind = sequential
    obj_vir_addr_index = elem_ID
    obj_num_elem = 1
    cur_seq_obj_num_elem = 1
  ELSEIF (obj_kind is sequential)
    IF ((obj_vir_addr_index + obj_num_elem) equals elem_ID)
/* next element in the sequential object */
      increment obj_num_elem
      increment cur_seq_obj_num_elem
    ELSE
      obj_kind = stride
      increment obj_num_elem
      obj_stride = (elem_ID - (obj_start_vir_addr + obj_num_elem))
      obj_seq_set_count = 2
      seq_obj_num_elem = obj_num_elem
      cur_seq_obj_num_elem = 1
      last_elem_ID = elem_ID
    ENDIF
  ELSEIF (obj_kind is stride)
    last_elem_ID = elem_ID
    increment obj_num_elem
    increment cur_seq_obj_num_elem
    IF (cur_seq_obj_num_elem > seq_obj_num_elem)
/* new sequential set of elements begins; check stride */
      IF ((last_elem_ID - seq_obj_num_elem + 1 +
        obj_stride) not equals elem_ID) /* unknown object kind */
        exit
      ELSE /* valid stride */
        increment obj_seq_set_count
        cur_seq_obj_num_elem = 1
        last_elem_ID = elem_ID
      ENDIF
    ENDIF
  ENDIF
```

Figure 4.2. Algorithm to update the kind of an object.

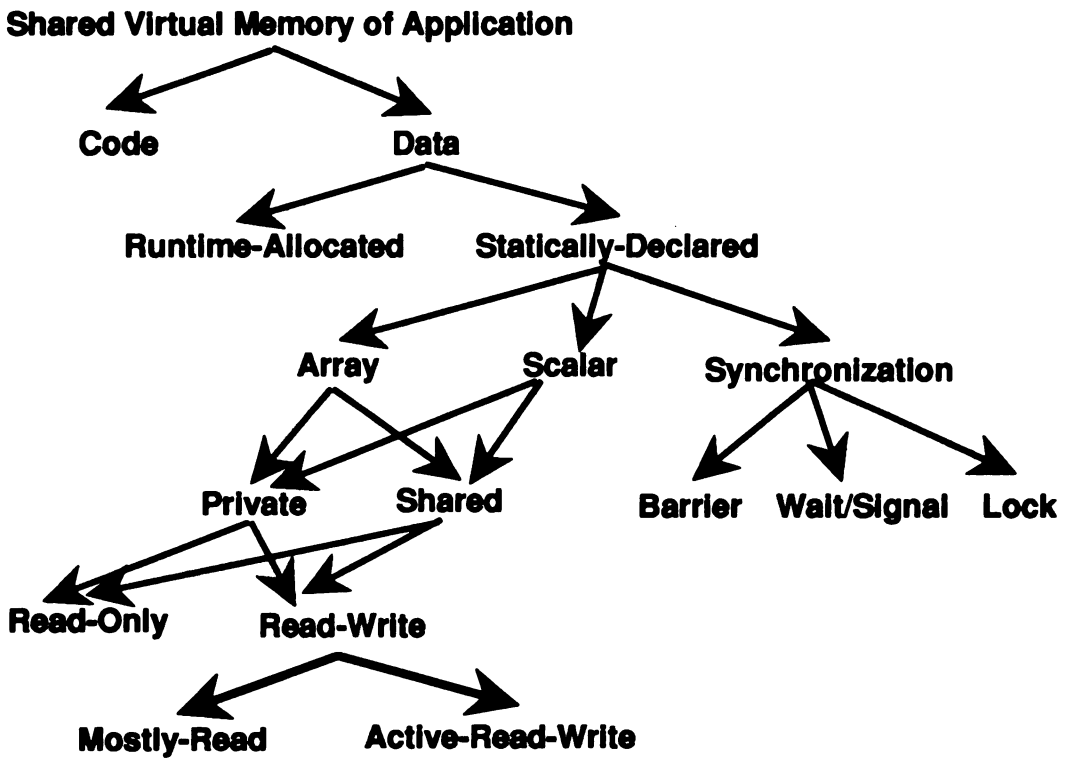

Figure 4.3. Placement information provided by OCRP.

each object. The placement information allows the runtime overhead of adaptive placement schemes to be reduced or in certain cases eliminated altogether.

OCRP specifies the variable type of an object to be scalar, array, barrier, wait/signal, or lock. The information it provides about an object's reference pattern depends on the data structure used to store the data reference pattern. We consider the case when the exact number of read and write references for each data element are available in its read and write counters, respectively. If the counters of only one processor of such an object are non-zero, it is a private object. If all the write counters of the object are zero, it is a read-only object. If the sum of all the read counters of the object is far greater than the sum of the write counters, it is read more often than written and is specified to be a mostly-read object. Finally, objects with almost identical, non-zero, read and write counters for the various processors are equally read and written by all processors and are specified to be active-read-write

objects. The placement information provided by OCRP is shown in Figure 4.3. When the reference pattern is represented using bits instead of counters, OCRP is still able to provide the same placement information except that it is unable to classify read-write objects into mostly-read and active-read-write objects.

Previous studies on block-level placement have made it straightforward to choose a placement scheme, given the placement information provided by OCRP. For example, a private object is best placed statically in the local memory of the processor that references it, while mostly-read and read-only shared objects can be replicated. Consistency need not be enforced for read-only shared objects and it is better not to migrate or replicate an actively shared read-write object. Barrier synchronization and wait-signal synchronization objects (shared actively in a fine-grain manner) are best placed statically in the local memory of one of the processors that references them.

Since the placement information is specified for each code segment, OCRP allows changes in placement strategies from one code segment to another. For example, an object might be shared between processors A and B in a given code segment, and processors C and D in the next code segment. Using the placement information provided by OCRP, the object is replicated in the local memory of A and B in the first code segment, and in the local memory of C and D in the next code segment. Similarly, an object referenced by processor A in a given code segment and by processor B in the next code segment is migrated from the local memory of A to that of B between these code segments.

In the case of conditional statements, the data reference pattern is determined separately for each case of a conditional statement. The remap operations and placement information are also specified conditionally for each such case. When the data reference pattern is inexact, the objects created by OCRP might be falsely-shared and so are the resulting virtual blocks. Further, the placement information provided by OCRP is not exact. Since OCRP does not specify when copies of data elements need to be updated, but only controls the mapping of the data elements to the SVM, applications compiled by applying OCRP execute correctly even in the presence of an

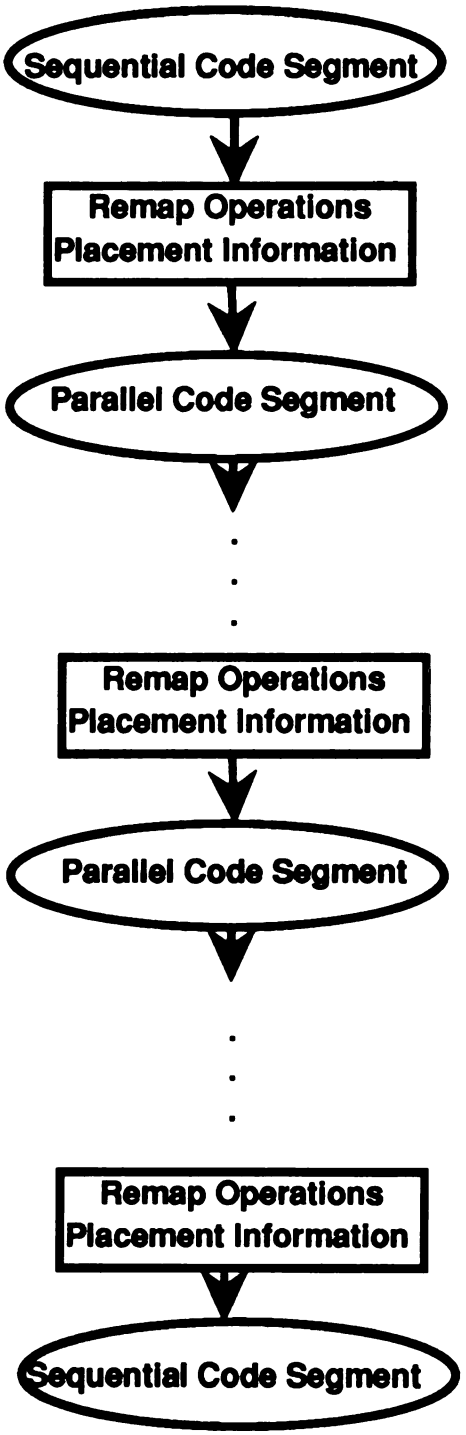

Figure 4.4. Parallel application after applying OCRP.

```
    DOSEQ S1
        Initialize A,B,C
S1  END DOSEQ
    DOALL S2 I = 1 TO n
        DOALL S3 J = 1 TO n
            DOSEQ S4 K = 1 TO n
                C(I,J) = C(I,J) + A(I,K) * B(K,J)
S4          END DOSEQ
S3      END DOALL
S2  END DOALL
    DOSEQ S5
        Output C
S5  END DOSEQ
```

Figure 4.5. Application **matrix multiplication**.

inexact reference pattern. In this manner, OCRP assists block-level placement at all levels of the memory hierarchy by creating virtual objects which are not falsely shared and which have temporal locality and by providing placement information for these objects. After applying OCRP, the parallel application is transformed as shown in Figure 4.4.

4.3 Compilation of Applications

In this section, we illustrate OCRP by applying it to certain applications. The first application we consider is **matrix multiplication**, and its typical parallel program is shown in Figure 4.5. For the sequential code segment S_1, OCRP creates three objects, one each for the array variables A, B, and C, and each variable's object contains all its elements. Let us assume that the parallel code segment represented by S_2, S_3, and S_4 is control-partitioned by allocating sets of consecutive values of the loop index

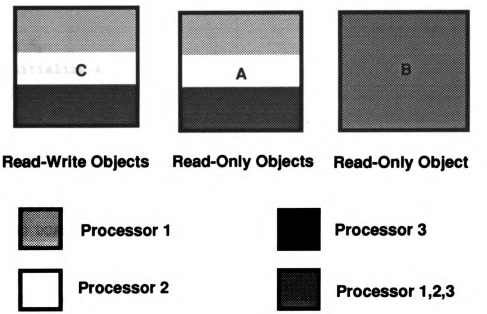

Figure 4.6. Objects created by OCRP for application **matrix multiplication**.

I to the various processors. For this case, OCRP creates a private read-write object for each processor containing the elements of the rows of C which are exclusively read and written by this processor. It also creates a private read-only object for each processor containing the elements of the rows of A which are exclusively read by this processor. Further, it creates a single shared read-only object for all the elements of B. Figure 4.6 illustrates object-creation in this manner when the parallel code segment is executed by three processors. For the sequential code segment S_5, OCRP creates a single object containing all the elements of C.

The next application we consider is the **parallel iterative solver** for a two-dimensional problem with a 5-point stencil, which is part of the workload for our experiments outlined in Chapter 2. For ease of reference, we repeat the typical parallel program of this application in Figure 4.7. Here, each iteration corresponding to loop S_2 uses the results of the previous iteration and hence must be executed sequentially. However, the iterations of the loops S_3 and S_4 can be executed in parallel. The parallel program consists of a sequential code segment S_1 for initialization, a parallel

```
   DOSEQ S1
      Initialize A
S1 END DOSEQ
   DOSEQ S2 K = 1 TO m
      DOALL S3 I = 1 TO n
         DOALL S4 J = 1 TO n
           A(I,J) = 0.25 * (A(I-1,J) + A(I+1,J) + A(I,J-1) + A(I,J+1))
S4       END DOALL
S3    END DOALL
S2 END DOSEQ
   DOSEQ S5
      Output A
S5 END DOSEQ
```

Figure 4.7. Application **parallel iterative solver**.

code segment represented by S_3 and S_4 that is executed m times (corresponding to the m iterations of loop S_2), and another sequential code segment S_5 for termination. **OCRP** creates a single object containing all the elements of array variable A, for each of the two sequential code segments S_1 and S_5. Let us assume that the parallel code segment represented by S_3 and S_4 is control-partitioned by assigning a different subset of the values for the I and J loop indices to the various processors. Each processor then computes the values of a rectangular partition of the array A for each iteration of loop S_2. For the parallel code segment, **OCRP** creates nine objects for each processor as shown in Figure 4.8. Here, the processors are labeled P_i, $1 \leq i \leq 9$, and the objects for P_5 are labeled O_i, $1 \leq i \leq 9$. O_1 contains elements written by P_5, and read by P_2, P_4, and P_5, O_2 contains elements written by P_5, and read by P_2 and P_5, and so on.

Another application we consider is the **recurrence relation**, and its typical parallel program is shown in Figure 4.9. **OCRP** creates a single object containing all

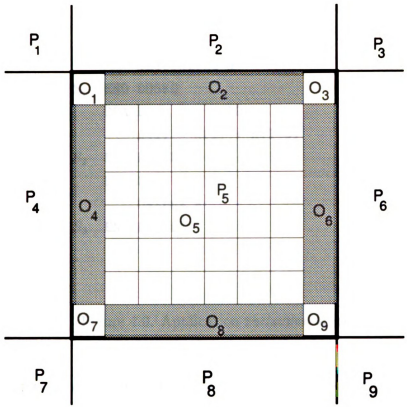

Figure 4.8. Objects created by `OCRP` for application `parallel-iterative-solver`.

the elements of array variable A for each of the two sequential code segments S_1 and S_3. The parallel code segment S_2 can be control-partitioned into independent execution sets, which contain the set of iterations: (1, 7, 13 ...), (2, 8, 14 ...), etc. For this case, `OCRP` creates a private read-write object for each processor containing the elements in its independent set.

Next, we illustrate `OCRP` for the application in Figure 4.10, which contains a DOACROSS loop with synchronization operations [114]. Assume that the iterations in the DOACROSS loop are control-partitioned such that each processor p executes the set of iterations starting from p until n, with a step equal to the number of processors. In this case, `OCRP` creates a single object for each array variable for the initialization code segment, and creates multiple objects for the parallel code segment based on the reference pattern of the data elements.

```
    DOSEQ S1
        Initialize A
S1  END DOSEQ
    DOALL S2 I = 6 TO n
        A(I) = A(I) + A(I-6)
S2  END DOALL
    DOSEQ S3
        Output A
S3  END DOSEQ
```

Figure 4.9. Application **recurrence-relation**.

Examples of array variables occurring in more than one code segment are A in Figure 4.7 and C in Figure 4.5. In the case of each of these variables, for the initialization code segment, a single object containing all its elements is created. The single object is then split into multiple objects for the parallel code segment. Assuming that the arrays are mapped to the SVM in row major form, array C is split into multiple sequential objects, while array A is split into a combination of sequential and stride objects. In both cases, the multiple objects are later combined into a single object for the termination code segment.

In the case of all the above applications, barrier synchronization is implicit between successive code segments. **OCRP** allocates all the barrier synchronization variables of an application to a single object. Wait/signal synchronization variables are used to enforce dependencies in the DOACROSS loop shown in Figure 4.10. Using the data reference pattern and the arguments to a given SEND operation, it is possible to determine which processors are involved in the wait/signal synchronization and would reference the corresponding wait/signal synchronization variable. **OCRP** applies **H7** to create objects for these variables.

```
    DOSEQ S1
        Initialize
S1  END DOSEQ
    DOACROSS S2 I = 1 TO n
        WAIT(CSYNC,I-1)
        A(I) = B(I) + C(I-1)
        SEND(ASYNC,I)
        D(I) = A(I) * 2
        WAIT(ASYNC,I-1)
        C(I) = A(I-1) + C(I)
        SEND(CSYNC,I)
        WAIT(CSYNC,I-2)
        E(I) = D(I) + C(I-2)
S2  END DOACROSS
    DOSEQ S3
        Terminate
S3  END DOSEQ
```

Figure 4.10. Application with a DOACROSS loop.

4.4 Internal Fragmentation

In order to assist block-level placement, the objects created by OCRP are not allowed to share virtual pages. The virtual pages of each object are mapped to one or more physical blocks (e.g., pages, cache lines) during runtime. When the size of the virtual page is large relative to the number of elements in the last virtual page of an object, internal fragmentation and wastage of virtual memory occurs. Similarly, when the size of the physical block is large compared to the number of elements in the last physical block for an object, internal fragmentation and wastage of physical memory occurs. In the extreme situation, it is impossible to satisfy the virtual and/or physical memory requirements of certain applications when they are compiled using OCRP.

Out of the factors influencing internal fragmentation, the block size is an architectural parameter, while the number of objects and the number of elements per object are determined by application characteristics such as the size and the reference pattern of its data. Hence, for a given block size, internal fragmentation occurs for certain applications and not for others. For example, internal fragmentation of virtual memory does not occur for the application **matrix multiplication** if each object in Figure 4.6 needs an integer number of pages. In the case of application **parallel iterative solver**, the occurrence of internal fragmentation for objects O_i, $i = 2,4,5,6,8$, which are shown in Figure 4.8, depends on the number of processors and the size of array variable A. Objects O_i, $i = 1,3,7,9$ in the same figure, however, contain just one element, irrespective of these factors, and are prone to internal fragmentation for all page sizes greater than one.

The extra physical memory needed to assist block-level placement can be compared to that needed to improve the availability and access time of data by replication. Such a space-time tradeoff is justified in all but the extreme situations of insufficient virtual and physical memory. To illustrate our claim, consider the application **matrix multiplication**, and let us assume that the page size is 4Kbytes, that each data element occupies four bytes, and that A, B, and C are all 32×32 array variables. Further, let us assume that there are p processors and that the parallel code segment is control-partitioned by assigning consecutive sets of values of the loop index I to these processors. OCS creates a single virtual page for each array variable, which contains all of its elements, and is shared by all p processors. On the other hand, OCRP creates p private objects, one per processor for array variables A and C, each containing elements referenced by this processor. It creates a single object for all the elements of B which is shared by all processors. When page-level replication is used, in both cases, p physical pages are allocated for each array variable, one each in the local memory of every processor. The amount of data transferred for OCS is more than that for OCRP. Further, OCRP offers a better runtime performance than OCS because it eliminates false sharing, while OCS incurs the overhead of maintaining consistency of its falsely-shared pages. Note, however, that though OCRP requires the same amount

of physical memory as that needed by OCS, it requires more virtual memory.

Solutions: In general, the problem sizes in numerical and scientific applications are large enough not to cause internal fragmentation of virtual and physical memory, when applying OCRP. Internal fragmentation is a problem under certain conditions however, and therefore, we propose schemes to help alleviate this problem. One solution is for the memory architecture to support pages of various sizes as in the MIPS R4000 [115]. The appropriate page size can then be chosen based on application characteristics and the available virtual and physical memory. Another solution is for the compiler to map objects of different variables with the same reference pattern to the same virtual page. When the reference pattern is represented by a read bit and a write bit for each processor, this comparison can be done inexpensively using a logical AND operation. If counters are used instead of bits, the comparison is more expensive. One solution is to convert the counter information into the bit format for this purpose. When applied to variables in the same code segment, this solution alleviates internal fragmentation of both virtual and physical memory. This solution can also be applied to variables belonging to different code segments, in which case it reduces the internal fragmentation of virtual but not physical memory, assuming that the size of the working set is determined by the memory required for each code segment. In this case, it is not necessary for the variables to have the same reference pattern because placement decisions are decided individually for each code segment. Modified versions of the combine and split operations are necessary when using this solution. For example, pages of an object containing variables referenced in the next code segment should not be deallocated during the combine operation before this code segment.

A third solution is to use a relaxed version of OCRP which reduces false sharing instead of eliminating it altogether. For example, a relaxed version of OCRP applied to the parallel code segment in application **parallel iterative solver** creates one object per processor containing elements that are written by the processor. A fourth solution, the effectiveness of which depends on application characteristics, is to reduce the number of processors allocated to the application, thereby increasing the number

of elements in objects which are prone to internal fragmentation. There is a tradeoff in this solution between the performance gain due to assisting block-level placement and the performance loss due to the reduced parallelism. For illustration of this solution, consider application **matrix multiplication**, and let m be the number of physical pages available, and e be the number of elements in array variables A, B, and C. Further, let us assume that there are p processors and that the parallel code segment is control-partitioned by assigning consecutive sets of values of the loop index I to these processors. Since only one object is created for B, the potential for internal fragmentation exists only for A and C. We need to allocate at least $(3 \times e/m)$ elements to each object of A and C in order to prevent internal fragmentation. While this solution helps in the case of all objects for application **matrix multiplication**, in the case of application **parallel iterative solver**, it helps in the case of objects O_i, $i = 2, 4, 5, 6, 8$, but not for objects O_i, $i = 1, 3, 7, 9$.

4.5 Code Generation for Array References

In order to assist block-level placement, the objects created by **OCRP** are not allowed to share virtual pages. Therefore, it is possible for elements of an array variable to be mapped non-contiguously in the SVM. Existing code generation algorithms assume contiguous storage of array elements in the SVM, and calculate the virtual address of the element referenced in a given array reference as follows [116]. Consider a k-dimensional array, where the subscript in dimension j can take values from 1 to d_j, and $1 \leq j \leq k$. The virtual address of the element referenced in an array reference $A(s_1, s_2, ..., s_k)$ is given by:

$$
\begin{aligned}
& (s_1 - 1)d_2\ d_3\ ...\ d_k\ +\ (s_2 - 1)d_3\ d_4\ ...\ d_k\ +\ ...\ + \\
& (s_{k-1} - 1)d_k\ +\ (s_k - 1) \\
= \ & (s_1\ d_2\ d_3\ ...\ d_k\ +\ s_2\ d_3\ d_4\ ...\ d_k\ +\ ...\ +\ s_{k-1}\ d_k\ +\ s_k) - \\
& (d_2\ d_3\ ...\ d_k\ +\ d_3\ d_4\ ...\ d_k\ +\ ...\ +\ d_k\ +\ 1) \\
= \ & (\ ...\ ((s_1\ d_2\ +\ s_2)d_3\ +\ s_3)\ ...\ )d_k\ +\ s_k\ -
\end{aligned}
$$

$$(d_2\ d_3\ ...\ d_k\ +\ d_3\ d_4\ ...\ d_k\ +\ ...\ +\ d_k\ +\ 1) \tag{4.1}$$

In general, each subscript s_k in an array reference is a function of all loop induction variables which are valid in its scope. For example, considering the array reference $A(I, K)$ in the parallel code segment in Figure 4.5, the loop induction variables which are valid in its scope are I, J, and K, and its subscripts are functions of I and K, respectively. With respect to a given loop depth and its induction variable, all other induction variables are constants. Hence, the virtual address of an element referenced in an array reference is a function of only the loop depth's induction variable. For example, with respect to loop depth S_4, I and J are constants, and K is the only variable. Thus, the virtual address of the element referenced in $A(I, K)$ is a function of only K. This property is used to move invariant factors outside a given loop depth when computing the virtual address using Equation 4.1. For example, in the case of loop depth S_4 and array reference $A(I, K)$, an address register is initialized to the virtual address of the element corresponding to the initial values of K and I, and is incremented by one to obtain the virtual address of elements referenced for successive iterations.

Since **OCRP** does not guarantee contiguous storage of array elements in the SVM, the algorithm mentioned above is inapplicable. We develop a new algorithm to generate code for array references when applying **OCRP**. Our algorithm is shown in Figure 4.11 and is applicable for a perfectly nested loop. Our algorithm exploits the fact that with respect to a given loop depth, each array reference is a function of a single loop induction variable. It divides the loop into several subloops such that within each subloop, all array references traverse a contiguous portion of the SVM. Therefore, the traditional code generation algorithm [116] can be used to generate code for each subloop. Between successive subloops, the address registers corresponding to each array reference with a break in contiguity are loaded appropriately.

We introduce some definitions to explain our new algorithm for code generation. We consider the subscripts of an n-dimensional array reference as representing the coordinates of an n-dimensional plane. Any point in this plane corresponds to a

```
/* The following algorithm is used to generate code for each processor.
The variable loop_index is initialized to the first value of the
loop induction variable assigned to the processor. MAXITERATIONS is the
maximum value of the loop induction variable. LASTITERATION is the
last value of the loop induction variable assigned to the processor.
Each array reference has: (1) an address register containing the virtual
address to be used in code generation; it is initialized to the virtual
address of the element corresponding to the initial value of loop_index
and (2) a variable valid_loop_index containing the loop index till
which the current address register contents are valid (i.e., a contiguous
portion of the shared-virtual-memory is traversed).*/

  Initialize loop_index
  prev_loop_index = loop_index
  FOR each array reference
    Determine type and direction of traversal
    Initialize address register and valid_loop_index
  END FOR
  WHILE loop_index ≤ LASTITERATION
    FOR each array reference
      IF loop index > valid_loop_index
/* break in contiguity; obtain next virtual address */
        address register = virtual address of next element on traversal
        valid_loop_index = {loop index corresponding
          to last element without break in contiguity}
        IF last contiguous region for the traversal
          valid_loop_index = MAXITERATIONS
        END IF
      END IF
    END FOR
    loop_index = minimum valid_loop_index
    create subloop for iterations from prev_loop_index till loop_index
    increment loop_index
    prev_loop_index = loop_index
  END WHILE
```

Figure 4.11. Code generation for array references when applying OCRP.

specific set of values for these subscripts and hence a given element of the array. We define a *traversal* for a given array reference as containing points in this plane that correspond to elements which are referenced due to this array reference. The following steps are needed for each array reference in order to apply our new algorithm: (1) determination of the type of the traversal and (2) creation of a virtual-address data structure for this traversal which contains a list of contiguous portions of the SVM that are traversed. We illustrate this process for the case of a two-dimensional array reference with subscripts (s_1, s_2), where $s_1 = a_1\, i_p + b_1$ and $s_2 = a_2\, i_q + b_2$. Here, a_1, b_1, a_2, and b_2 are integer constants, and i_p and i_q are loop induction variables. Let i_m be the loop induction variable corresponding to the loop depth of interest. We have four cases to consider:

Case A: $(i_p \neq i_m) \wedge (i_q \neq i_m)$

In this case, s_1 and s_2 are both constants within the loop depth of interest, and hence a single element of the array is referenced.

Case B: $(i_p = i_m) \wedge (i_q \neq i_m)$

In this case, s_1 is a linear function of i_m, while s_2 is a constant, and a given column of the array is traversed.

Case C: $(i_p \neq i_m) \wedge (i_q = i_m)$

In this case, s_1 is a constant, while s_2 is a linear function of i_m, and a given row of the array is traversed.

Case D: $(i_p = i_m) \wedge (i_q = i_m)$

In this case, both s_1 and s_2 are linear functions of i_m. Rearranging the equations for s_1 and s_2, we obtain $i_p = (s_1 - b_1)/a_1$ and $i_q = (s_2 - b_2)/a_2$, respectively. Substituting the condition for Case D viz., $i_p = i_q$, we obtain

$$a_2\, s_1 + a_1\, s_2 + (a_1\, b_2 - a_2\, b_1) = 0 \tag{4.2}$$

Hence, the traversal is a straight line which is parallel neither to the x nor the y axis. For the specific case when a_1 and a_2 are equal to 1, and b_1 and b_2 are equal to 0, the traversal is along the diagonal. The traversals corresponding to these four cases

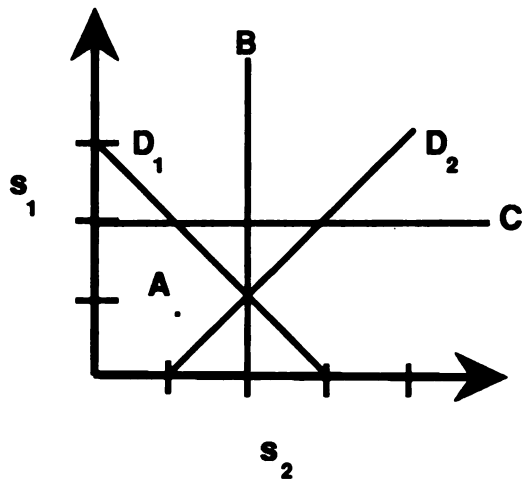

Figure 4.12. Traversals for an array reference with linear subscripts.

are shown in Figure 4.12, where D_1 and D_2 are the two possible subcases of Case D and they correspond to straight lines with negative and positive slopes, respectively. In this manner, the type of the traversal corresponding to an array reference can be determined.

Once the type of the traversal for an array reference is determined, the next step is to create a data structure that contains a list of contiguous portions of the SVM referenced for this traversal. Note that the manner in which this data structure is maintained should depend on the direction of the traversal. In general, the direction of the traversal is determined by the sign of the step of the loop induction variable. For example, the values of $(1, 20, 1)$ and $(20, 1, -1)$ for the parameters (lower bound, upper bound, step) of a loop induction variable I will cause forward and reverse traversals for the array reference $A(I)$, respectively. The direction of the column and row traversals of a two-dimensional array reference can be determined in a similar manner because for both cases, only a single subscript varies within the loop just as in the case of the one-dimensional array reference $A(I)$. The direction of the traversal for Case D depends on the sign of the step of the loop induction variable as well as that of a_1 and a_2. For example, when all three are positive, the traversal is along

D_2 in the upward direction. When a_1 and a_2 are positive and the step of the loop induction variable is negative, the traversal is along D_2 in the downward direction. The other cases can be derived similarly.

The virtual-address data structure for each array reference can be created by finding out the traversed elements and their virtual addresses. It is straightforward to determine the elements traversed for Case A and for the column and row traversals of Case B and C, respectively. The elements on the traversal for the case D_1 can be determined by varying s_2 from zero and obtaining the corresponding value of s_1 using Equation 4.2, until s_1 is zero. The elements on the traversal for the case D_2 can be determined by varying s_1 from zero and obtaining the corresponding value of s_2 using Equation 4.2, until the values of s_1 and s_2 correspond to the element in the top right corner of the array. For each set of values of s_1 and s_2, the corresponding element on the traversal can be computed. The virtual address of each element depends on its relative position within the object to which it is allocated by **OCRP**. Each object's start virtual address is either known at compile-time (when page padding is used) or obtained using runtime primitives. In this manner, the virtual-address data structure for each array reference can be created. This method can be extended to arrays of dimensions greater than two.

The number of times the address register for a given array reference is loaded is equal to the number of entries in the virtual-address data structure for this array reference. This number is influenced by the number of objects scanned in a traversal because each object causes a potential discontiguity in storage. Further, this number also depends on whether or not successive elements on a given traversal that belong to the same object have consecutive virtual addresses. In order to illustrate this fact, consider the application **matrix multiplication** and assume that each processor computes the values of elements belonging to a set of rows of matrix C. Let us assume that **OCRP** processes the elements of each array variable in a columnwise order. Consider the elements traversed in the row traversal corresponding to array reference $A(I, K)$ by one of the processors. **OCRP** allocates all these elements to a single object but they do not have consecutive virtual addresses. On the other hand, elements

traversed for the column traversal corresponding to $B(K, J)$ belong to a single object and have consecutive virtual addresses. The number of times the address register is loaded can be minimized if the order in which `OCRP` processes the elements of an array variable is tailored to the references of the array variable. For example, A and B in the application **`matrix multiplication`** can be processed rowwise and columnwise, respectively. This optimization however, cannot be applied when an array has references for both the row and the column traversals.

4.6 Compilation Overhead

The additional steps needed for each code segment when compiling using `OCRP` instead of `OCS` are:

1. Determination of the data reference pattern.
2. Object-creation.
3. Creation of virtual-address data structure for each array reference.
4. Code generation using our new algorithm.
5. Generation of remap operations and placement information.

Time Complexity: We already derived the time complexity for step 1 in the previous chapter. The algorithm used in step 2 is shown in Figure 4.13. Without loss of generality, we assume that in step 2, the elements of a k-dimensional array variable are processed by varying subscript s_p faster than subscript s_q, where $2 \leq p \leq k$, $1 \leq q \leq k-1$, and $p > q$. Further, the subscript in dimension j can take values from one to d_j, where $1 \leq j \leq k$. Also, the objects are organized as a separate list for each value of subscript s_1. For example, the elements of a two-dimensional array are processed rowwise and the objects are stored in a separate list for each row. Let e be the maximum out of the number of elements in each array variable and p be the number of processors allocated to the application. In the worst-case, each element

```
FOR each code segment
  FOR each variable
    Determine reference pattern of all elements
    FOR each element
      IF an object does not exist corresponding to its reference pattern
        create an object corresponding to its reference pattern
      END IF
      add element to object corresponding to its reference pattern
    END FOR
  END FOR
  FOR each object
    map contiguously in the SVM; disallow objects to share virtual pages
  END FOR
END FOR
```

Figure 4.13. Algorithm used by `OCRP` to create objects.

of an array variable is of a different reference pattern and is allocated to a distinct object, in which case there are e/d_1 objects in each list. When adding an element, on an average it is necessary to look up $e/(d_1 \times 2)$ objects. For each such lookup, it is necessary to compare the reference pattern of the object and the element. This comparison takes constant time when bits are used to represent the reference pattern. When counters are used, the time taken for comparison is $O(p)$. Hence, the worst-case time complexity for step 2 for an array variable is given by $O(e \times (e/(d_1 \times 2)) \times p)$.

The time complexity for step 3 depends on the number of distinct traversals and the number of elements in each traversal. For each element, its object is looked up from the information generated in step 2. For example, to generate the objects for the column traversal of a two-dimensional array, in the worst-case, e/d_1 objects need to be looked up to get each element's object. Further, as in step 2, $e/(d_1 \times 2)$ objects need to be looked up to insert this information in the objects for the column traversal.

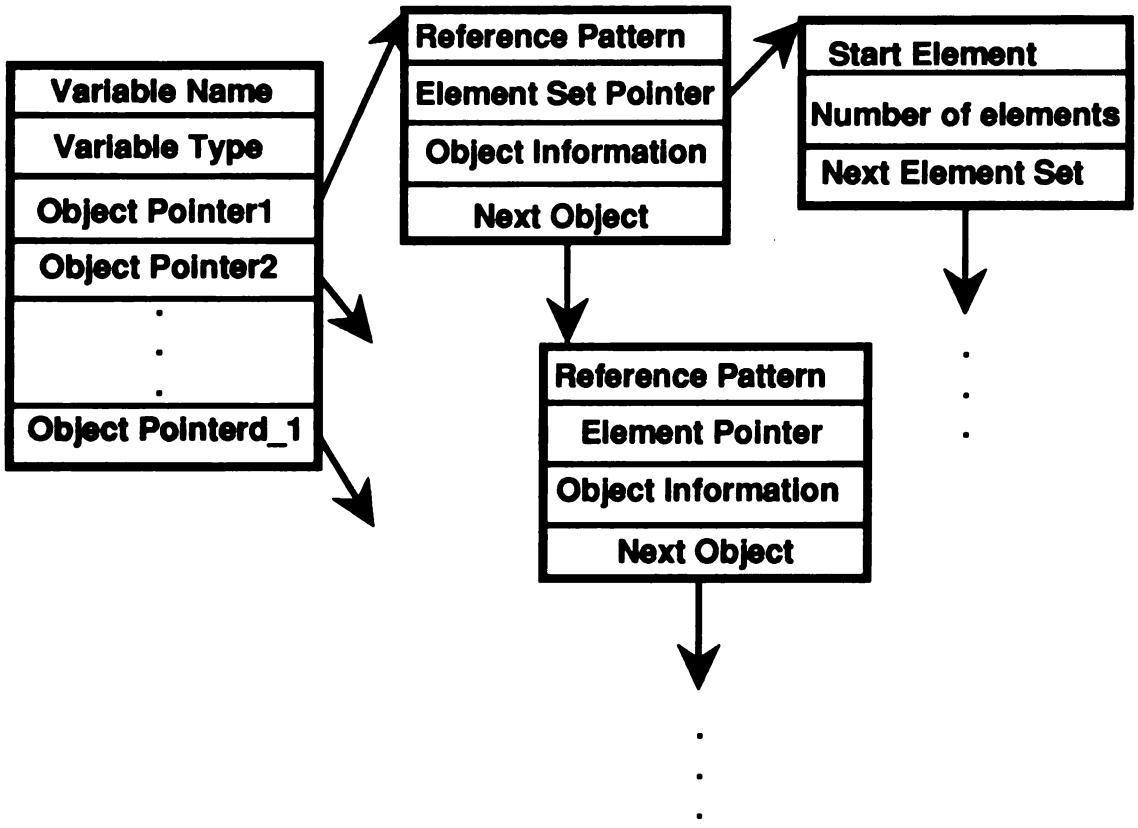

Figure 4.14. Data structure to store objects created by OCRP for an array variable.

Therefore, the worst-case time complexity is $O(e \times (e/d_1) \times (e/(d_1 \times 2)) \times p)$. The worst-case time complexity for Case D discussed earlier is $O(t \times (e/d_1) \times t/2 \times p)$, where t is the number of elements in the traversal. The time complexity for steps 4 and 5 depend on the number of array references and the number of entries in the virtual-address data structure for each such reference. Hence, it depends on the characteristics of the application and the method of control-partitioning.

Space Complexity: We already derived the space complexity for step 1 in the previous chapter. The data structure we use to store the objects for a given traversal is shown in Figure 4.14. The space complexity for steps 2 and 3 depend on the number of traversals and the number of entries in the virtual-address data structure for each traversal.

4.7 Performance of Applications

The code for OCRP is larger than that for OCS due to the placement information, calls to runtime primitives, and extra loads of address registers, and thus additional physical memory is needed.

We now discuss the various factors that affect the performance of applications when compiled with OCRP in relation to its performance when compiled using OCS. With respect to block size, a tradeoff between prefetching and false sharing exists for OCS because it prefetches and places data in terms of blocks. At one end of the spectrum, false sharing can be eliminated by placing data at an element-level but then there is no prefetching. Block-level data placement with larger blocks enhances prefetching but also increases the potential of false sharing. OCRP on the other hand creates virtual objects that provide the benefits of prefetching and at the same time eliminate false sharing. OCRP reduces the execution time of an application obtained with OCS by:

1. eliminating false sharing,
2. enabling prefetching of an object into physical memory before the code segment in which it is referenced, while with OCS, each physical block of the object is fetched individually on demand,
3. reducing the runtime overhead of determining the reference pattern of blocks, and
4. providing placement information.

In order to assist block-level placement, the objects created by OCRP are mapped to the SVM such that they do not share virtual pages. Due to this restriction, the following factors increase the execution time of the application obtained with OCS:

1. additional loads of address registers due to the need to use new algorithms for code generation,

2. additional system calls to obtain virtual addresses of objects in using the new algorithms for code generation, when the virtual memory for objects is allocated by runtime mechanisms,

3. possible inapplicability of certain compiler optimizations due to non-contiguous storage of array variables,

4. remap operations before each code segment, and

5. pages which are referenced in two consecutive code segments not being reused because of deallocation due to the remap operations between them.

Clearly, there is a tradeoff between these two set of factors. The factor that appears to significantly degrade performance in the case of OCRP are the remap operations. Recall that an array variable is remapped when its reference pattern changes between code segments. In such a case, processors reference a different set of its elements in each of these code segments. Therefore, even in the case of OCS, it might be necessary to fetch additional blocks. Further, the combine operation of OCRP deallocates physical blocks that were allocated for the previous code segment. In the case of OCS, these blocks are deallocated only when they are chosen for replacement. Blocks that are not deallocated are falsely-shared and contribute to the consistency overhead. Hence, the performance gain due to the elimination of false sharing and the prefetching and deallocation features of the remap operations when using OCRP may outweigh the performance loss due to the prevention of reuse of pages. Further, the runtime overhead of adaptive placement is less for OCRP than for OCS. Hence, we expect OCRP to perform better than OCS for a majority of applications. In Chapter 6, we quantitatively compare the performance of applications for the two schemes.

4.8 Summary

In this chapter, we developed a new object-creation scheme that assists block-level placement by eliminating false sharing and providing placement information. We also

proposed solutions to the issues of internal fragmentation and code generation that arise when applying our object-creation method. We derived the overhead involved in compiling applications using our scheme and discussed the various factors that affect the resulting runtime performance. Most of the factors that contribute to the compile-time and runtime overhead when applying our method are due to the remap operations. These operations are necessitated by the inherent nature of block-level placement schemes which invoke data placement operations for every reference to a block irrespective of whether these operations are actually required or not. In order to assist such strategies in placing data, the compiler needs to properly lay out data in the SVM and hence the remap operations. It appears that instead of laying out data to accommodate unnecessary placement operations, the compiler can be of better help if it is allowed to direct or specify when data placement operations are actually required. Such a facility is provided by object-level placement and in the next chapter, we present our work on compiler-directed object-level placement.

CHAPTER 5

COMPILER-DIRECTED OBJECT-LEVEL PLACEMENT

In this chapter, we present our work on compiler-directed object-level placement when providing a shared virtual memory (SVM) in a NUMA multiprocessor. We first discuss how compiler-directed object-level placement is able to use the compiler's knowledge about the data reference pattern more effectively than compiler-assisted block-level placement. Next, we present our object-creation scheme which achieves compiler-directed object-level placement. After illustrating our scheme for certain applications, we derive the compilation overhead involved in applying it. We then discuss the various factors that influence the performance of an application when using block-level placement, compiler-assisted block-level placement, and compiler-directed object-level placement. We conclude by presenting the results of our study which models the execution of an application when object-level placement is provided.

5.1 Motivation

In the previous chapter, we considered block-level placement schemes that invoke data placement operations whenever each block is referenced. For example, when block-level replication and sequential consistency is provided, whenever a write occurs, all blocks containing the referenced data element are made consistent. On the other hand, when release consistency is provided, whenever a release operation occurs, all

blocks which are referenced after the corresponding acquire operation and before this release operation, are made consistent. The performance of such schemes suffers when blocks are falsely-shared and when their reference patterns change rapidly. We provided compiler assistance to address these problems by using our object-creation scheme **OCRP**.

In order to assist block-level placement schemes, objects created by **OCRP** for each code segment are not allowed to share virtual pages. This restriction leads to performance loss due to certain factors. For example, array variables which are split into multiple objects need additional virtual memory to map these objects. Further, when their elements are mapped non-contiguously to the SVM, there is a loss of efficiency in the code generated. Array variables with a different set of objects in distinct code segments need to be remapped between these code segments. Also, internal fragmentation of virtual and physical memory occurs when there are few elements in the last block for an object relative to the block size.

Block-level placement schemes are unable to effectively use the information the compiler knows about the data reference pattern because they (1) invoke data placement operations on every reference to a block irrespective of whether these operations are actually required or not and (2) provide data placement in terms of blocks rather than objects. The compiler's knowledge about the data reference pattern can be used more effectively when it specifies when data placement operations are necessary, and also when it specifies them at the level of an object instead of a block. For example, this knowledge is best used by systems such as **SSVM** [23], which entrust the applications programmer with the responsibility of the placement of data in terms of objects.

SSVM does not allow the objects specified by the applications programmer to overlap and requires them to entirely cover the region of the SVM belonging to the application. In addition to the reference pattern, the information about each object consists of its kind, base, size, skip, and count, which together specify the layout of its elements in the SVM. As discussed in Chapter 4 in the context of combine and split operations, the kind of an object can be sequential or stride. The base of an

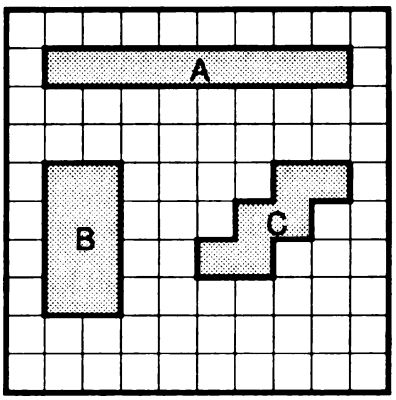

Figure 5.1. Examples of objects for elements of a two-dimensional array.

object is its start virtual address, and the size is the size of the first element of the object (which is the whole object for the sequential case or the first element in the stride case) in bytes. The skip of an object is the number of bytes to skip before the next element in the stride case, and the count is the number of times to repeat the size and the skip elements in the stride case.

For example, assume that the elements of the two-dimensional array in Figure 5.1 are mapped in row major form to the SVM, and that each element needs one byte of storage. Then, A is a sequential object with size=8, B is a stride object with size=2, skip=8, and count=4, and C is a stride object with size=2, skip=7, and count=3. Objects are placed in physical memory by copying them to interested processors (as specified by the reference pattern of the object) on detecting programmer-specified object-synchronization operations which are similar to release operations. These operations also update the address map for the virtual pages of the object. We extend this concept and propose that when a single physical copy of an object is maintained as is done for actively-shared objects, object-synchronization operations only update the address map. Note that the single copy is in local memory and not in cache, because in general, each processor cannot directly reference the cache of the other processors. The object-synchronization operations can take place asynchronously between processors, through a centralized server processor, or by means of a set of

distributed server processors.

SSVM therefore uses programmer-specified information to eliminate the performance loss due to false sharing and also to prefetch objects before their use. Note that objects are allowed to share virtual pages, and only physical blocks containing updated elements of an object are allocated. For example, only pages or cache lines containing at least one element of the stride object C in Figure 5.1 are allocated in the local memory or the cache respectively, of the processor in which object C is updated. Since all physical blocks containing at least one referenced element is allocated in a processor's local memory, it is possible that portions of certain physical blocks are unreferenced. Note that the restriction in **SSVM** that disallows objects to overlap is unnecessary, and as we show in a later section, it leads to a loss in performance. Hence, our compile-time object-creation scheme allows objects to overlap and provides the programmer-specified information required by **SSVM**.

5.2 Compile-Time Object-Creation

Similar to **OCRP**, our object-creation scheme creates objects based on the type and the reference pattern of each variable. The type of each variable is determined as in the case of **OCRP**, and the data reference pattern for each code segment is computed using one of the algorithms outlined in Chapter 3. Our scheme uses the same heuristics as those used by **OCRP** to create objects for synchronization variables. As outlined below, it uses a different algorithm however to create objects for array and scalar variables. Since the case when objects are not replicated is a subset of the replication case, we restrict our discussion to the latter. Further, in order to simplify our presentation, we assume that the copies of data elements allocated in local memory or the cache are not replaced. It is easy to extend the discussion to the case when replacements occur and the data is fetched from the next level of the memory hierarchy.

Goals: Object-creation for array and scalar variables is motivated by the following goals:

G1 Guarantee the following conditions: (1) a copy of a data element should be

updated only if it will be read and (2) before any data element's copy is read, it should be consistent with the copy last written.

G2 Create objects that satisfy **G1**, and allow them to overlap if necessary to minimize the time taken to update objects, provided correct execution of the application is guaranteed.

G3 Specify information about each object.

G4 Determine when object-synchronization operations are needed.

G1 determines when placement operations are necessary, and the first step to satisfy it is to determine when it is necessary to update data elements. Clearly, a data element needs to be updated only if it is read but not last written by the same processor. We consider the updates necessary for each code segment separately, and classify these updates into those which are required before and those that are needed during the execution of the code segment. The data elements which need to be updated before the execution of a given code segment are determined as follows. A data element in a given code segment can be: (1) neither read nor written, (2) only written, (3) only read, (4) first read, and (5) first written.

The first three cases can be detected using the reference pattern of the data element. The data element need not be updated for cases 1 and 2, and for case 3, it needs to be updated from the copy which was last written in one of the previous code segments. The data element needs to be updated similarly for case 4, and it need not be updated for case 5. In order to distinguish between cases 4 and 5 however, the read and write references to the data element within the code segment need to be ordered, which can be done using the data dependence graph. Alternately, the data element can be updated for case 5 as well because such an update does not cause incorrect execution of the application. A data element therefore needs to be updated before a given code segment if it is read in the code segment.

The next step in satisfying **G1** is to determine which copy of the data element was last written and by which processor. The barrier between successive code segments

ensures that reads and writes of a given code segment occur before those in the following code segments. Further, as is true in most applications, we assume that within a given code segment, every data element is written by at most one processor. Hence, data read in a given code segment should be updated from the code segment in which it was last written. This code segment as well as the processor which writes the data element can be determined using the reference pattern of the data element in the previous code segments.

In addition to making sure that each data element has the correct value before the execution of a code segment begins, **G1** also requires that they have the correct values during the execution of the code segment. Within a sequential code segment or a parallel code segment containing DOALL loops, it is not necessary to update a data element which is written and then read because both these references belong to the same processor. On the other hand, if these references occur within a parallel code segment containing DOACROSS loops, they could belong to different processors, in which case, updates to data elements are needed within the code segment. Such updates can be added before synchronization operations, which are used by parallelizing compilers to enforce dependencies within DOACROSS loops [117, 118, 114]. Each data element is updated from the only processor that writes it to one or more processors which read it, and these processors are determined from the data element's reference pattern.

Once the information about data elements that need to be updated are determined in this manner, the next step is to create objects such that these updates are carried out efficiently. Unlike block-level placement, object-level placement allows physical blocks to be falsely-shared. Therefore, the restriction which is present in both **OCRP** and **SSVM** that disallows objects to overlap is unnecessary. **G2** removes this restriction and allows the use of optimizations that help minimize the object update time. The optimization we use is to create a single object for all the elements of a given variable that need to sent from one processor to another. **G3** ensures that the information needed to carry out the object-synchronization operations properly is provided. Each object's reference pattern is recorded when it is created, and the other information

Table 5.1. Inexact reference pattern and OCDEP.

Exact Reference Pattern	Inexact Reference Pattern	Comment
Written	Not Written	N/A
Read	Not Read	N/A
Not Read	Read	Updates harmless, but unnecessary
Not Written	Written	Incorrect execution if written element is used in update

can be determined by the algorithm outlined in Chapter 4 which identifies sequential and stride objects. **G4** can be satisfied by inserting object-synchronization operations in the processor that writes each of the objects.

We refer to the sequence of read and write references to a data element by the various processors as its *data exchange pattern*, and it is determined by using the data element's reference pattern in the various code segments. Since our object-creation scheme creates objects based on the data exchange pattern, we call it **OCDEP**, an acronym for **O**bject **C**reation based on **D**ata **E**xchange **P**attern. As discussed in Chapter 3, when array references cannot be resolved, the reference pattern of related data elements is inexact. Considering the reference pattern to a data element by a given processor, the four possible combinations of values for the inexact and the exact reference pattern are listed in Table 5.1. Since we assume that the reference pattern is estimated conservatively, two of the cases are inapplicable. **OCDEP** specifies when copies of data elements are updated, and therefore an inexact data reference can sometimes cause incorrect execution of the application, as shown in Table 5.1.

In related work, other researchers have used software-based schemes to keep caches consistent when cache-line-level replication is provided. For example, Cheong and Veidenbaum [108, 109, 110, 111] have proposed three schemes. Two of these schemes ([108] and [109, 110]) assume that global memory always has the correct copy, and use compile-time dependence analysis in order to selectively invalidate cache lines so that the correct copy is fetched from global memory on the next reference. Since these

schemes do not make use of the data reference pattern, they cannot determine which processor last wrote an obsolete cache line. Hence, they unnecessarily invalidate cache lines that were written by the same processor which is currently reading it. The third scheme [111] addresses this problem by associating version numbers with each cache line, and these are updated on each write during runtime. A cache line is invalidated and its latest copy is fetched from global memory only when its version number is less than the current version number for the variable. Additional hardware is however required to implement the second and the third schemes. Cytron et al. [112] also propose a scheme to manage caches in software using compile-time dependence analysis which is similar to the second scheme discussed above.

OCDEP also uses dependence analysis to determine when consistency operations are necessary, but differs from the other approaches in the manner in which it makes data consistent. It uses the data reference pattern to determine which processor has the latest copy and therefore avoids the problems of the first two schemes mentioned above. It then creates objects and inserts object-synchronization operations so that data elements are made consistent in an efficient manner. Also, it transfers data in terms of objects not blocks, and hence, does not transfer unreferenced data elements such as those in falsely-shared blocks, and further, it facilitates prefetching. Under conditions of sufficient physical memory, it allows a distributed exchange of data among the processors, while in the other schemes all processors fetch data from global memory.

In the case of DOACROSS loops, with the other schemes, before executing the wait synchronization operation, a processor invalidates its cache line containing the data it requires. Other data elements in this cache line which have the current value are also invalidated in this process. When the waiting processor is signaled by another processor, the cache line containing the correct value of the variable is fetched from global memory on reference. In contrast to this pull-based approach, **OCDEP** uses a push-based approach where the data is pushed into the local memory of the waiting processor using object-synchronization operations. In this manner, at the cost of additional compilation overhead, **OCDEP** has the potential of achieving a

```
     DOSEQ S1
         Initialize A,B,C
S1   END DOSEQ
     DOALL S2 I = 1 TO n
         DOALL S3 J = 1 TO n
             DOSEQ S4 K = 1 TO n
                 C(I,J) = C(I,J) + A(I,K) * B(K,J)
S4           END DOSEQ
S3       END DOALL
S2   END DOALL
     DOSEQ S5
         Output C
S5   END DOSEQ
```

Figure 5.2. Application **matrix multiplication**.

better performance than the schemes mentioned above and in addition, it does not require extra hardware.

5.3 Compilation of Applications

In this section, we illustrate OCDEP by applying it to certain applications, and in all cases we refer to the single processor that executes the initialization (termination) code as the initialization (termination) processor. The first application we consider is **matrix multiplication** shown in Figure 5.2. We assume that the parallel code segment represented by S_2, S_3, and S_4 is control-partitioned by allocating sets of consecutive values of the loop index I to the various processors. Since the only parallel code segment contains DOALL loops, data elements need not be updated within code segments. They need to be updated before every code segment except the first one containing the initialization code. For the parallel code segment, OCDEP

```
   DOSEQ S1
      Initialize A
S1 END DOSEQ
   DOSEQ S2 K = 1 TO m
      DOALL S3 I = 1 TO n
         DOALL S4 J = 1 TO n
           A(I,J) = 0.25 * (A(I-1,J) + A(I+1,J) + A(I,J-1) + A(I,J+1))
S4       END DOALL
S3    END DOALL
S2 END DOSEQ
   DOSEQ S5
      Output A
S5 END DOSEQ
```

Figure 5.3. Application **parallel iterative solver**.

creates one object for each processor containing the elements of the rows of C which are exclusively read and written by this processor. It also creates one object for each processor containing the elements of the rows of A which are exclusively read by this processor. Further, it creates one object per processor containing all the elements of B. These objects are updated from the initialization processor to all the processors executing the parallel code segment. For the code segment containing the termination code, **OCDEP** allocates the elements of C which are computed by each processor to a distinct object. The objects thus created are updated from these processors to the termination processor.

Next, we consider object-creation by **OCDEP** for the application **parallel iterative solver** shown in Figure 5.3. Let us assume that the parallel code segment represented by loops S_3 and S_4 is control-partitioned by assigning a different subset of the values for the I and J loop indices to the various processors. In other

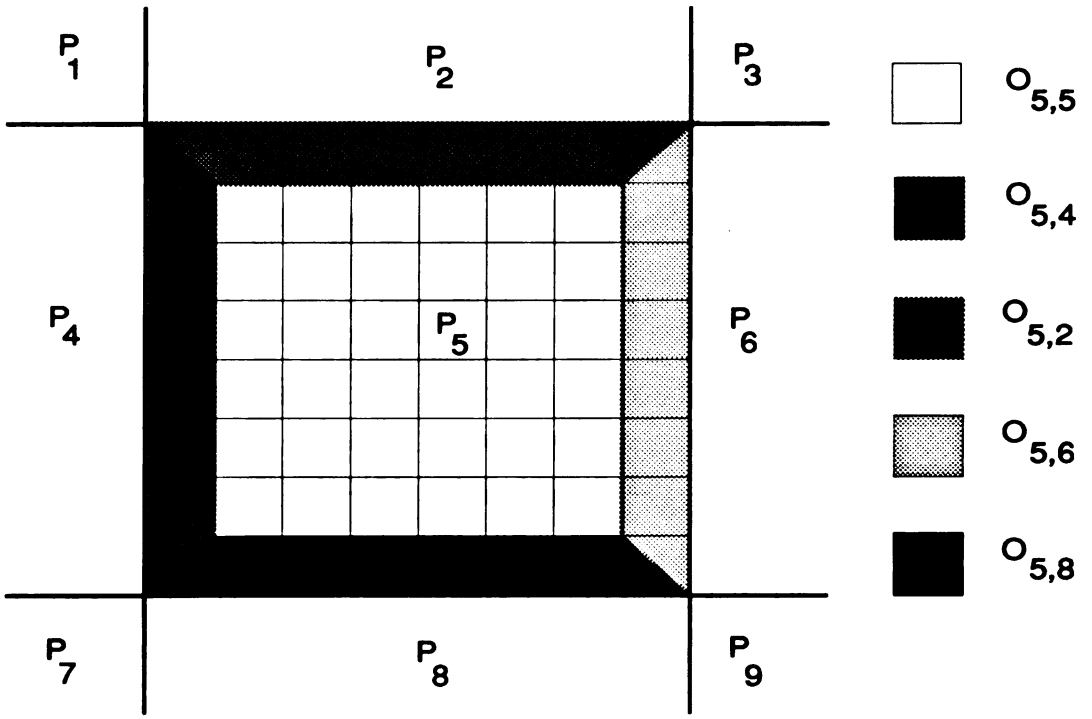

Figure 5.4. Objects created by OCDEP for application **parallel iterative solver**.

words, each processor computes the values of a rectangular partition of the array A for each iteration of loop S_2. As in the application **matrix multiplication**, all parallel code segments contain DOALL loops, and therefore, data elements need to be updated only before each of the code segments, except the first one containing the initialization code. For the parallel code segment corresponding to the first iteration of loop S_2, OCDEP creates one object per processor containing all the elements in its partition, and these objects are updated from the initialization processor to all the processors executing the parallel code segment.

The objects created by OCDEP for each of the following parallel code segments for an example processor P_5 are $O_{j,k}$, $j = 5$, $k = 2,4,6,8$, as shown in Figure 5.4. Here, the array elements are represented by rectangles, the different shaded regions represent distinct objects, and the number of shades in a rectangle represents the number of objects containing the corresponding element. Each object $O_{j,k}$ contains

```
     DOSEQ S1
         Initialize
S1   END DOSEQ
     DOACROSS S2 I = 1 TO n
         WAIT(CSYNC,I-1)
         A(I) = B(I) + C(I-1)
         SEND(ASYNC,I)
         D(I) = A(I) * 2
         WAIT(ASYNC,I-1)
         C(I) = A(I-1) + C(I)
         SEND(CSYNC,I)
         WAIT(CSYNC,I-2)
         E(I) = D(I) + C(I-2)
S2   END DOACROSS
     DOSEQ S3
         Terminate
S3   END DOSEQ
```

Figure 5.5. Application with a DOACROSS loop.

elements that are written by P_j and read by P_k. For example, the region shaded black is $O_{5,8}$, and contains elements written by P_5 and read by P_8. Further, objects are allowed to overlap, and there is no consistency problem because elements shared among overlapping objects are updated in different processors. For example, the element in the top left corner of array A is allocated to $O_{5,2}$, $O_{5,4}$, and $O_{5,5}$. Finally, for the sequential code segment containing the termination code, **OCDEP** creates one object per processor containing all elements in its partition. The objects thus created are updated from these processors to the termination processor.

Next, we illustrate **OCDEP** for an application where data elements need to be updated both before and within code segments. We consider the application in Fig-

ure 5.5, which contains a DOACROSS loop with synchronization operations [114]. In order to update data elements before each code segment, **OCDEP** creates objects as explained earlier for the other applications. In addition, it inserts object-synchronization operations before each SEND operation in order to update the copy of the corresponding data element in the local memory or cache of the processors waiting for it. The processor that writes the element and those that read it are determined using the data element's reference pattern. For example, assume that the iterations in the DOACROSS loop are control-partitioned such that each processor p executes the set of iterations starting from p until n, with a step equal to the number of processors. In this case, before the first processor executes SEND(ASYNC,1), **OCDEP** ensures that the copy of A(1) is updated in the second processor's local memory or cache.

Next, we illustrate **OCDEP** for an array variable with different reference pattern in distinct code segments. An example is the variable A which appears in the application **multi-code-segment-array** shown in Figure 5.6. Here, all elements of A are read in loops S_1 and S_3, and a fraction of the elements are written in loops S_2 and S_4. The loops are partitioned by allocating consecutive sets of iterations to various processors. Note that the lower bound of the loop index in loops S_1 and S_2 is different from that in loops S_3 and S_4, thereby making the data reference patterns different. **OCDEP** creates one object per processor containing all the elements it reads in loop S_1, and these are updated before loop S_1 from the initialization processor. No objects are updated before loops S_2 and S_4. Each element which is read in loop S_3 and written in loop S_2 needs to be updated from the corresponding processor. Elements which are not written in loop S_2 and for which a local copy does not exist need to be updated from the initialization processor. **OCDEP** creates objects to satisfy these conditions using the data reference pattern of these loops and applying the algorithm shown in Figure 5.7.

5.4 Compilation Overhead

The algorithm used by **OCDEP** to create objects for updating data elements before each code segment is shown in Figure 5.7. The objects created by **OCDEP** that need to

```
/* All referenced elements of A are read in loops S1 and S3.
Element A(I) is written in loops S2 and S4 if
I mod (WF * e) = 0 */

     DOSEQ
          Initialize
     END DOSEQ
     DOALL S1 I = 0 TO e-1
          ...  = A(I)
          .
          .
          .
S1   END DOALL
     DOALL S2 I = 0 TO e-1
          A(I) = ...
          .
          .
          .
S2   END DOALL
     DOALL S3 I = k TO e-1
          ...  = A(I)
          .
          .
          .
S3   END DOALL
     DOALL S4 I = k TO e-1
          A(I) = ...
          .
          .
          .
S4   END DOALL
     DOSEQ
          Terminate
     END DOSEQ
```

Figure 5.6. Application **multi-code-segment-array**.

```
/* The following algorithm creates objects that need to be updated before
each code segment. Each object for a given code segment contains
elements which are read and written by a unique set of processors.
rproc denotes a processor that reads the data element in the
current code segment. pwproc denotes the processor that last
wrote the data element in one of the previous code segments. It is
assumed that all elements are written at least once before being read. */

  FOR each data element
    FOR each code segment
      FOREACH processor rproc that reads the element
        IF not (local_copy(rproc))
          IF (rproc ≠ pwproc)
            IF object(codeseg, var, rproc, pwproc) not present
              Create object(codeseg, var, rproc, pwproc)
            ENDIF
            Add element to object(codeseg, var, rproc, pwproc)
          END IF
          local_copy(rproc) = TRUE
        END IF
      END FOREACH
      IF the data element is written
        pwproc = processor that writes the element
        local_copy(pwproc) = TRUE
        FOREACH processor proc not equals pwproc
          local_copy(proc) = FALSE
        END FOREACH
      ENDIF
    END FOR
  END FOR
/* Algorithm above has been applied to all variables for all code segments */
  FOR each object
    Update information
    Synchronize before code segment in processor wproc
  END FOR
```

Figure 5.7. Algorithm used by OCDEP for updates before a code segment.

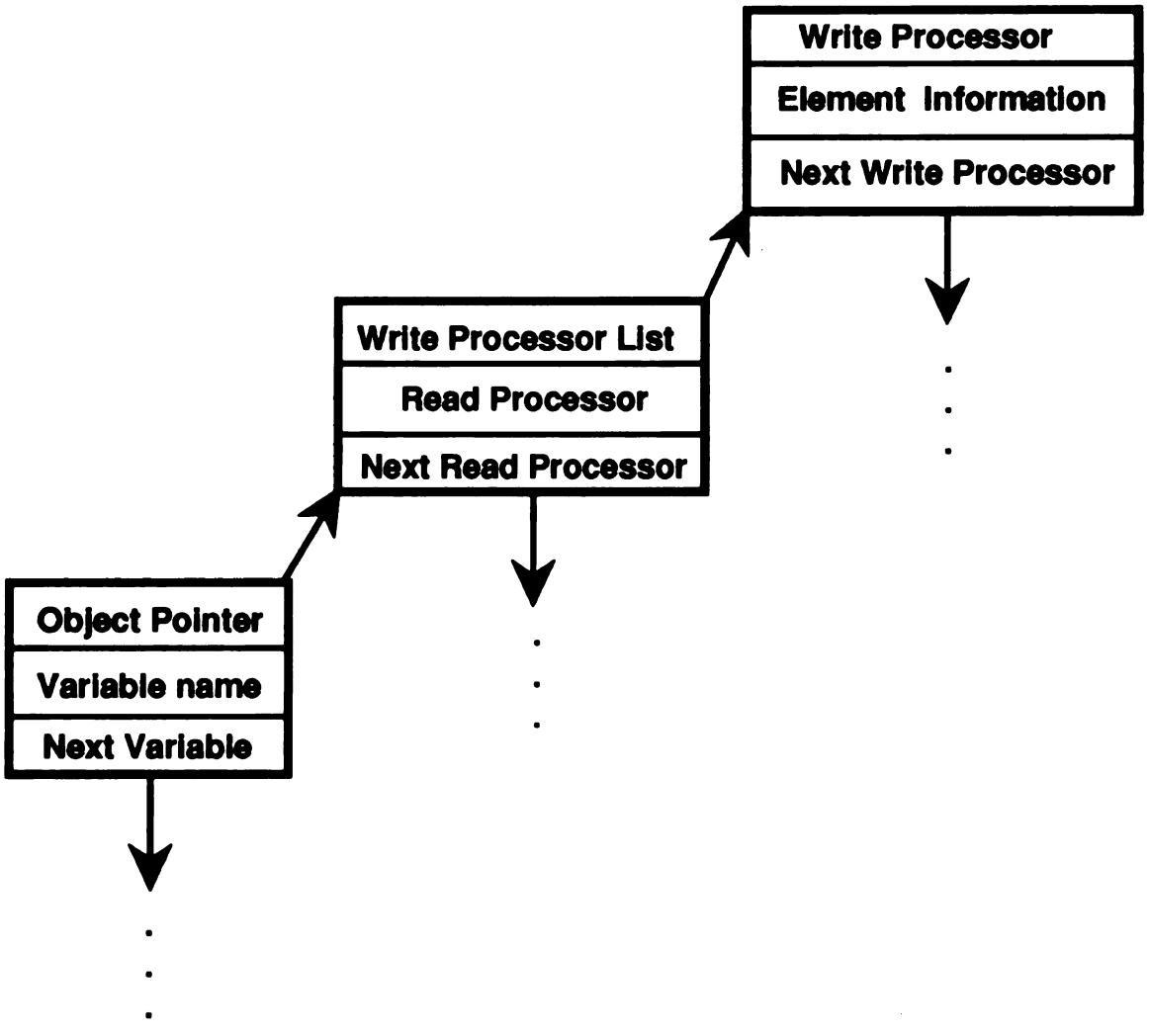

Figure 5.8. Object list for OCDEP.

be updated before a given code segment are organized as shown in Figure 5.8. The worst-case time complexity of the algorithm in Figure 5.7 for each array variable is $O(e \times s \times p \times (\log p)^2)$. The worst-case space complexity is $(2 \times e \times p \times s)$ bits for storing the reference pattern, and $O(p^2)$ for the object list in Figure 5.8.

5.5 Performance of Applications

In this section, we provide a qualitative comparison of the performance of an application when compiled using each of the object-creation schemes OCS, OCRP, and OCDEP. The experiments we conduct to compare them quantitatively are covered in the next chapter. We assume that data is replicated and sequential consistency is maintained for the case of block-level placement. For a given partitioning, the components of the execution time which are spent on processing, memory references, and synchronization are the same for all the three schemes. They however differ in the time spent in transferring data and keeping it consistent. We consider three cases: (1) OCS with block-level placement, (2) OCRP with block-level placement, and (3) OCDEP with object-level placement.

OCS maps the application's data to the SVM without any consideration of the data reference pattern and does not provide placement information. The time spent in transferring data and keeping it consistent depends on the block size. Performance loss occurs due to false sharing and the inability to adapt to rapidly changing reference patterns. OCRP creates objects which are not falsely-shared and which have temporal locality. Data transfer messages are fewer than in the case of OCS because of the prefetching facilitated by the temporal locality of the objects. By disallowing these objects from sharing virtual pages, false sharing of the corresponding physical blocks is eliminated. Hence, the number of consistency operations is fewer than that for OCS. The restriction of not allowing objects to share pages however leads to other factors such as remap operations and code generation which might result in a loss in performance. Since object-level placement allows objects to share pages, these factors are absent. Further, since data elements are updated only when necessary rather than on every write or release operation to a block, the number of consistency operations are fewer for OCDEP when compared to OCRP. We therefore expect OCDEP with object-level placement to perform better than either OCS or OCRP with block-level placement.

The virtual and physical memory requirements for OCS and OCDEP are the same.

OCRP requires additional virtual memory in order to ensure that objects do not share virtual pages. It is possible for portions of the physical blocks to be unused in the case of OCS and OCDEP. The temporal locality of objects created by OCRP ensure that each physical block except the last one are always fully utilized. Internal fragmentation of the last block however occurs under certain conditions and might increase the virtual and physical memory needs considerably.

5.6 Application Execution Model

In the next two sections, we present the results of our initial study [55, 56] on object-level placement. In this study, we model the execution of the application **parallel iterative solver** when object-level placement is provided. We consider the case when objects are allowed to overlap and that when they are not, and we refer to them as OCDEP and OCDEP_NOVLP, respectively. Our model allows us to study the various factors affecting performance when object-level placement is provided, and it also helps us to determine the performance improvement obtained by allowing objects to overlap. We choose this particular application because it has been studied extensively in the literature and its loop construct occurs in several important applications including the area of image processing [119]. For example, Reed et al. [68] have studied several partitioning strategies for this application including rectangular partitions which are shown in Figure 5.9. Here, an $n \times n$ array is partitioned into p partitions, each of which is a $n/r \times nr/p$ array, where r is divisible by p and is a divisor of both n and p. The array is divided into horizontal strips when $r = 1$, and vertical strips when $r = p$. For our study, we use a value of $\sqrt{p}$ for r, and hence the array is divided into square partitions, which is clearly possible only when p is a perfect square.

The objects created by OCDEP and OCDEP_NOVLP are the same for the first parallel code segment and the termination code segment, but are different for the other parallel code segments. The objects created by OCDEP_NOVLP for each of these parallel code segments are shown in Figure 5.10. An object-synchronization operation involves the overhead of looking up the object's information and that of sending as many messages

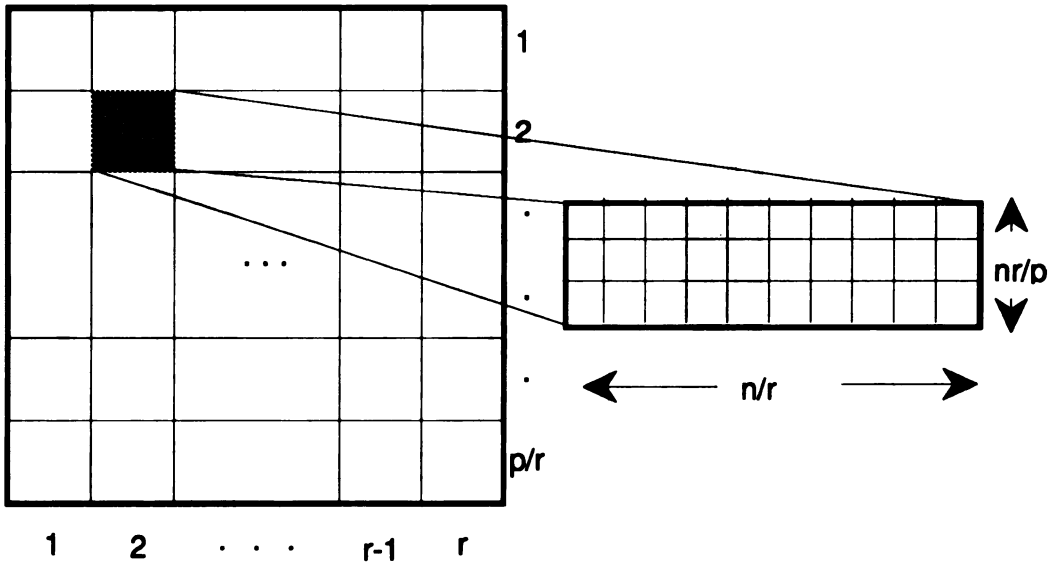

Figure 5.9. Rectangular partitioning of a $n \times n$ array.

as the number of processors in which the object needs to be updated. OCDEP limits the number of objects and messages sent by each processor to the minimum possible value, which is equal to the number of processors reading data written by this processor, while OCDEP_NOVLP does not guarantee this condition. Considering the single-element object O_1 created by OCDEP_NOVLP, two messages are required to send it to P_2 and P_4, for which the software overhead of the object update may be much greater than the overhead of transferring the single element. Further, two more messages are sent around the same time to each of these processors to update two other objects (O_2 and O_3 for P_2; O_4 and O_7 for P_4). Hence, we expect OCDEP to offer a better performance than OCDEP_NOVLP.

We model the execution of the application **parallel iterative solver** under the following assumptions. Each processor has a local memory and cannot address remote memory directly. In addition, there is a global memory that is directly addressable by all processors. For efficiency, synchronization variables reside in global memory, while all other variables are replicated in local memories on reference. Processors communicate with each other by means of messages through a mesh inter-

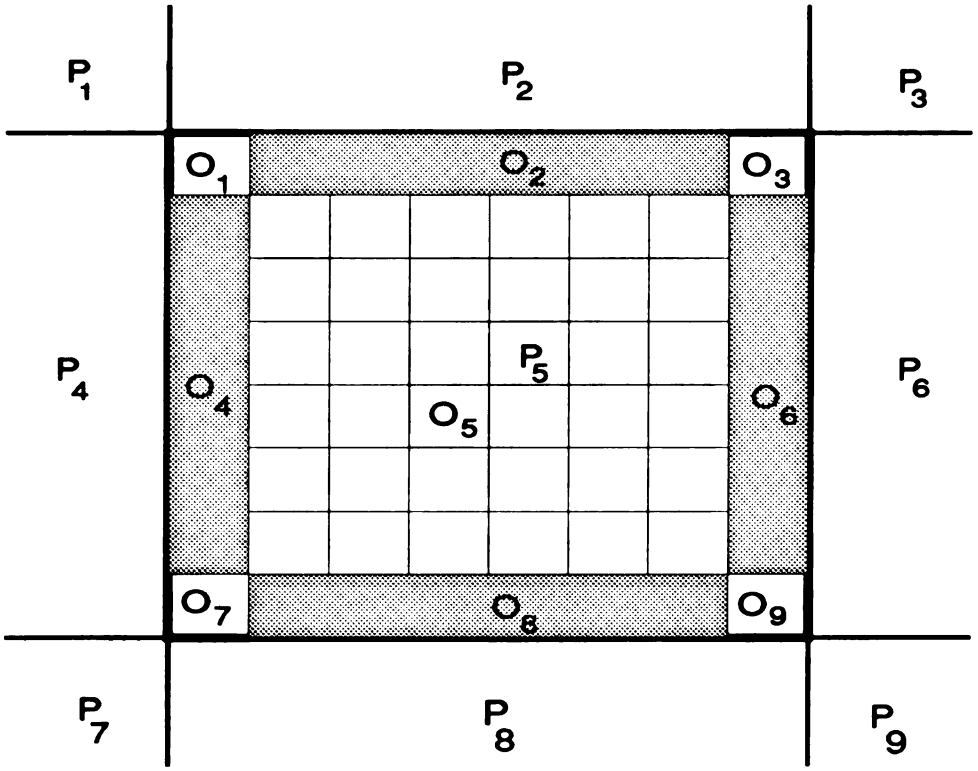

Figure 5.10. Objects created by **OCDEP_NOVLP** for application **parallel iterative solver**.

connection network. Message transmission is handled by a router that can send and receive messages simultaneously in all four directions. Messages may carry objects or acknowledgements and are assumed to be of sufficient size so that it is not necessary to split an object into multiple messages. The header of each message contains the message type, source, destination, and object-ID. While sending a message, the router sends the message size which is used and then discarded by the receiving router. Information about relevant objects is distributed to respective processors during the initialization phase and processors retrieve information about an object using the object-ID in the received message. Object update is done in a distributed manner, that is, each processor individually sends (receives) objects to (from) relevant processors. We do not model prefetching and pipelining and also ignore the time needed for address translation using the TLB and the page tables. Further, we assume that sufficient amount of main memory is available to ensure that application pages are

not paged out.

Typically, the parallel program consists of an initialization phase when the variables are read from disk into memory, a computation phase when the loop iterations are executed, and a termination phase when the results are written back to disk from memory. For the compute-intensive parallel iterative solver, the times for initialization and termination are small compared to the time for computation. Further, assuming that the time taken for each outermost loop iteration is the same, it is sufficient to consider the time taken for one such iteration with i processors, $t_{iter}(i)$, in comparing **OCDEP_NOVLP** and **OCDEP**. We refer to this time as the execution time and its value depends on the times spent in processing, memory references, object update, and synchronization. Though **OCDEP_NOVLP** and **OCDEP** lead to different times for updating objects, the resulting reduction in execution time, normalized by the execution time with **OCDEP_NOVLP**, depends on the fraction of $t_{iter}(i)$ spent in updating objects. This fraction depends on the time taken by the other components, out of which the times for processing and memory references depend on technology, and the synchronization time depends on the method of synchronization. To compare **OCDEP_NOVLP** and **OCDEP** for two widely different ways of synchronization, we study a method using barrier synchronization (referred to as **BARRIER**) and another using wait_signal synchronization (referred to as **WAIT_SIGNAL**). Table 5.2 lists the model parameters and their typical values in CPU cycles. These values are based on existing multiprocessors such as the BBN's TC2000 and processors such as the Intel's iWARP and the MIPS R4000. We assume a 50MHz clock driving the CPU, which amounts to a CPU cycle time of 20 ns. Entries in the table that have two values are for the **BARRIER** and the **WAIT_SIGNAL** cases, respectively.

BARRIER: The following code is executed by each processor for every outermost loop iteration:

```
Step 1.  Compute the value of assigned elements
Step 2.  Barrier
/* All processors have finished computation */
```

Table 5.2. Parameters of the application execution model.

Parameter	Notation	Value
Size of the message header	h	6
Context restore time	$t_{context\_restore}$	100
Context save time	$t_{context\_save}$	100
Time for a floating point addition	t_{fpa}	4
Time for a floating point multiplication	t_{fpm}	8
Global memory access time	t_{gm}	40
Local memory access time	t_{lm}	10
Time to look up object information	$t_{os\_obj\_lookup}$	100
Time spent in checks before sending an object message	$t_{os\_object\_msg\_send}$	0/100
Time spent in updating counters on receiving an object message	$t_{os\_object\_msg\_rec}$	0/100
Time spent in checks before sending an acknowledgement message	$t_{os\_ack\_msg\_send}$	0
Time spent in updating counters on receiving an acknowledgement message	$t_{os\_ack\_msg\_rec}$	0
Time to process a router receive interrupt	$t_{rec\_int\_proc}$	20
Time to initiate a router send	$t_{initiate\_send}$	20
Transmission time per 32-bit word	t_t	5

```
Step 3.  Send (receive) locally-written (locally-needed) objects
Step 4.  Barrier
/* All processors have finished object update */
```

where, barrier synchronization is implemented as follows:

```
/* barrier_counter is initialized to (number of processors) */
acquire(barrier_lock)
decrement(barrier_counter)
IF (barrier_counter is 0)
/* last processor to reach the barrier */
    barrier_counter = number of processors
    barrier_done = TRUE
    release(barrier_lock)
```

```
ELSEIF (barrier_counter is equal to (number of processors - 1))
/* first processor to reach the barrier */
    barrier_done = FALSE
    release(barrier_lock)
    poll UNTIL barrier_done
ELSE
/* all other processors */
    release(barrier_lock)
    poll UNTIL barrier_done
ENDIF
```

To reduce the contention on "barrier_counter", we use an additional variable "barrier_done". The longest time taken by a processor for a single iteration determines the overall execution time for all processors for that iteration. Processors may start the computation phase at various instants and arrive at the barrier at different instants. We assume that these delays do not increase the time spent in the barrier more than the time for each processor to execute the barrier code serially. Processors may also start the object update phase at different instants; in the worst case, actual object update begins only after all processors have left the barrier. Similarly, processors may leave the object update phase and arrive at the second barrier at different instants and the assumptions for the first barrier also apply to the second one.

Let the worst-case times for processing, memory reference, and barrier synchronization for i processors and an $n \times n$ array be $t_{proc}(n,i)$, $t_{mem}(n,i)$, and $t_{barrier}(i)$, respectively. If $t_{object}(n,i,\mathtt{OC})$ is the object update time when objects are created by the policy OC, the worst-case time $t_{iter}(n,p,\mathtt{OC})$ for executing one iteration with p processors is given by:

$$t_{iter}(n,p,\mathtt{OC}) = t_{proc}(n,p) + t_{mem}(n,p) + t_{barrier}(p) + t_{object}(n,p,\mathtt{OC}) \qquad (5.1)$$

Since individual processor delays in starting the processing phase is compensated by

the wait at the barrier, the processing time (in step 1) is given by:

$$t_{proc}(n,p) = \frac{n^2}{p} \times (3t_{fpa} + t_{fpm}) \tag{5.2}$$

Since data is replicated in local memory on reference, the time spent in memory references (in step 1) is given by:

$$t_{mem}(n,p) = \frac{n^2}{p} \times (5t_{lm}) \tag{5.3}$$

Based on the barrier implementation code and since all synchronization variables reside in global memory, the total barrier time (in steps 2 and 4) is given by:

$$t_{barrier}(p) = 2(7p + 1)t_{gm} \tag{5.4}$$

Since remote memory is not directly addressable, objects are updated using messages. The mesh interconnect matches the near-neighbor communication for the 5-point stencil case and therefore, no message forwarding is needed. The shared virtual space of the user process executing the partition is also available in the supervisor state, thereby allowing the processor to directly update each received object in the appropriate portion of the shared virtual space. In calculating the object update time, we assume that all processors start the object update phase at the same time. Therefore, while a processor is sending a message corresponding to a given object, another processor is sending a similar message to it. For example, considering the objects in Figure 5.10, when P_5 sends its O_1 to P_4, P_6 is sending its O_1 to P_5.

Each processor is involved in the following activities while sending an object (for example, consider P_5 sending O_1 to P_4): (1) save context of user process (object-synchronization is implemented as a system call), (2) look up object information, (3) write object message in router buffer, (4) initiate message send, (5) save context and wait for receive interrupt (the wait time is the greater of object message transmission time and context save time), (6) process interrupt for received object message (for example, P_5 now receives O_1 from P_6), (7) look up object information, (8) update

object in the user virtual space, (9) write acknowledgement message in router buffer, (10) initiate message send, (11) wait for receive interrupt, (12) process interrupt for received acknowledgement (for example, by now P_5 receives the acknowledgement for O_1 from P_4), (13) restore context to continue object-synchronization system call in supervisor state, and (14) restore context of user process after completion of object-synchronization system call. Here, steps (3) through (13) are repeated for every processor in which the object is updated.

Since the time for the barrier in step 2 includes the time until all processors leave the barrier, it includes the time spent in waiting for processors entering the object update phase at different instants. We also assume that the additional time in sending/receiving stride objects when compared to sequential objects is insignificant compared to the time for other factors in an object update. Hence, the time to send an object of size s in k processors including the time to process an acknowledgement is given by:

$$\begin{aligned} t_{sendobj}(s,k) &= t_{context\_save} + t_{os\_obj\_lookup} + k \times (t_{os\_obj\_msg\_send} + \\ &\quad t_{lm} \times (s+h) + t_{initiate\_send} + t_{wait\_to\_receive} + t_{rec\_int\_proc} + \\ &\quad t_{os\_ack\_msg\_rec} + t_{context\_restore}) + t_{context\_restore} \end{aligned} \tag{5.5}$$

The time to receive an object of size s including the time to send an acknowledgement is given by:

$$\begin{aligned} t_{recobj}(s) &= t_{rec\_int\_proc} + t_{os\_obj\_lookup} + t_{os\_obj\_msg\_rec} + st_{lm} + \\ &\quad t_{os\_ack\_msg\_send} + ht_{lm} + t_{initiate\_send} + ht_t \end{aligned} \tag{5.6}$$

In using **OCDEP_NOVLP**, eight objects are sent and 12 objects (three from each neighboring processor) are received, and the object update time (in step 3) is given by:

$$t_{object}(n,p,\textbf{OCDEP\_NOVLP}) = 4 \times t_{sendobj}(1,2) + 4 \times t_{sendobj}(\frac{n}{\sqrt{p}} - 2, 1) +$$

$$8 \times t_{recobj}(1) + 4 \times t_{recobj}(\frac{n}{\sqrt{p}} - 2) \tag{5.7}$$

In using **OCDEP**, four objects are sent and four objects are received, and the object update time (in step 3) is given by:

$$t_{object}(n, p, \textbf{OCDEP}) = 4 \times t_{sendobj}(\frac{n}{\sqrt{p}}, 1) + 4 \times t_{recobj}(\frac{n}{\sqrt{p}}) \tag{5.8}$$

Note that as the number of processors is increased, the times taken for processing, memory reference, and object update all decrease, while the barrier time increases. The execution times for **OCDEP_NOVLP** and **OCDEP** are obtained using Equations (5.1) through (5.8).

WAIT_SIGNAL: The following code is executed by each processor for every outermost loop iteration:

```
Step 1.  Compute the value of assigned elements
Step 2.  Call "ready-to-receive-object"
Step 3.  Send (receive) locally-written (locally-needed) objects
Step 4.  Call "object-reception-status"
```

The times spent in processing and memory references in step 1 are the same as in the **BARRIER** case. Before starting the object update phase (step 3), each processor needs to wait only until all the destination processors for its objects finish computation. This wait can be ensured by using wait_signal synchronization between processor pairs (steps 2 and 4). For every processor, we allocate a lock-protected "ready" flag (initialized to 0) in global memory for each processor from which it receives objects. For the case of square processors and a 5-point stencil, each processor receives objects from its four neighbors and hence four "ready" flags. Setting a "ready" flag to 1 indicates that the processor is ready to receive objects from the corresponding neighbor. Also, each processor maintains a counter "num-received-objects" to record the number of objects it receives. Once step 1 is completed, a system call "ready-to-receive-object" is executed (step 2), parameters to which include the number of

objects to be received from each neighbor and the total number of objects to be received. This call sets "num-received-objects" to 0 and atomically sets all the "ready" flags to 1.

A processor sends an object only if the "ready" flag of the receiving processor is 1. For example, P_5 in Figure 5.10 sends O_1 to P_2 only if the "ready" flag of P_2 corresponding to P_5 is 1. Further, each processor sets the "ready" flag corresponding to a given processor to 0 on receiving the last object, before sending an acknowledgement for that object. These operations guarantee that the elements are updated for a new iteration only after they have been used for the previous iteration. Also, each processor increments "num-received-objects" whenever it receives an object. After all objects have been sent (in step 3), computation for the next iteration begins only when all objects have been received. The reception of objects is checked by executing the system call "object-reception-status" which compares "num-received-objects" to the expected number of objects and returns a boolean value. Since "num-received-objects" is maintained in the supervisory state, it is not directly accessible in the user state and hence the system call. These operations guarantee that each user process begins the computation phase for the next iteration only after the objects it needs for this iteration have been updated. The time for executing one iteration for the **WAIT_SIGNAL** case is given by:

$$t_{iter}(n, p, \mathtt{OC}) = t_{proc}(n, p) + t_{mem}(n, p) + t_{wait\_signal} + t_{object}(n, p, \mathtt{OC}) \qquad (5.9)$$

The **BARRIER** case allows comparison of **OCDEP_NOVLP** and **OCDEP** policies for a poor method of synchronization. To allow comparison for an ideal synchronization method, we assume that the "ready" flags and their locks are allocated in global memory such that operations on them can proceed in parallel in all processors. The value of $t_{os\_obj\_msg\_send}$ is higher for the **WAIT_SIGNAL** case than in the **BARRIER** case to account for the checks performed on the "ready" flags while sending an object message. Similarly, $t_{os\_obj\_msg\_rec}$ is higher to account for the time spent in keeping track of the last object received from each processor and also incrementing "num-received-objects".

Therefore, the time for wait_signal synchronization is given by:

$$t_{wait\_signal} = 2t_{context\_save} + 2t_{context\_restore} + 2 \times (5 \times 4 \times t_{gm}) + 3t_{lm} \qquad (5.10)$$

Equations (5.9) and (5.10) can be used along with other equations derived for the **BARRIER** case to obtain the execution times for both **OCDEP_NOVLP** and **OCDEP**. Since all components of the execution time decrease as the number of processors is increased, the **WAIT_SIGNAL** case is more scalable than the **BARRIER** case.

5.7 Results of the Application Execution Model

In this section, we present the results for the **BARRIER** and the **WAIT_SIGNAL** cases obtained by using typical values for the model parameters as listed in Table 5.2. The results presented here are derived by varying n from 128 to 1024 in steps of 2 and in each case varying p from 4 to 1024 in steps of 4. Figure 5.11 shows how the reduction in object update time for **OCDEP** from that for **OCDEP_NOVLP** varies with the number of processors and the array size, in the **BARRIER** case. This reduction is measured as $(t_{object}(n, p, \texttt{OCDEP\_NOVLP}) - t_{object}(n, p, \texttt{OCDEP}))/t_{object}(n, p, \texttt{OCDEP\_NOVLP})$, using Equations (5.7) and (5.8). The difference in object update times is independent of the array size and the number of processors because both **OCDEP_NOVLP** and **OCDEP** transfer the same number of elements. Also, for a given array size, $t_{object}(n, p, \texttt{OCDEP\_NOVLP})$ decreases as the number of processors is increased and hence the reduction increases. For a given number of processors, $t_{object}(n, p, \texttt{OCDEP\_NOVLP})$ increases as the array size is increased, and hence the reduction decreases. The curves for the **WAIT_SIGNAL** case shown in Figure 5.12 are similar and all values are higher than the corresponding values in the **BARRIER** case because of the higher values of parameters $t_{os\_object\_msg\_rec}$ and $t_{os\_object\_msg\_send}$.

The fraction of the execution time spent in updating objects for policy **OC** is measured as $(t_{object}(n, p, \texttt{OC})/t_{iter}(n, p, \texttt{OC}) \times 100)$ using the equations derived in the previous section. Figures 5.13 and 5.14 show how this fraction varies with the number

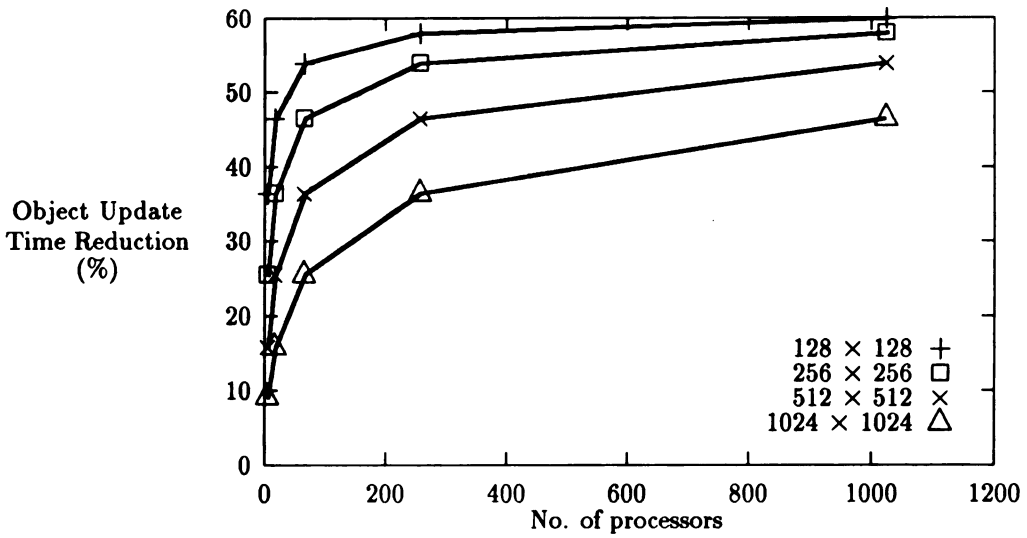

Figure 5.11. Object update time reduction due to **OCDEP/BARRIER**.

of processors and the array size, for the **BARRIER** and **WAIT_SIGNAL** cases, respectively. As the number of processors is increased, the times for processing and memory reference that are affected by p (Equations (5.2) and (5.3)) decrease at a faster rate than the object update time which is affected only by $\sqrt{p}$ (Equations (5.5) through (5.8)). This fact explains why in the case of both **OCDEP_NOVLP** and **OCDEP**, with a given array size, the object update fraction of the execution time increases as the number of processors is increased, for the entire range of processors in the **WAIT_SIGNAL** case. The same behavior occurs for small number of processors in the **BARRIER** case, but for large number of processors there is a sharp decrease in object update fraction due to the large amount of time spent in the barrier. The drop occurs at a higher number of processors for larger array sizes because of the higher non-barrier components of the execution time in those cases. In all cases, for a given number of processors, the fraction is higher for smaller grid sizes, because of the corresponding lower times for processing and memory references. Further, the fraction for **OCDEP** is always lower than that for **OCDEP_NOVLP** because of the reduced object update time for the latter. It is seen that for the inefficient **BARRIER** case, the highest value of

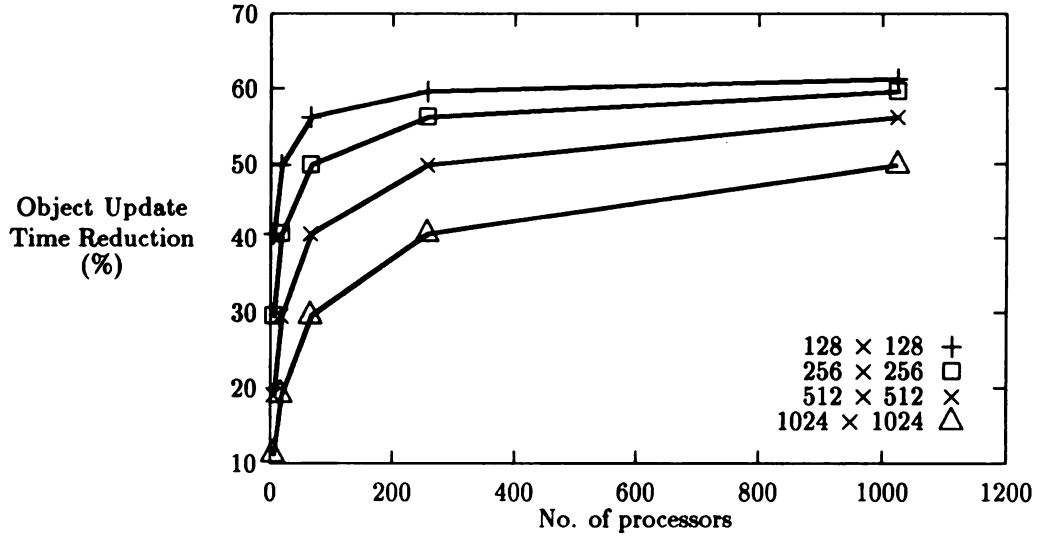

Figure 5.12. Object update time reduction due to OCDEP/WAIT_SIGNAL.

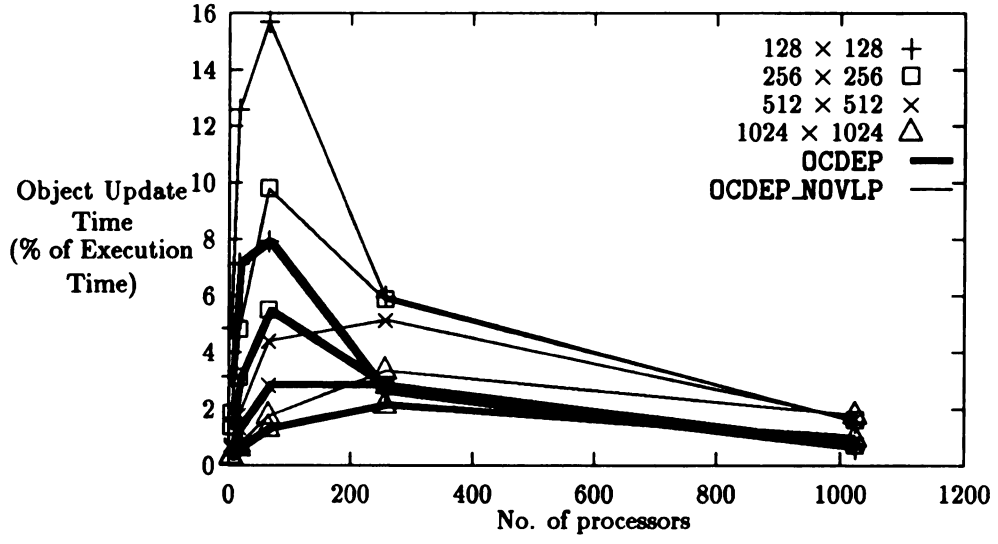

Figure 5.13. Object update time fraction/BARRIER.

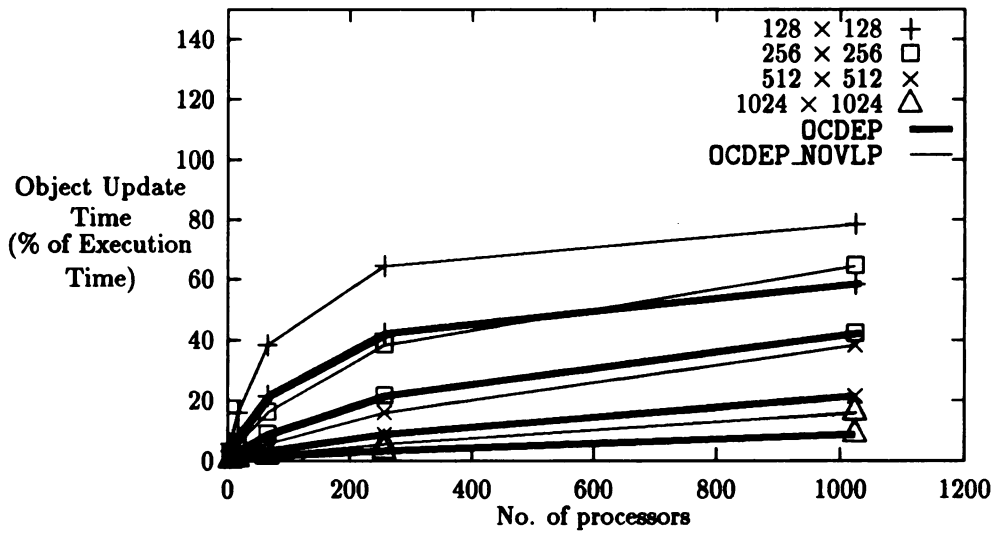

Figure 5.14. Object update time fraction/WAIT_SIGNAL.

the fraction is only 16% compared to 80% for the WAIT_SIGNAL case. Since only the object update time component of the execution time is different in the case of the two object creation policies, the normalized reduction in execution time measured as $((t_{iter}(n,p,\texttt{OCDEP\_NOVLP}) - t_{iter}(n,p,\texttt{OCDEP}))/t_{iter}(n,p,\texttt{OCDEP\_NOVLP}) \times 100)$ is similar to the object update fraction, as shown in Figures 5.15 and 5.16. It is seen that execution time reductions as high as 48% are achieved for the WAIT_SIGNAL case.

Conclusions: OCDEP always performs better than OCDEP_NOVLP, with object update time reductions as high as 46-60% and 50-61% in the BARRIER and WAIT_SIGNAL cases, respectively. The performance improvement due to OCDEP as measured by the normalized reduction in execution time is dependent on the other components of the execution time. The time spent in synchronization depends on the efficiency of the method used and for the efficient WAIT_SIGNAL scheme, the execution time reduction is as high as 48% for the smallest array size studied, while it is significant (8%) even for the largest array size studied. Similarly, a faster processor will reduce the fraction of the execution time spent in processing and lead to a higher object update time fraction and consequently lower execution time.

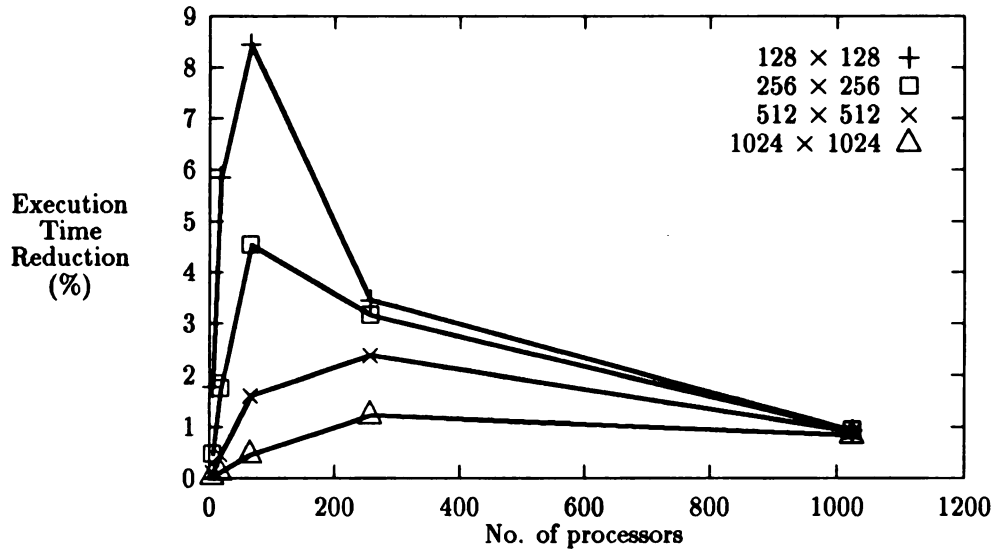

Figure 5.15. Execution time reduction due to **OCDEP/BARRIER**.

The component of the execution time that is different for **OCDEP** and **OCDEP_NOVLP** is that spent in transferring data and keeping it consistent. Both schemes achieve these functions by creating and transferring data objects. The time spent in transferring objects depends on the number of objects, the number of processors that share each object, and the amount of data transferred. When the transfer of each object between a given pair of send and receive processors requires one message, the best performance is achieved for **OCDEP** which creates the least number of objects. **OCDEP** achieves this better performance by exploiting the nature of the application's data exchange patterns when creating objects. It minimizes the number of objects by allowing them to overlap and combining objects which are created by **OCDEP_NOVLP** that are updated around the same time. Our results demonstrate that performance can be improved by allowing objects to overlap, and allocating to a single object elements of an array variable that need to be updated around the same time. It follows that when possible, objects of different array variables that need to be updated around the same time should also be combined and sent in a single message.

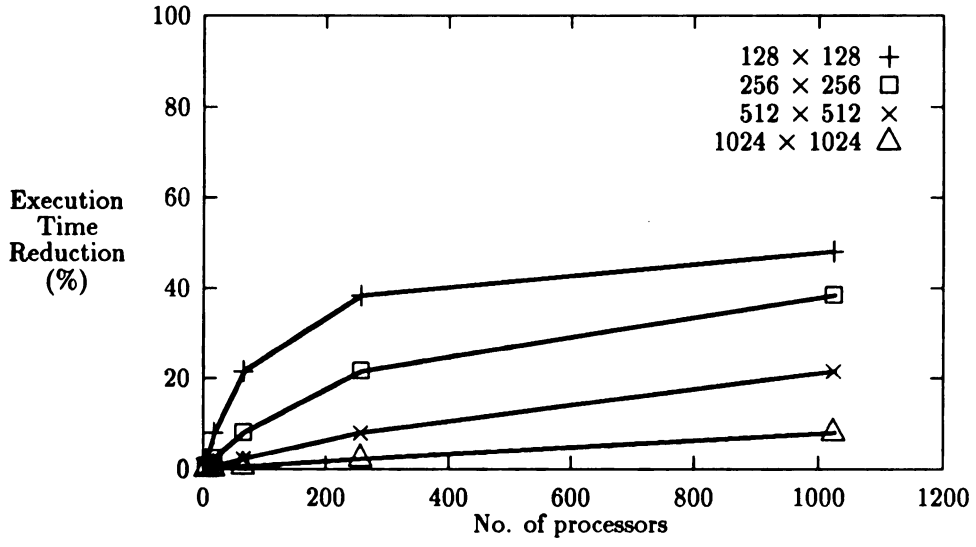

Figure 5.16. Execution time reduction due to OCDEP/WAIT_SIGNAL.

5.8 Summary

In this chapter, we presented our new compile-time object-creation scheme which achieves compiler-directed object-level placement. We outlined the algorithm used by our scheme and derived the compilation overhead in applying it. We discussed how compiler-directed object-level placement can provide a better performance than block-level placement. We also identified possible performance optimizations in providing object-level placement by modeling the execution of a typical application. These optimizations motivate the performance measures we use in the next chapter to quantitatively compare various types of placement schemes.

CHAPTER 6

EXPERIMENTAL COMPARISON OF DATA PLACEMENT SCHEMES

In this chapter, we present the results of our experimental study in which we compare various data placement schemes that can be used when providing a shared virtual memory (SVM) in a NUMA multiprocessor. As discussed in the previous chapters, these schemes differ in the degree to which the compiler controls data placement operations. We first state the goals, assumptions, and performance measures of our study. We then outline the manner in which we determine the performance measures for the various data placement schemes. After discussing in detail the results of our experiments for various applications, we end by summarizing our conclusions.

6.1 Goals and Assumptions

In the previous chapters, we presented our work on compiler-assisted block-level and compiler-directed object-level placement. We claimed that the compiler's knowledge about the data reference pattern can be better used if it is allowed more control over data placement operations. Our goal in this chapter is to demonstrate our claim by quantitatively comparing data placement schemes which differ in the degree to which the compiler controls data placement operations. We consider block-level and object-

level placement, and vary the amount of compiler control in data placement operations by compiling the application with different object-creation schemes. More specifically, we compare the performance of a given application for each of the following cases: (1) block-level placement with no compiler assistance (compiled with `OCS`), (2) compiler-assisted block-level placement (compiled with `OCRP`), and (3) compiler-directed object-level placement (compiled with `OCDEP`). We refer to these three cases as `OCS`, `OCRP`, and `OCDEP`, respectively. Since our study [55, 56] which modeled object-level placement demonstrates that there is performance improvement in allowing objects to overlap, we choose `OCDEP` instead of `OCDEP_NOVLP` for the third case.

We make the following assumptions. A single copy of the application exists in the hard disk, and a *server* processor is responsible for the transfer of data between the disk and the main memory. The server executes all the sequential code segments of the application and is distinct from the set of processors which execute the parallel code segments. We restrict ourselves to data placement at a given level of the memory hierarchy (cache or main memory), and assume an infinite amount of physical memory in that level. The latter assumption is typical of studies on memory performance and removes the non-determinism introduced by replacement schemes. The interconnect allows reliable FIFO communication between any two processors, and we do not assume any specific broadcast or multicast capabilities. Each processor has an instruction cache which is large enough to hold its code, and we ignore instruction references.

6.2 Performance Measures

In general, the execution time of a parallel application is comprised of times spent on initialization, termination, processing, synchronization, and memory references. The times spent on initialization and termination is dominated by the time taken for the transfer of data between the disk and the main memory of the server, and is the same for all the three cases of our study. Further, the times spent on processing and synchronization are the same for the three cases. Hence, we consider only

the time spent in memory references and this time depends on the data placement scheme. In the case of OCS and OCRP, we assume block-level data replication within each code segment, with sequential consistency maintained using the write-update (WU) scheme. The remap operations between successive code segments in the case of OCRP deallocate and allocate physical blocks and hence, migrate data under certain conditions. In the case of OCDEP, object-synchronization operations are used to replicate data and keep copies consistent as and when necessary.

Since data is replicated in all the three cases, any memory reference is serviced locally. The three cases however differ in the amount of time spent for (1) transferring data to local memory on a data miss, and (2) keeping the various copies of data consistent. We compare the software overhead involved in transferring data by measuring in each case the number of data transfer and consistency messages, and the total number of messages. We also compare the overhead for actual transmission of data by measuring in each case the total amount of data transferred during both consistency and data transfer messages. Also, both OCRP and OCDEP package data into objects, and we compare the corresponding overhead by recording the total number of objects transferred by each of these schemes. Finally, in order to study the severity of fragmentation when applying OCRP, we measure the amount of physical and virtual memory required for the three cases as a percentage of the problem size.

Other studies on providing a SVM in NUMA multiprocessors such as [51] have also measured performance by the number of messages and the amount of data transferred. Our performance measures do not account for the runtime overhead of adaptive placement, which is higher for OCS compared to the other schemes. It also does not include the possible performance loss due to inefficient code generation when applying OCRP. This loss is however insignificant when compared to the additional runtime overhead for OCS. Therefore, in comparing all the three cases, our results are skewed in favor of OCS. In comparing OCRP and OCDEP, our results are skewed in favor of OCRP.

6.3 Methodology

In the case of **OCS**, we calculate the performance measures separately for each code segment as follows. We determine each data element's reference pattern using **Algorithm A** outlined in Chapter 3. For the purposes of our experimental study, we restrict the implementation of **Algorithm A** to handle applications in which: (1) the parallel code segment consists of a perfectly nested loop with the lower bound, upper bound, and step being integer constants, (2) the depth of the loop is at most three, (3) the dimension of the array variables is at most two, (2) there are no conditional statements, and (3) array references have linear subscripts, i.e., of the form $(a \times i + b)$, where a and b are integers and i is the loop index.

We calculate each block's reference pattern based on the reference pattern of its data elements. For example, consider a block which is referenced by p processors and written w times, and assume a block size of b data elements. For each of the p processors, the block is transferred if it does not already exist in the local memory of the processor. Assuming that the location of the block can be looked up locally and does not require any messages, each such transfer needs two messages, one to request for the block and the other to receive it. Hence, the number of data transfer messages for the block is $(2 \times p)$.

Each write needs to update the block's copy in the server as well as the other referencing processors. Assuming again that the information about the referencing processors can be looked up locally and does not require any messages, two messages are needed for updating each copy, one to send the update and the other to receive an acknowledgement. Hence, the number of consistency messages is $(2 \times p \times w)$. We ignore the data sent during block requests and WU acknowledgements, and therefore, the total amount of data transferred is $(p \times b + p \times w)$. Such a calculation gives us the steady-state performance because we assume that all processors start execution of the code segment at the same time. In practice, processors incur different delays and therefore it is possible for one of them to write a block before it is fetched by some of the others, in which case the block's copy in those processors need not be

updated.

In the case of OCRP and OCDEP, we distinguish between consistency and data transfer messages as follows. We consider messages sent from the server after the initialization code segment and those sent by the other processors before the termination code segment as data transfer messages. The rest of the messages are used for consistency operations within and between code segments, and we consider them as consistency messages. OCRP transfers data in terms of objects, and maintains block-level consistency within parallel code segments. In addition, it maintains consistency between code segments by the remap operations which transfer data in terms of objects. The number of objects and the amount of data transferred for the data transfer and the remap operations of OCRP is determined using the algorithm outlined in Chapter 4. The number of messages and the amount of data transferred for the block-level consistency operations of OCRP are determined based on each block's reference pattern as in the case of OCS. OCDEP on the other hand uses object-synchronization operations both for data transfer and consistency operations. We determine the number of objects and the amount of data transferred for OCDEP using the algorithm outlined in Chapter 5.

We define a *data transfer unit* as containing data to be transferred between a given pair of send and receive processors. In our previous study [55, 56], we considered each object as a single data transfer unit. We demonstrated that performance can be improved by minimizing the number of objects and hence the number of data transfer units. We did so by allowing objects belonging to the same array variable to overlap and combining several of these objects into a single object. On similar lines, performance can be improved by combining into a single data transfer unit objects of different array variables that need to be transferred around the same time between the same pair of send and receive processors.

We apply the above optimization for both OCRP and OCDEP, and assume that two messages are needed for each data transfer unit, one for the actual transfer and the other for the acknowledgement from the receiving processor after it updates the objects in its local memory. In this manner, we determine the number of consistency

and data transfer messages for OCRP and OCDEP which are required to transfer objects during consistency and data transfer operations, respectively. As in the case of OCS, we do not include in the amount of data transferred data which is sent during acknowledgements for the data transfer units.

Both OCS and OCDEP map the application's data contiguously to the SVM. They require the same amount of virtual memory, and further, they both guarantee that all virtual pages except possibly the last one is fully utilized. We determine their virtual memory needs by using the information about the number of data elements of the application and the virtual page size. OCRP on the other hand uses more virtual memory than the other two schemes because it ensures that objects for a given code segment do not share virtual pages. Assuming that virtual space for objects can be reused in each parallel code segment, the additional virtual memory needed by OCRP is the maximum out of that required for each of the parallel code segments. OCRP guarantees that all virtual pages of a given object except possibly the last one are fully utilized. Hence, fragmentation of virtual memory can occur when there are few elements in the last virtual page of an object relative to the page size. We determine the virtual memory needs of OCRP by using information about its objects and the virtual page size.

Both OCS and OCDEP allocate in local memory any non-local block which contains at least one referenced data element. They both use the same amount of physical memory, but portions of physical blocks might be unreferenced. We determine their physical memory needs by using each block's reference pattern and the physical block size. OCRP on the other hand requires physical memory for the objects it creates for each of the code segments. The remap operations of OCRP deallocate blocks at the end of each parallel code segment and therefore physical memory can be reused. The amount of physical memory for OCRP is that required by the server and the maximum out of that needed for each of the parallel code segments. OCRP guarantees complete utilization of all physical blocks except possibly the last one used by each object. Fragmentation of physical memory occurs only if there are very few elements mapped to the last block relative to the block size. We determine the physical memory needs

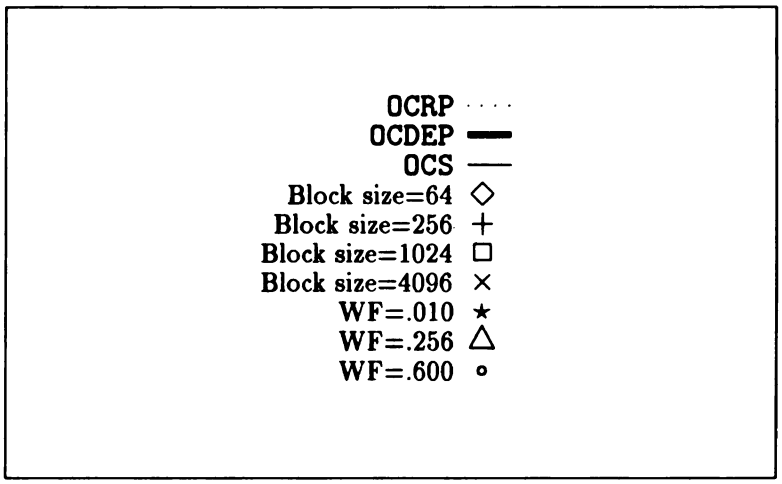

Figure 6.1. Legend for the performance graphs.

of OCRP using information about its objects and the physical block size. Since OCDEP requires the same amount of virtual and physical memory as OCS, we omit it in our performance graphs for virtual and physical memory, and plot only the results for OCS and OCRP.

6.4 Experimental Results

We choose our workload to highlight when the different amounts of compiler control in data placement leads to a difference in performance and when it does not. Our workload consists of applications **matrix multiplication** and **parallel iterative solver**, and also the application **multi-code-segment-array** for the three different values (.01, .256, and .6) of the parameter WF. In all cases, we assume that each element occupies four bytes of storage, and that both OCS and OCDEP maps array variables to the SVM in column major form. We conduct experiments for block sizes of 64, 256, 1024, and 4096 bytes, which are representative of block sizes at both the cache and the main memory levels of the memory hierarchy. In the case of each application, for each of the three schemes, we determine the number of consistency

```
     DOSEQ S1
         Initialize A,B,C
S1   END DOSEQ
     DOALL S2 I = 1 TO n
         DOALL S3 J = 1 TO n
             DOSEQ S4 K = 1 TO n
                 C(I,J) = C(I,J) + A(I,K) * B(K,J)
S4           END DOSEQ
S3       END DOALL
S2   END DOALL
     DOSEQ S5
         Output C
S5   END DOSEQ
```

Figure 6.2. Application **matrix multiplication**.

and data transfer messages, the total number of messages, the total number of objects, the amount of data transferred, and the physical and virtual memory requirements. Note that the number of objects are applicable only for **OCRP** and **OCDEP**, and the virtual memory requirements is applicable only when the block is a virtual page. The symbols used in all our performance graphs are shown in Figure 6.1, where the block size is specified in bytes.

6.4.1 Application matrix multiplication

Our first application **matrix multiplication** is shown in Figure 6.2. It consists of array variables which occur in a single parallel code segment and hence, no remap operations are needed between parallel code segments. We consider multiplication of 128×128 square matrices and partition the application by dividing the matrix C into as many square partitions as the number of processors (p). The iterations of loops

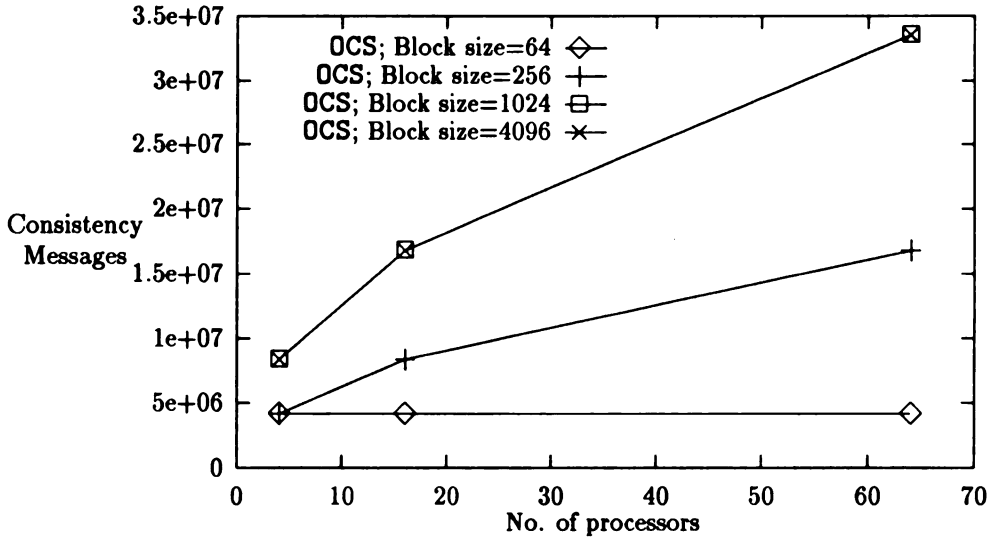

Figure 6.3. Consistency messages vs. Number of processors (**matrix multiplication** for 128 × 128 square matrices).

S_2 and S_3 are each divided into $\sqrt{p}$ sets and each processor executes one of the p resulting combinations. With such a partitioning, elements of A and B are shared read-only, those of C are each read and written by a single processor, and no data element exhibits read-write sharing. `OCRP` and `OCDEP` both apply optimizations to reduce the number of objects and therefore create the same number of objects. We discuss below the other performance measures.

Consistency Messages: `OCRP` eliminates false sharing, and `OCDEP` updates copies of elements only when necessary. Since no data element is shared in a read-write fashion, the number of consistency messages is zero for both `OCRP` and `OCDEP`. `OCS` on the other hand suffers from false sharing and has a non-zero number of consistency messages. Figure 6.3 shows how the number of consistency messages varies with the number of processors and the block size for `OCS`. For a given number of processors, this number increases as the block size increases because of the larger amount of false sharing.

The consistency overhead due to false sharing is absent for blocks belonging to A and B because they are not written. It exists for the blocks of matrix C and is

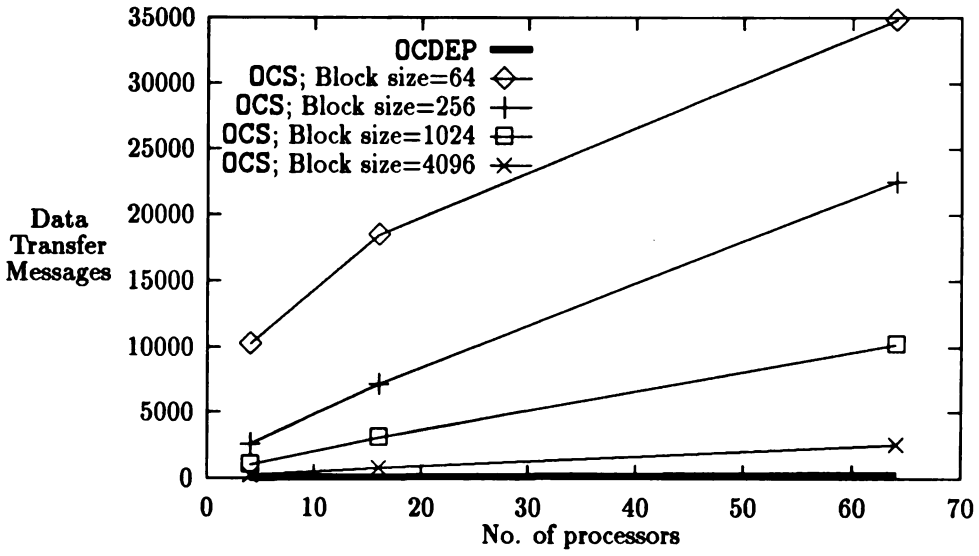

Figure 6.4. Data transfer messages vs. Number of processors (**matrix multiplication** for 128 × 128 square matrices).

proportional to the number of processors sharing each such block. Since the array elements are mapped to the SVM in column major form and C is partitioned into square partitions, at most $\sqrt{p}$ processors can share each block of C, as long as it does not cross one of the $\sqrt{p}$ column partitions. For our experimental parameters, this condition is satisfied for block sizes of 1024 and 4096, and hence they have an equal number of consistency messages. For a given block size, the number of consistency messages is higher for more processors because each block of C is shared by more processors.

Data Transfer Messages: Both OCRP and OCDEP fetch data in terms of objects and apply optimizations that reduce the number of data transfer units needed to transfer these objects. For example, objects for array variables A, B, and C that need to be sent from the server to a given processor are all combined into a single data transfer unit. For both schemes, the number of data transfer messages is the same and further, it is independent of block size. Due to the lack of compiler assistance, OCS transfers data in terms of blocks. It replicates in local memory any non-local block containing a referenced data element. Therefore, the number of data transfer

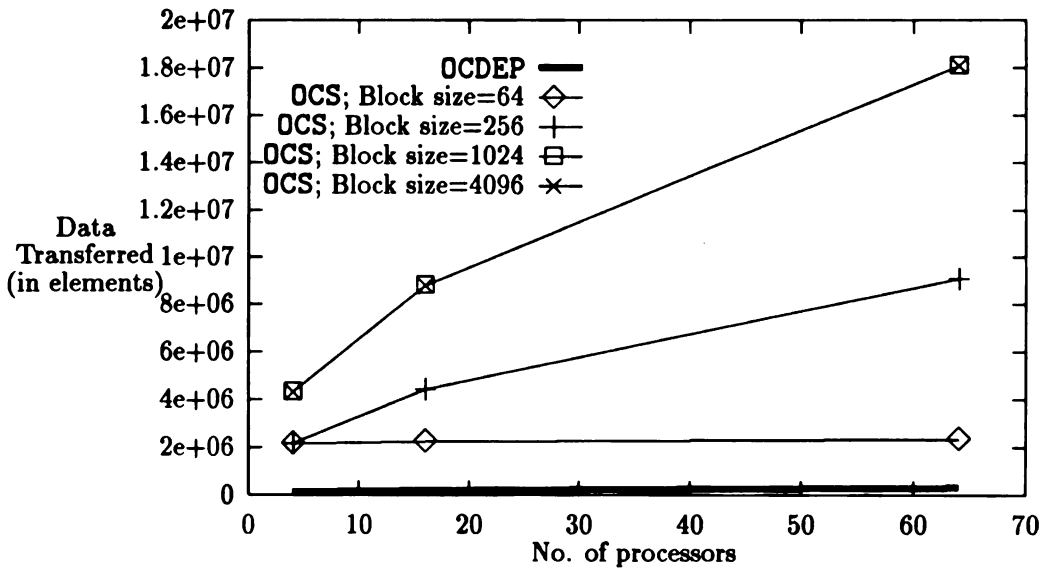

Figure 6.5. Data transferred vs. Number of processors (**matrix multiplication** for 128 × 128 square matrices).

messages for OCS depends on the block size. Figure 6.4 shows how the number of data transfer messages for OCDEP and OCS varies with the number of processors and the block size. It is seen that the number for OCS is much higher than that for OCDEP in the case of all block sizes studied. For a given number of processors, OCS needs more data transfer messages for smaller block sizes due to the reduced benefits of prefetching. For all schemes, the number of data transfer messages increases as the number of processors increases because data needs to be sent to more processors.

Amount of Data Transferred: OCDEP transfers only those data elements that will be read and actually need to be transferred. In general, OCRP transfers more data than OCDEP due to remap operations. Since these operations are absent for this application, both OCDEP and OCRP transfer the same amount of data, and further, this amount is independent of the block size. OCS transfers data in terms of blocks and each falsely-shared block contains data that is unreferenced and is therefore transferred unnecessarily. In addition, since consistency messages out number data transfer messages (Figures 6.3 and 6.4), they account for a significant fraction of the amount of data transferred.

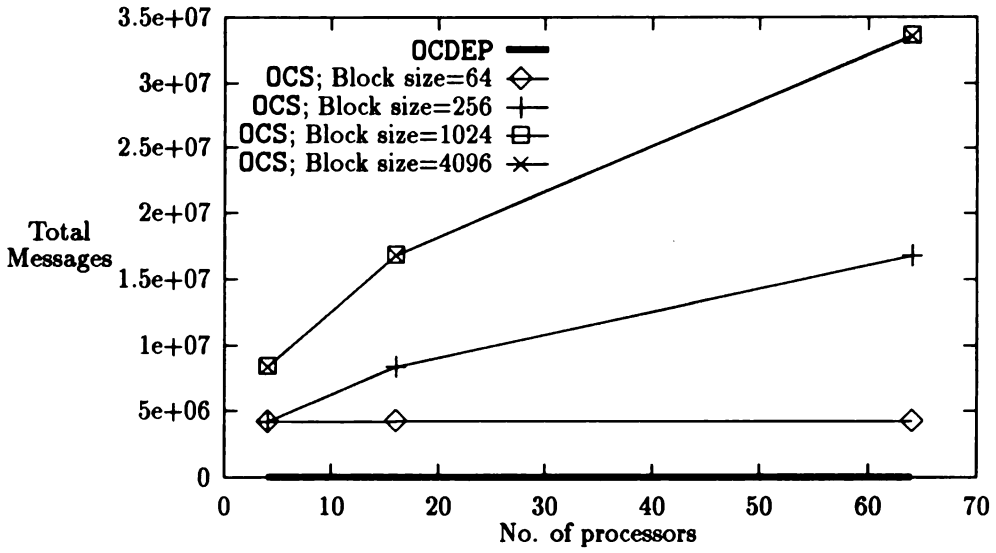

Figure 6.6. Total messages vs. Number of processors (**matrix multiplication** for 128 × 128 square matrices).

Figure 6.5 shows how the amount of data transferred varies with the number of processors for OCDEP and OCS, and with the block size for OCS. OCDEP transfers fewer data elements than OCS for all values for the block size and the number of processors. OCS transfers almost the same amount of data for block sizes of 1024 and 4096 because they have an equal number of consistency messages. For a given number of processors, it transfers more data for larger blocks because of the larger amount of false sharing. Both OCDEP and OCS transfer more data for a higher number of processors because some of the data is now shared by more processors.

Total Messages: OCDEP and OCRP have the same number of consistency and data transfer messages and therefore the same total number of messages. Since consistency messages out number data transfer messages for OCS, the plots for the total number of messages shown in Figure 6.6 are similar to those for the consistency messages (Figure 6.3). Since the number of consistency messages for OCDEP is zero, the plot for its total number of messages is similar to that for its data transfer messages (Figure 6.4). It is seen that the total number of messages for OCDEP is much smaller than that for OCS for all values of the block size and the number of processors.

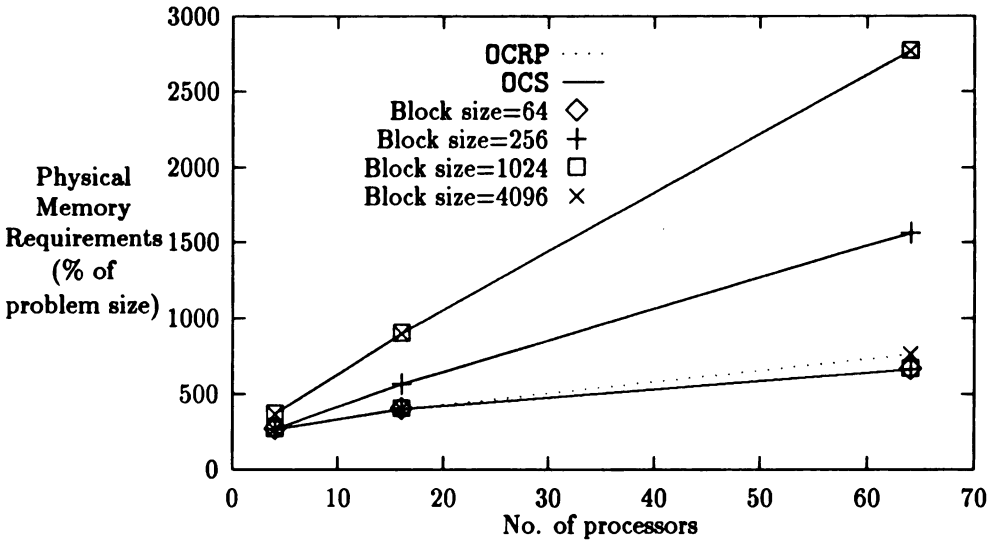

Figure 6.7. Physical memory requirements (% of problem size) vs. Number of processors (**matrix multiplication** for 128 × 128 square matrices).

Physical Memory Requirements: Figure 6.7 shows the variation of physical memory requirements for OCRP and OCS with the block size and the number of processors. For small block sizes, the physical memory needs for OCRP and OCS are the same. For large block sizes however, OCRP needs less physical memory than OCS because its fragmentation is not severe and because OCS leaves larger unreferenced portion in the physical blocks. For a given number of processors, both schemes require more physical memory for larger blocks because of the higher fragmentation for OCRP and the higher degree of false sharing for OCS. For a given block size, both need more physical memory for more processors because of the increased number of objects for OCRP and the increased degree of sharing of blocks for OCS.

Virtual Memory Requirements: Figure 6.8 shows the variation of virtual memory requirements for OCRP and OCS with the block size and the number of processors. For the parameters used in our experiments, for all values of the block size and the number of processors, the virtual memory needed by OCS is almost the same as the problem size. OCRP however requires additional virtual memory and this amount is higher for larger blocks and more processors because of the increased potential for

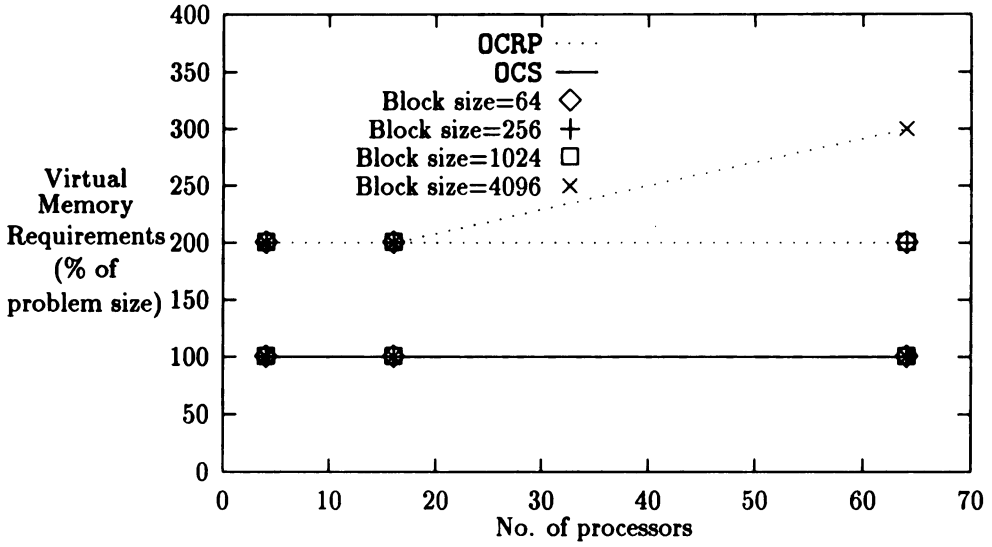

Figure 6.8. Virtual memory requirements (% of problem size) vs. Number of processors (**matrix multiplication** for 128 × 128 square matrices).

fragmentation.

Summary: OCRP and OCDEP perform several orders of magnitude better than OCS in terms of the total number of messages and the amount of data transferred. For these performance measures as well as the number of objects, OCRP and OCDEP perform the same because of the absence of remap operations and that of block-level consistency operations within parallel code segments. OCRP needs more virtual memory and less physical memory than OCDEP and OCS.

6.4.2 Application parallel iterative solver

Our second application **parallel iterative solver** is shown in Figure 6.9. It consists of an array variable which occurs in more than one parallel code segment, but with the same reference pattern in each of them. Again, no remap operations are necessary between parallel code segments. We consider parallel iterative solution with a 5-point stencil on a 130 × 130 array. The elements of the interior 128 × 128 array need to be computed, and we partition the computation by dividing this array into

```
   DOSEQ S1
      Initialize A
S1 END DOSEQ
   DOSEQ S2 K = 1 TO m
      DOALL S3 I = 1 TO n
         DOALL S4 J = 1 TO n
           A(I,J) = 0.25 * (A(I-1,J) + A(I+1,J) + A(I,J-1) + A(I,J+1))
S4       END DOALL
S3    END DOALL
S2 END DOSEQ
   DOSEQ S5
      Output A
S5 END DOSEQ
```

Figure 6.9. Application **parallel iterative solver**.

as many square partitions as the number of processors (p). Unlike the previous application **matrix multiplication**, data elements are shared in a read-write manner in the parallel code segments of this application.

As mentioned earlier, we assume block-level replication with sequential consistency in the case of OCS and OCRP. When data is placed in this manner, in practice, two arrays e.g., C and D are used instead of the single array A. Odd iterations of loop S_2 read data from C and write to D, while even iterations of loop S_2 read data from D and write to C. Such a scheme ensures that the values of the elements for the previous iteration of loop S_2 are not overwritten before being used for the current iteration. Since the data reference pattern for C and D are the same as that of A, for our experiments, we consider a single array and further, we measure the performance for one iteration of loop S_2. Also, we assume that processors compute either for a given number of iterations of loop S_2 or until convergence is attained. In the latter

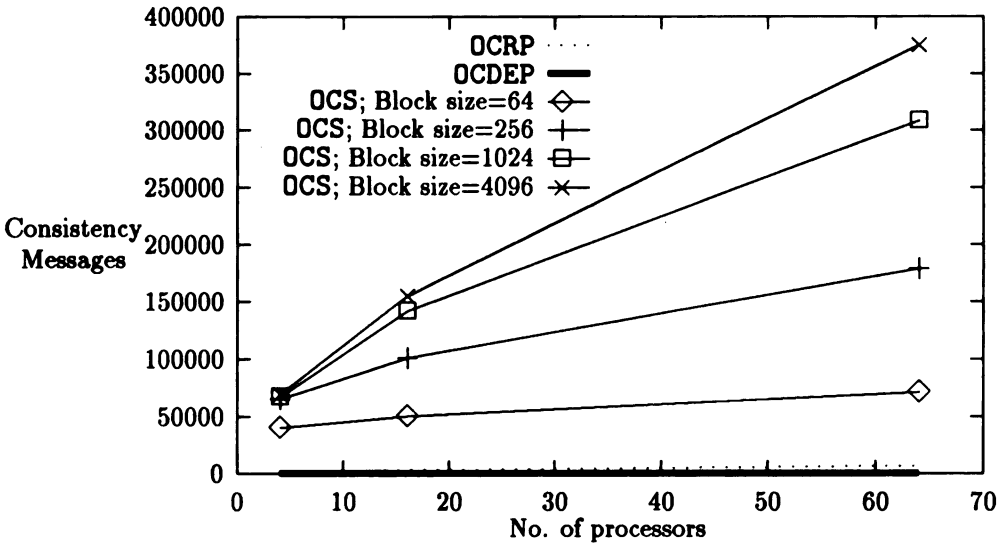

Figure 6.10. Consistency messages vs. Number of processors (**parallel iterative solver**).

case, each processor locally checks the convergence of the elements belonging to its partition. Since the computation involved in checking for convergence is the same for all the three object-creation schemes, we do not include it in our performance measure.

Consistency Messages: The parallel code segments corresponding to the iterations of loop S_2 each contain a DOALL loop. Therefore, OCDEP does not maintain consistency within these code segments, but uses object-synchronization operations to maintain consistency between them. OCRP and OCS on the other hand maintain block-level consistency within these code segments for each write operation. OCS however incurs more consistency operations than OCRP because its blocks are falsely-shared. As shown in Figure 6.10, for all the three schemes, the variation of the number of consistency messages with the number of processors is similar to that for application **matrix multiplication**. Also, the number of consistency messages is independent of the block size for OCDEP and OCRP, while in the case of OCS, for a given number of processors, it is higher for larger blocks. As expected, the number of consistency messages is the least for OCDEP and is the maximum for OCS. Figure 6.11 shows that the

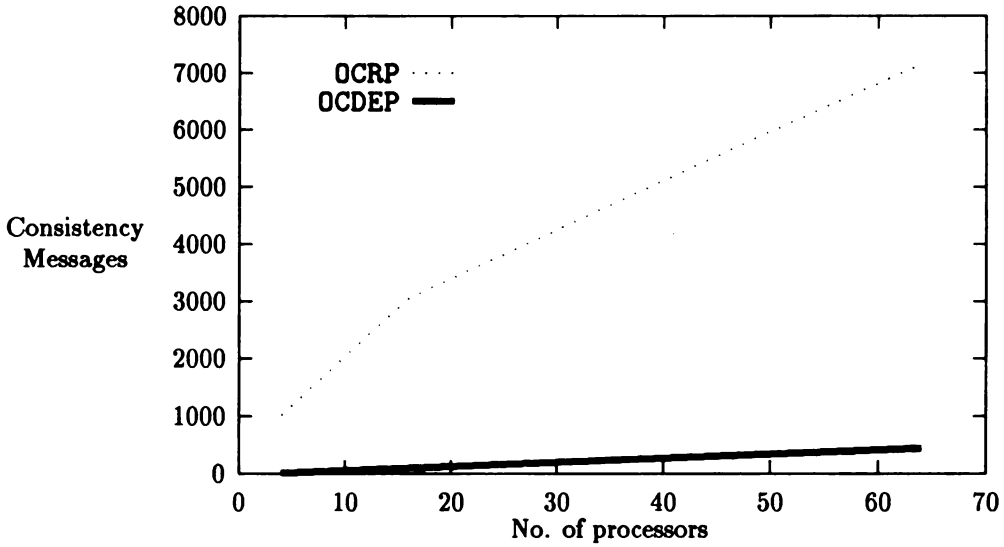

Figure 6.11. Consistency messages vs. Number of processors (**parallel iterative solver**; OCRP, OCDEP).

number of consistency messages for OCDEP is significantly lower than that for OCRP. Note that this number is for just one iteration of the loop S_2.

Data Transfer Messages: As in the case of application **matrix multiplication**, both OCDEP and OCRP apply optimizations in transferring objects and incur the same number of data transfer messages. Moreover, this number is independent of the block size for both schemes. The number of data transfer messages for OCS is higher than that for OCDEP and OCRP, and for a given number of processors, it is higher for smaller blocks. As shown in Figure 6.12, the number of data transfer messages for all the three schemes increases for a larger number of processors, as in the case of application **matrix multiplication**.

Amount of Data Transferred: During consistency operations, both OCDEP and OCRP transfer data elements which are written in a given iteration to processors that will read them in the next iteration. Further, due to the absence of remap operations OCRP does not transfer data between parallel code segments. Also, as in the case of application **matrix multiplication**, both schemes transfer the same amount of data during data transfer messages as well. The amount of data transferred is hence

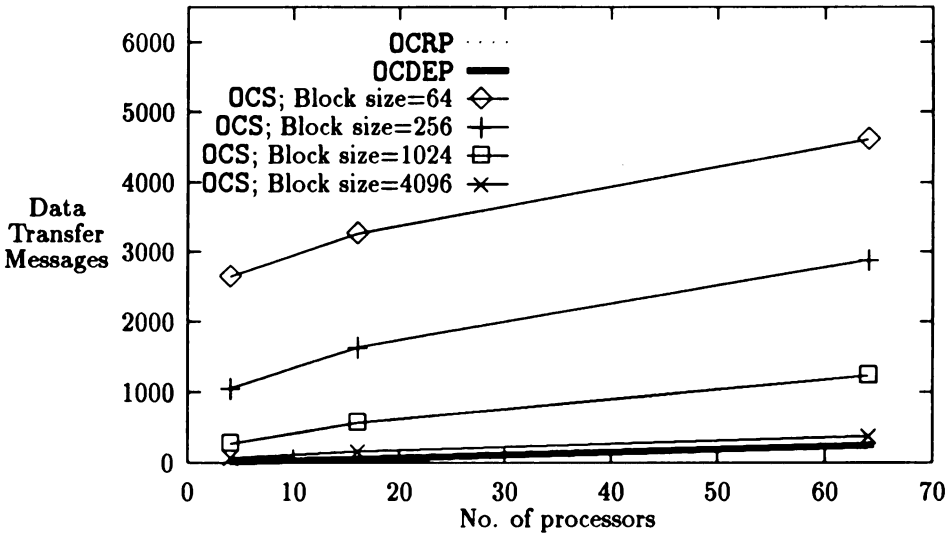

Figure 6.12. Data transfer messages vs. Number of processors (**parallel iterative solver**).

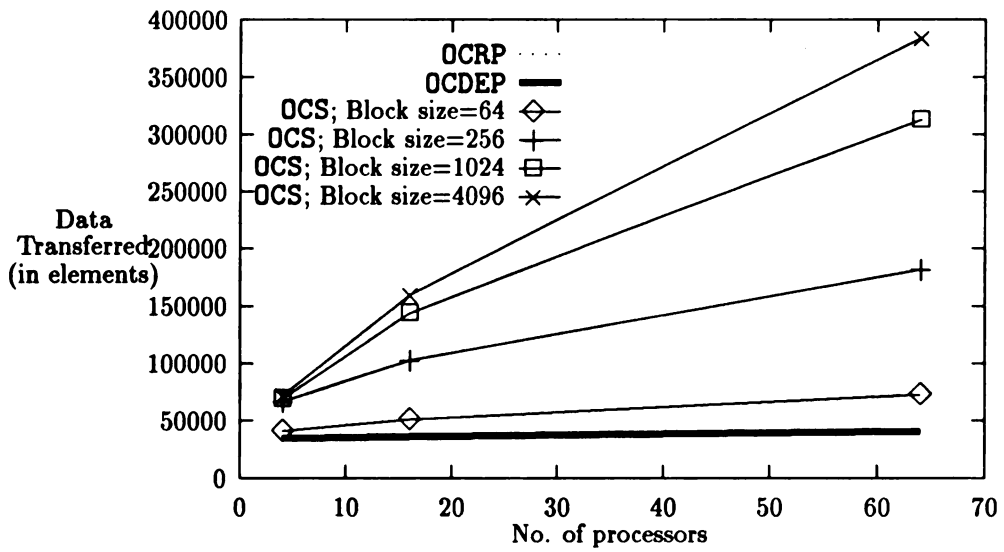

Figure 6.13. Data transferred vs. Number of processors (**parallel iterative solver**).

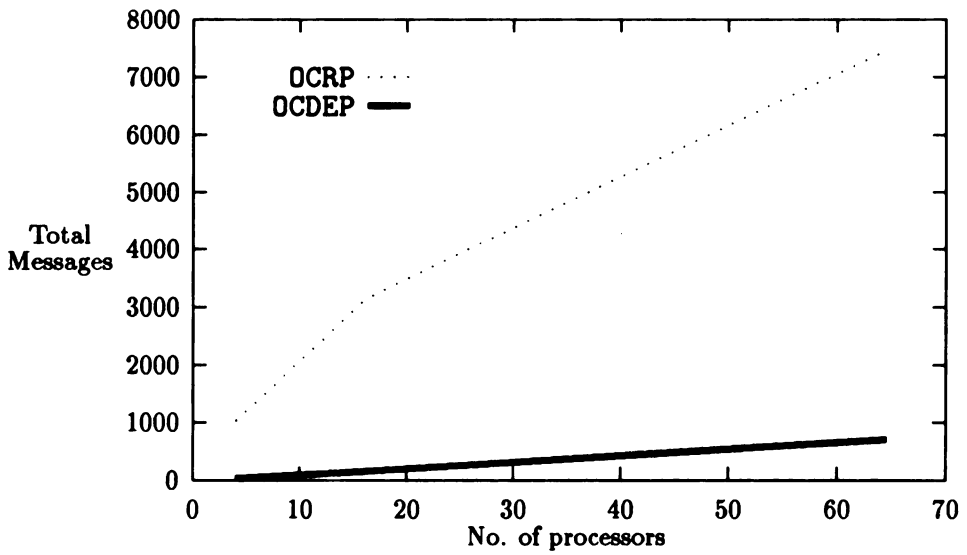

Figure 6.14. Total messages vs. Number of processors (**parallel iterative solver**; OCRP, OCDEP).

the same for both OCDEP and OCRP. Also, this amount is independent of the block size for both schemes. OCS transfers more data than OCDEP and OCRP because it incurs more consistency messages, and further, it unnecessarily transfers unreferenced data in falsely-shared blocks. For a given number of processors, OCS transfers more data for larger blocks because of the higher amount of false sharing. As shown in Figure 6.13, the amount of data transferred for the three schemes increases for more processors as in the case of application **matrix multiplication**.

Total Messages: Since consistency messages are fewer for OCDEP than for OCRP, and the data transfer messages for both schemes are equal, as shown in Figure 6.14, the total number of messages are fewer for OCDEP than OCRP. In addition, the total number of messages for both schemes are independent of the block size because its component consistency and data transfer messages are independent of the block size. Since consistency messages out number data transfer messages for OCS, the graph for the total number of messages shown in Figure 6.15 is similar to that for the number of consistency messages (Figure 6.10). It is seen that the total number of messages is the smallest for OCDEP and the largest for OCS for all values of the block size and

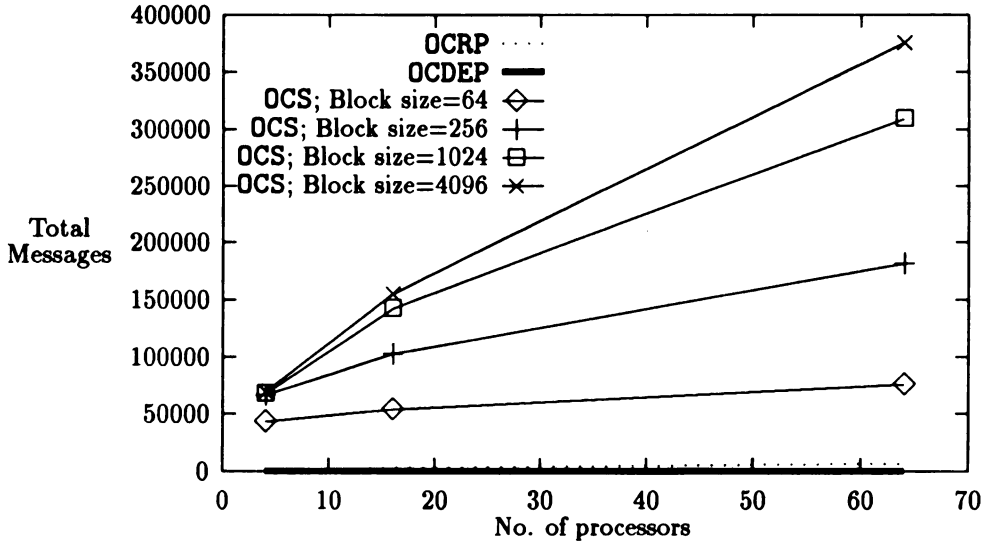

Figure 6.15. Total messages vs. Number of processors (**parallel iterative solver**).

the number of processors.

Total Objects: OCRP creates objects based on the reference pattern while OCDEP creates them based on the data exchange pattern. Unlike application **matrix multiplication**, due to the presence of read-write sharing of data in this application, OCRP creates more objects than OCDEP (Figure 6.16).

Physical Memory Requirements: OCS and OCDEP require the same amount of physical memory. In the case of OCRP, fragmentation is severe for large blocks in the case of the single-element objects. Hence, the physical memory required by OCRP is higher than that required by OCS and OCDEP, in spite of the unreferenced portions in the blocks created by the latter schemes. For small blocks however, fragmentation for OCRP is not severe, and it uses less physical memory than OCS and OCDEP. Figure 6.17 shows the variation of physical memory requirements for OCRP and OCS with the block size and the number of processors, which is similar to that for application **matrix multiplication**.

Virtual Memory Requirements: Figure 6.18 shows the variation of virtual memory requirements for OCRP and OCS with the block size and the number of processors.

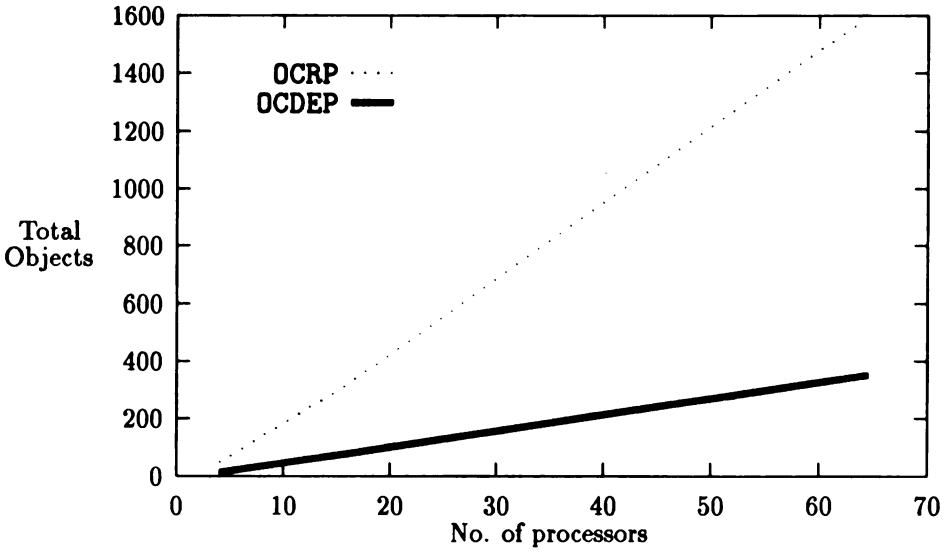

Figure 6.16. Total objects vs. Number of processors (**parallel iterative solver**; **OCRP**, **OCDEP**).

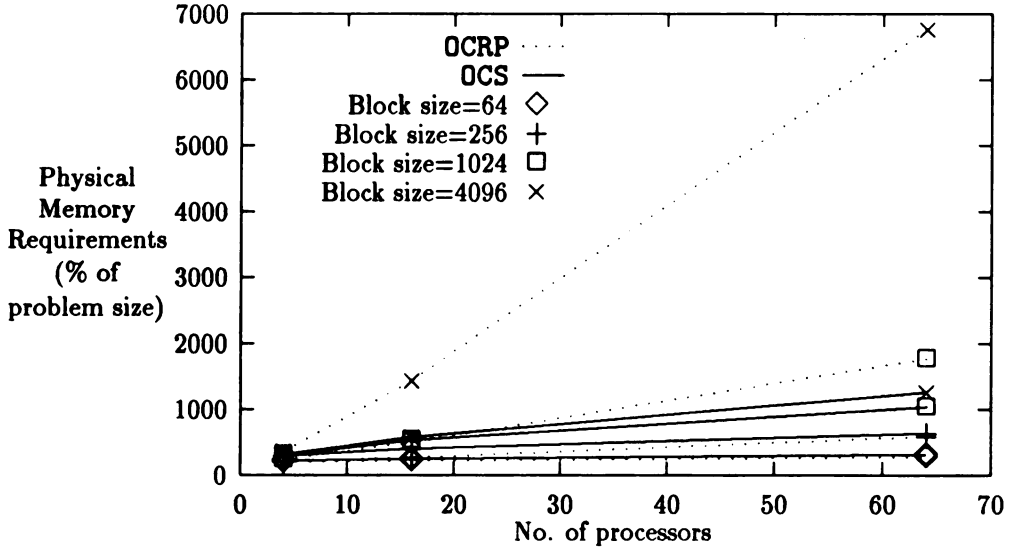

Figure 6.17. Physical memory requirements (% of problem size) vs. Number of processors (**parallel iterative solver**).

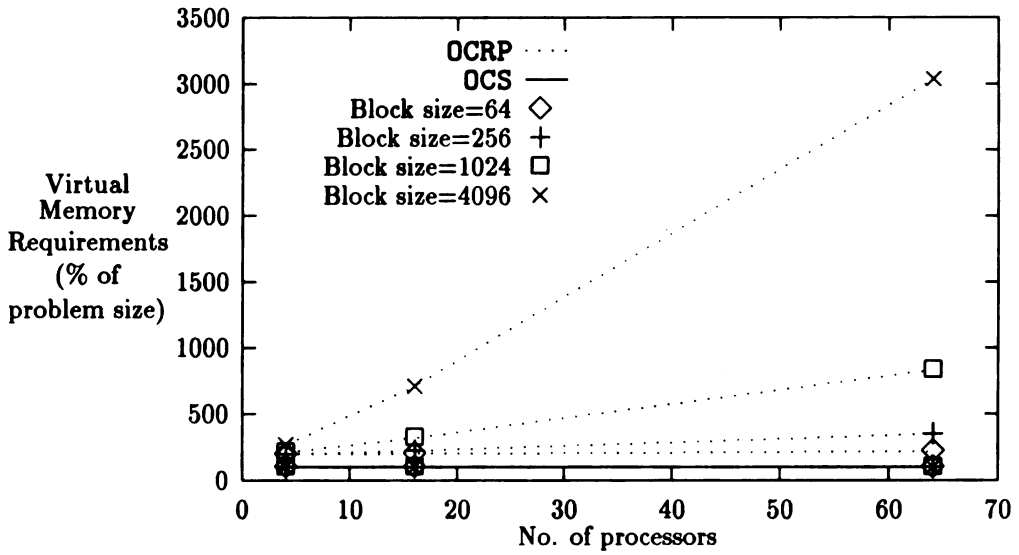

Figure 6.18. Virtual memory requirements (% of problem size) vs. Number of processors (**parallel iterative solver**).

For the parameters used in our experiments, the virtual memory needed by **OCS** and **OCDEP** is almost the same as the problem size, and is lower than that needed by **OCRP**. The potential for fragmentation is higher for larger blocks and more processors and hence, the virtual memory needed by **OCRP** is also correspondingly higher.

Summary: In terms of the total number of messages, **OCDEP** performs the best and **OCS** performs the worst. **OCRP** sends more messages than **OCDEP** due to the block-level consistency operations within the parallel code segment. Both **OCRP** and **OCDEP** transfer the same amount of data, which is less than that for **OCS**. **OCRP** transfers more objects than **OCDEP**, and requires more virtual and physical memory than **OCDEP** and **OCS**.

6.4.3 Application `multi-code-segment-array`

We choose our next application with the aim of comparing the three schemes for array variables with a different reference pattern in distinct code segments. A typical application usually consists of several such array variables. In the presence of an infinite

number of blocks however, the performance of each of them can be considered separately. Further, a multi-dimensional array variable is mapped to the SVM in a linear fashion, and is therefore equivalent to a one-dimensional array variable. Also, the performance of an array variable which occurs in more than two parallel code segments is characterized by its performance for each successive pair of code segments for which its reference pattern is different. Our next application **multi-code-segment-array** shown in Figure 6.19 consists of a one-dimensional array variable with one such pair of parallel code segments. An example of an application with such an array variable is the Burg algorithm [120] used in the area of image processing.

All elements of A are read in loops S_1 and S_3, while a fraction of the elements are written in loops S_2 and S_4. This fraction is characterized by the parameter *write frequency* (WF). For example, an element i is written if (i mod ($WF \times e$)) is zero, where e is the total number of array elements. As WF increases, fewer elements are written, and we vary WF to simulate various amounts of the consistency overhead for the falsely-shared blocks of **OCS**. We choose a value other than zero for k, the lower bound of I in loops S_3 and S_4 in order to change the reference pattern of A in loop S_3. For our experiments, we choose array references such that there is no read-write sharing of data within the parallel code segments. The results obtained can be extrapolated to the case when read-write sharing is present by using the results for application **parallel iterative solver**.

We conduct experiments for $e = 8192$, $k = 1000$, and various values of WF. We present the results for those values of WF which demonstrate both the case when the consistency overhead of false sharing is low enough to allow **OCS** to perform as well as **OCRP** and **OCDEP**, and the case when it is not. We partition the loops S_i, $1 \leq i \leq 4$ by assigning a consecutive set of values of the loop index I to each processor. **OCRP** creates one private read-only object per processor for loop S_1 and a private write-only object per processor for loop S_2. Since the set of elements in a given processor's object for loop S_2 is a subset of that in its object for loop S_1, no remap operations are necessary. A similar argument also applies for loops S_3 and S_4. The difference in the lower bound of I from loop S_2 to loop S_3 changes the reference pattern and

```
/* All referenced elements of A are read in loops S1 and S3.
Element A(I) is written in loops S2 and S4 if
I mod (WF * e) = 0 */

     DOSEQ
          Initialize
     END DOSEQ
     DOALL S1 I = 0 TO e-1
          ...  = A(I)
          .
          .
          .
S1   END DOALL
     DOALL S2 I = 0 TO e-1
          A(I) = ...
          .
          .
          .
S2   END DOALL
     DOALL S3 I = k TO e-1
          ...  = A(I)
          .
          .
          .
S3   END DOALL
     DOALL S4 I = k TO e-1
          A(I) = ...
          .
          .
          .
S4   END DOALL
     DOSEQ
          Terminate
     END DOSEQ
```

Figure 6.19. Application **multi-code-segment-array**.

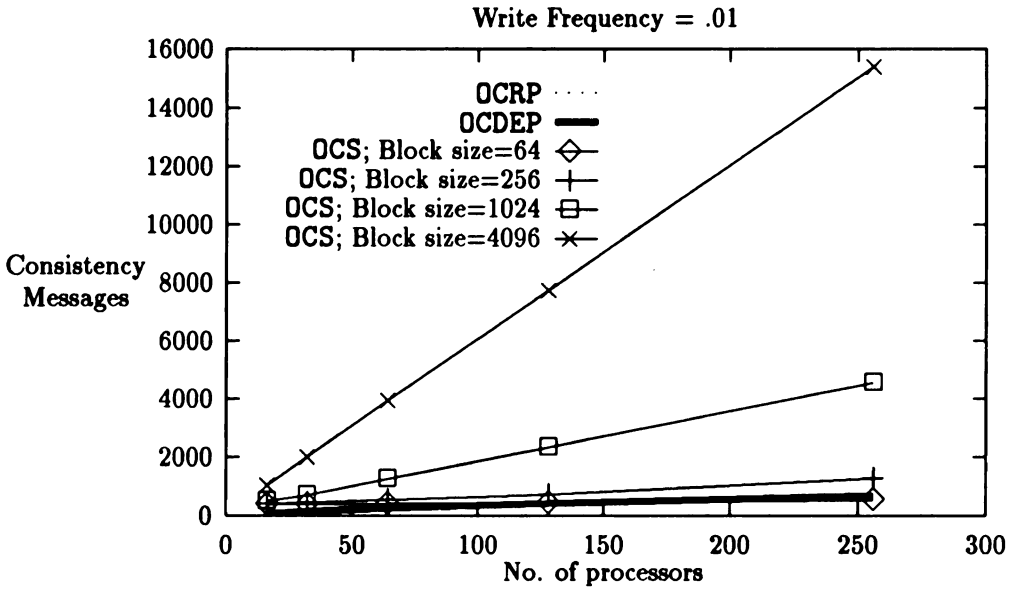

Figure 6.20. Consistency messages vs. Number of processors (**multi-code-segment-array**; WF=.01).

necessitates remap operations between these loops.

Consistency Messages: Since there is no read-write sharing of data in this application, in the case of both OCRP and OCDEP, consistency messages transfer objects between parallel code segments. They are almost equal in number for both the schemes, and further this number is independent of the block size. Consistency messages for OCS on the other hand maintain block-level sequential consistency for each write operation, and their number depends on the block size. Figure 6.20 shows how the number of consistency messages for the three schemes varies with the number of processors and the block size, when WF=0.01. For small blocks, all the three schemes have an almost equal number of consistency messages. As the block size increases however, due to the increase in false sharing, OCS incurs more consistency messages than the other two schemes. For a given block size, OCS incurs more consistency messages for a larger number of processors because of the increased sharing of elements across code segments. A larger number of processors also increases the number of consistency messages for OCRP and OCDEP because more processors write the same number of elements.

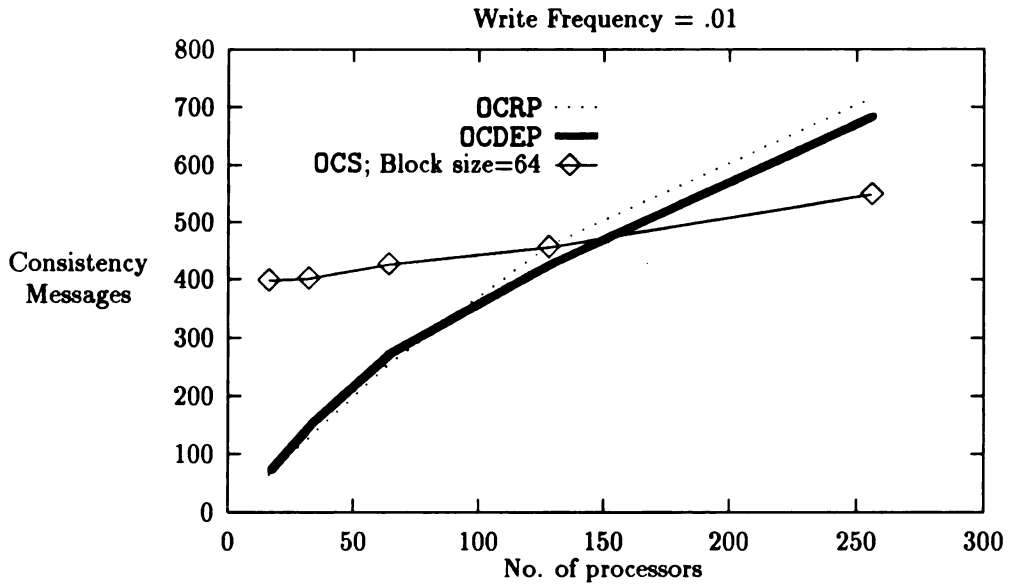

Figure 6.21. Consistency messages vs. Number of processors (**multi-code-segment-array**; WF=.01).

Figure 6.21 shows an expanded portion of the graph in Figure 6.20, where all three schemes have the same number of consistency messages. The consistency messages for **OCRP** are due to the remap operations (combine and split) between loops S_2 and S_3. When p processors are used and w of them write elements in loop S_2, the number of consistency messages for **OCRP** is $2 \times (w + p)$, where $2w$ and $2p$ messages are used for the combine and the split operations, respectively. The consistency messages for **OCDEP** include the messages needed to update elements that are read in loop S_3. Since consecutive elements of the array A are assigned to each processor, each of the w processors will send elements to at most two other processors. Also, elements that are not written in loop S_2 and for which there are no local copies need to be fetched from the server processor, and let us assume that s processors need such data transfers. The number of consistency messages for **OCDEP** is therefore $2 \times (2w + s)$.

For a small number of processors, the number of elements per processor is large. The probability of a processor writing an element in loop S_2 is high, and $w \sim p$. Hence, the number of consistency messages for **OCDEP** is higher than that for **OCRP**. For a large number of processors, the number of elements per processor is small. The

Table 6.1. Consistency messages (**multi-code-segment-array**; OCRP, OCDEP).

No. of processors	WF=.01		WF=.256		WF=.6	
	OCRP	OCDEP	OCRP	OCDEP	OCRP	OCDEP
16	64	72	40	36	36	32
32	128	150	72	68	68	64
64	256	274	136	132	132	128
128	460	426	264	260	260	256
256	716	684	520	516	516	512

probability of a processor writing an element in loop S_2 is small, and $w << p$. Hence, the number of consistency messages for OCDEP is lower than that for OCRP.

As shown in Table 6.1, the actual difference in the number of consistency messages for OCRP and OCDEP is quite small for the entire range of processors. OCRP requires that objects be combined and split in a server processor. A single server will prove to be a bottleneck and hence it is necessary to use multiple servers. The traffic in the case of OCDEP on the other hand is dependent on the nature of the data exchange pattern in the application.

Figures 6.22 and 6.23 show how the number of consistency messages varies with the block size and the number of processors for WF=0.256 and WF=0.6, respectively. Since fewer elements are written for higher values of WF, OCS incurs lower consistency overhead due to false sharing. Hence, the number of consistency messages for OCS is fewer than that for the other two schemes for all but one block size when WF=0.256, and for all block sizes when WF=0.6. Some of the consistency messages for OCDEP and OCRP however are used to update data which is transferred using data transfer messages in the case of OCS. Again, as shown in Table 6.1, the actual difference in the number of consistency messages for OCRP and OCDEP is quite small for the entire range of processors. OCDEP incurs fewer consistency messages than OCRP for all values of the number of processors when WF=0.256 and WF=0.6, because $w << p$ for all these cases.

Data Transfer Messages: Since both OCRP and OCDEP apply optimizations to re-

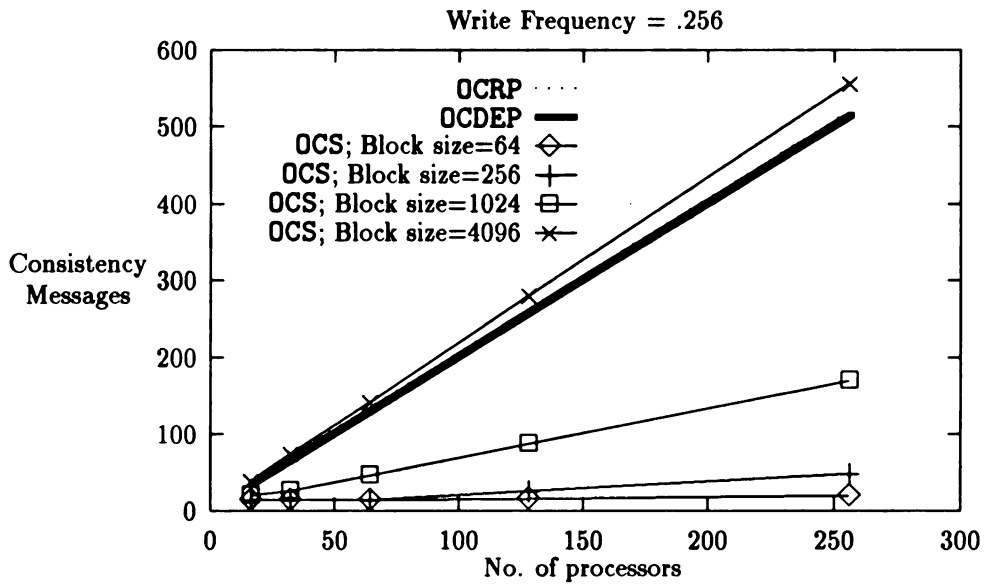

Figure 6.22. Consistency messages vs. Number of processors (**multi-code-segment-array**; WF=.256).

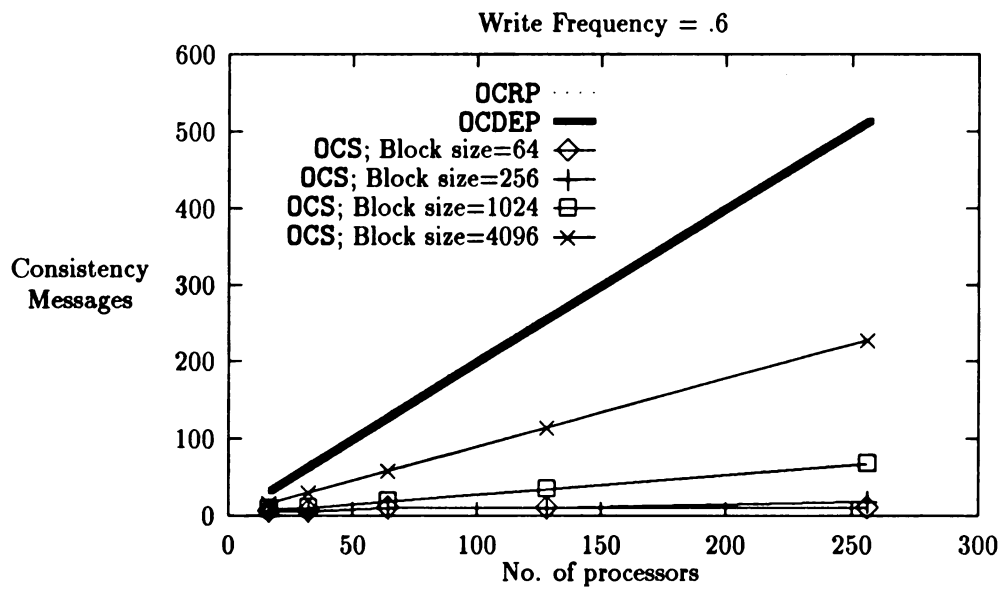

Figure 6.23. Consistency messages vs. Number of processors (**multi-code-segment-array**; WF=.6).

Table 6.2. Data transfer messages (**multi-code-segment-array**; OCRP, OCDEP).

No. of processors	WF=.01		WF=.256		WF=.6	
	OCRP	OCDEP	OCRP	OCDEP	OCRP	OCDEP
16	64	64	38	40	34	36
32	128	128	70	72	66	68
64	256	256	134	136	130	132
128	434	440	262	264	258	260
256	690	704	518	520	514	516

duce the number of data transfer units, they have almost the same number of data transfer messages (Table 6.2), and further this number is independent of the block size. The data transfer messages for both schemes transfer data from the server to the other processors before loop S_1. In addition, for OCRP, they transfer elements that are modified in loop S_4 to the server (combine operation). We however consider messages which transfer elements that are modified in loop S_2 to the server as consistency messages in the case of OCRP. In the case of OCDEP, data transfer messages transfer elements which are last modified in either loop S_2 or S_4 to the server before the termination code segment. For a large number of processors, the number of data transfer messages for OCDEP is hence slightly higher than that for OCRP.

The data transfer messages for OCS transfer non-local blocks that contain referenced elements, and therefore for a given number of processors, they are higher for smaller blocks. Figure 6.24 shows how the number of data transfer messages for the three schemes varies with the number of processors and the block size, when WF=0.01. Due to prefetching, for large blocks, the number of data transfer messages for OCS is comparable to that for OCRP and OCDEP. For small blocks however, OCS needs a higher number of data transfer messages than the other schemes. For a given block size, the number of data transfer messages for OCS is higher for more processors, because of the increased sharing of each block. The number of data transfer messages for OCRP and OCDEP also increases for more processors because additional data transfer units need to be sent to the extra processors. Plots for WF=.256 and

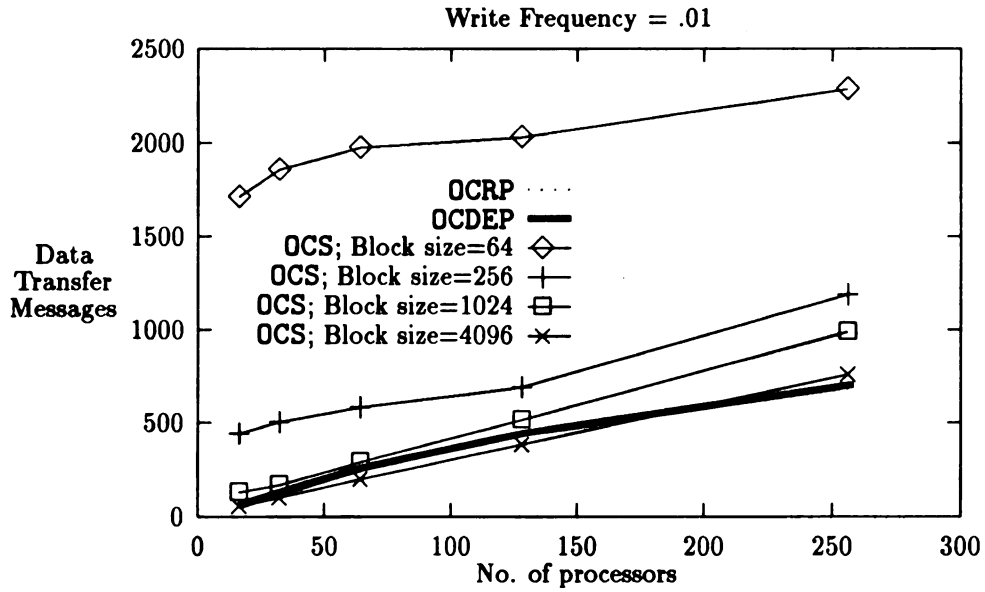

Figure 6.24. Data transfer messages vs. Number of processors (**multi-code-segment-array**; WF=.01).

WF=0.6 also exhibit the same behavior.

Data transfer messages for OCRP and OCDEP are fewer for a higher WF because fewer elements are written and need to be updated in the server processor. The number of these messages for OCS however is independent of the value of WF because this value does not affect the number of blocks containing elements read in loops S_1 and S_3, which are transferred using these messages. OCRP and OCDEP are able to adapt the number of data transfer messages to the application's characteristics, but OCS lacks this ability. Figure 6.25 shows how the number of data transfer messages for OCDEP varies with WF. It also contains plots for this number in the case of OCS for two of the larger block sizes. We omit the plots for OCRP because they are similar to those for OCDEP.

Amount of Data Transferred: The amount of data transferred includes that for both the consistency and the data transfer messages. OCDEP transfers the minimum amount of data because it updates data elements only when needed. OCRP transfers more data than OCDEP during consistency messages because of the remap operations (Table 6.3). For example, in the case of OCRP, elements that are written in loop S_2 are

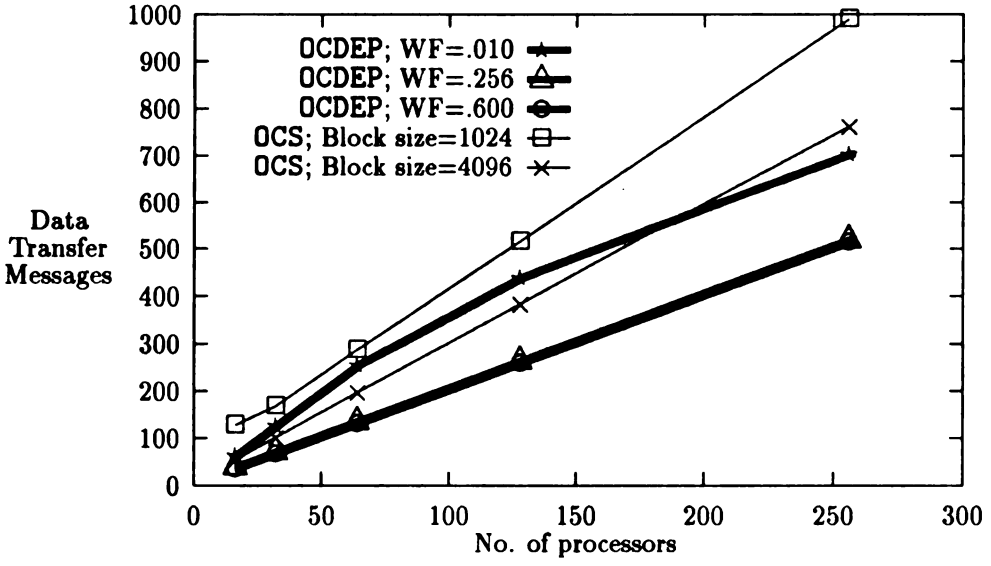

Figure 6.25. Data transfer messages vs. Number of processors (**multi-code-segment-array**; WF=.01, .256, .6).

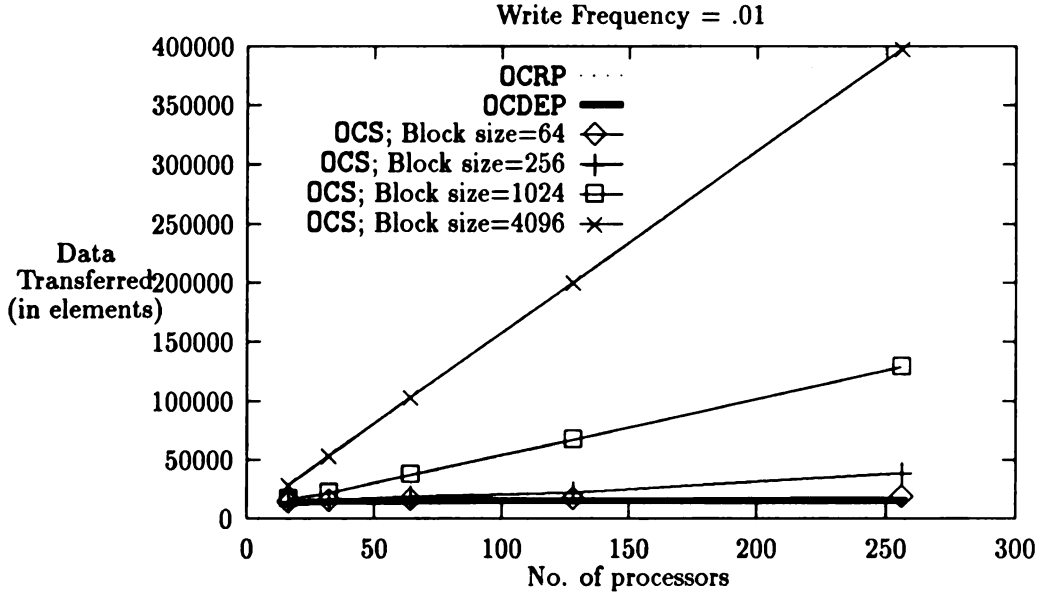

Figure 6.26. Data transferred vs. Number of processors (**multi-code-segment-array**; WF=.01).

Table 6.3. Data transferred (in elements) (**multi-code-segment-array**; OCRP, OCDEP).

No. of processors	WF=.01		WF=.256		WF=.6	
	OCRP	OCDEP	OCRP	OCDEP	OCRP	OCDEP
16	15575	13622	15391	13524	15387	13522
32	15575	14554	15391	14456	15387	14454
64	15575	15002	15391	14904	15387	14902
128	15575	15225	15391	15127	15387	15125
256	15575	15331	15391	15233	15387	15231

updated to the server after loop S_2 (combine operation). The server then transfers all the elements of A to the various processors based on the reference pattern of A for loop S_3 (split operation). OCDEP on the other hand transfers only elements that are written in loop S_2 to processors that do not have a local copy and read them in loop S_3. The amount of data transferred by OCS depends on the block size which determines the amount of false sharing of each block.

Figure 6.26 shows how the amount of data transferred for the three schemes varies with the number of processors and the block size, when WF=0.01. OCRP and OCDEP transfer almost the same amount of data relative to that transferred by OCS, and this amount is independent of the block size. Since consistency messages for OCS are much higher than data transfer messages, their contribution to the amount of data transferred is also higher. Due to more consistency messages, for larger blocks, OCS transfers more data than OCRP and OCDEP. For small blocks, all three schemes transfer comparable amounts of data. For a given number of processors, OCS transfers more data for larger blocks because of the increase in false sharing. For a given block size, OCS transfers more data for a higher number of processors because of the increased sharing of data elements. Plots for other given values of WF (.256 and 0.6) are similar.

OCRP transfers all the elements of A before loops S_1 and S_3, and the elements written in loops S_2 and S_4 after each of these loops, respectively. The amount of data transferred by OCRP is therefore independent of the number of processors. OCDEP

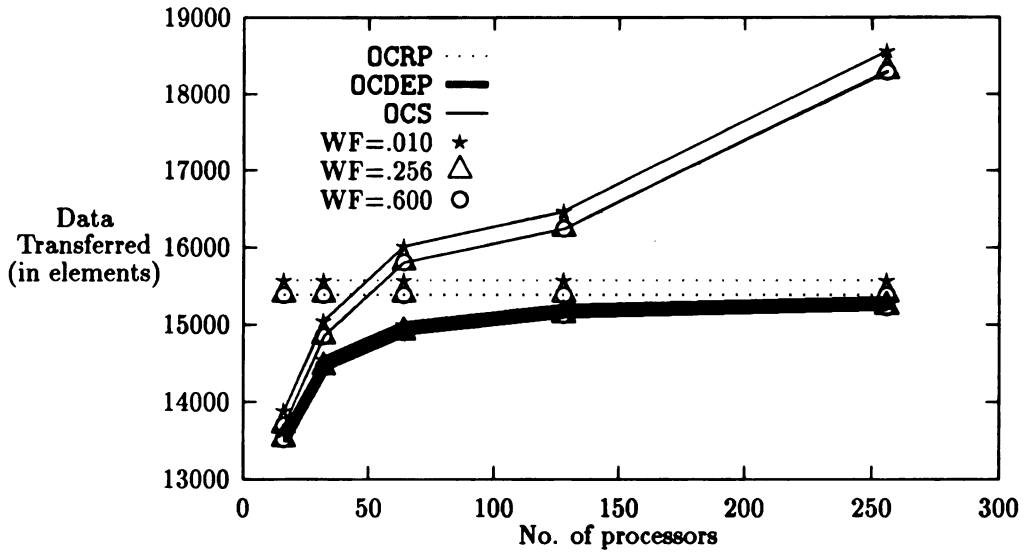

Figure 6.27. Data transferred vs. Number of processors (`multi-code-segment-array`; WF=.01, .256, .6).

transfers more data for a higher number of processors because elements are now written by a larger number of processors. The amount of data transferred in the case of `OCRP` and `OCDEP` for various number of processors and values of WF is shown in Table 6.3.

Figure 6.27 shows the variation of the amount of data transferred for the three schemes with the number of processors and the WF. In the case of `OCS`, we plot only the data for the minimum block size of 64 bytes when the amount of data it transfers is comparable to that for the other schemes. All schemes transfer less data for higher values of WF because fewer elements are written. In all cases, `OCDEP` transfers the least amount of data and `OCS` transfers the maximum amount of data.

Total Messages: The total number of messages is the sum of the number of the data transfer and the consistency messages. In the case of `OCS`, large blocks cause more consistency messages while small blocks result in more data transfer messages. Hence, a tradeoff exists between false sharing and prefetching as the block size is increased. Since consistency messages account for a dominant fraction of the total number of messages when WF=.01 for large blocks, the plots for the latter in Figure 6.28 are

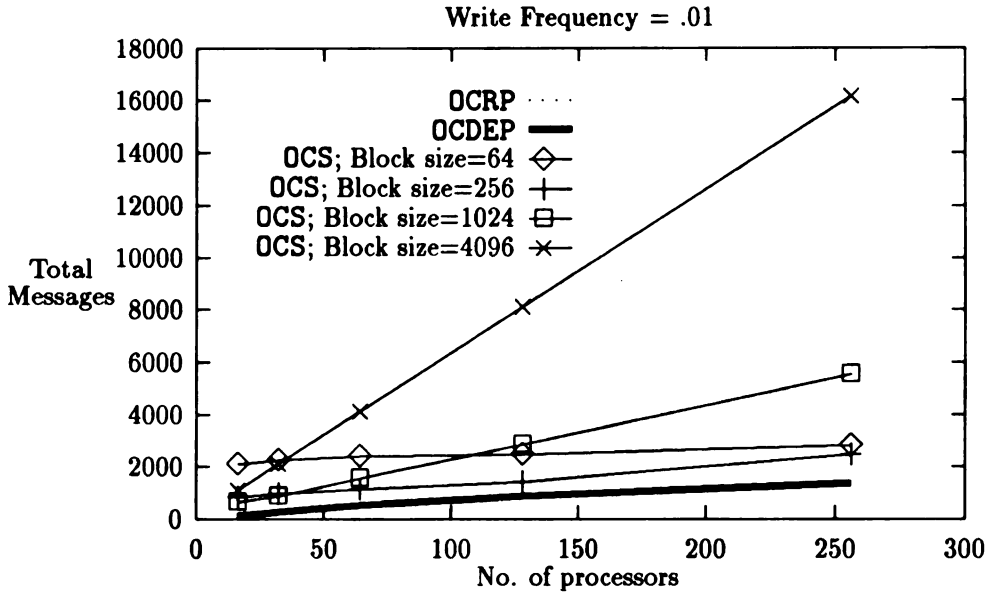

Figure 6.28. Total messages vs. Number of processors (**multi-code-segment-array**; WF=.01).

similar to those for the former in Figure 6.20. As WF increases, fewer elements are written, and the fraction of the total number of messages which is contributed by data transfer messages becomes higher. The plots for the total number of messages when WF=0.6 shown in Figure 6.30 are similar to the corresponding plots for the data transfer messages. When WF=.256, the plots for the total number of messages shown in Figure 6.29 follow the corresponding plots for both the consistency and the data transfer messages.

When the consistency overhead due to false sharing is low (WF=.600, when only one element is written), large blocks facilitate prefetching, and hence, OCS performs as well as OCDEP and OCRP. For most of the values of WF however ($\leq$ .256, when more than one element is written), OCDEP and OCRP perform better than OCS. Note however that our performance measure does not include the runtime overhead of adaptive placement which is higher for OCS than for OCDEP and OCRP. Hence, we expect OCRP and OCDEP to provide a better runtime performance than OCS.

The total number of messages for OCRP and OCDEP are almost the same as shown in Table 6.4. The total number of messages for OCDEP is higher than that for OCRP for a

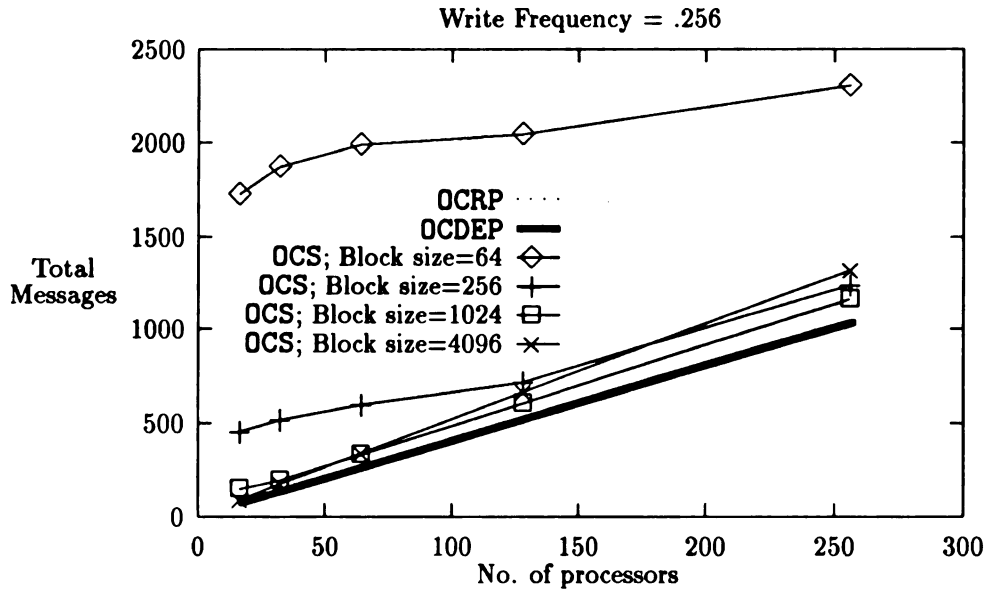

Figure 6.29. Total messages vs. Number of processors (**multi-code-segment-array**; WF=.256).

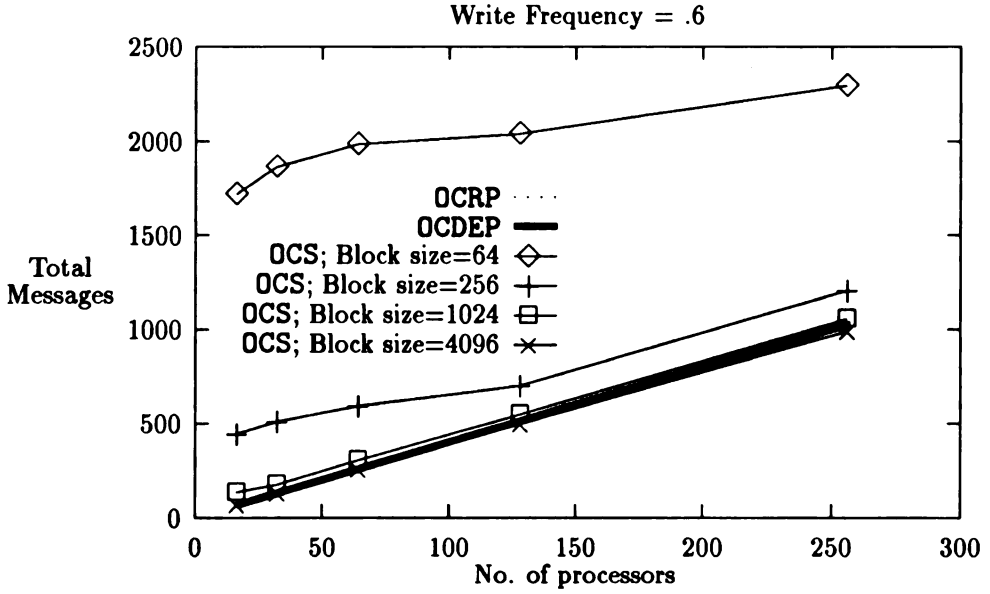

Figure 6.30. Total messages vs. Number of processors (**multi-code-segment-array**; WF=.6).

Table 6.4. Total messages (**multi-code-segment-array**; OCRP, OCDEP).

No. of processors	WF=.01		WF=.256		WF=.6	
	OCRP	OCDEP	OCRP	OCDEP	OCRP	OCDEP
16	128	136	78	76	70	68
32	256	278	142	140	134	132
64	512	530	270	268	262	260
128	894	866	526	524	518	516
256	1406	1388	1038	1036	1030	1028

Table 6.5. Total objects (**multi-code-segment-array**; OCRP, OCDEP).

No. of processors	WF=.01		WF=.256		WF=.6	
	OCRP	OCDEP	OCRP	OCDEP	OCRP	OCDEP
16	64	68	39	38	35	34
32	128	139	71	70	67	66
64	256	265	135	134	131	130
128	447	433	263	262	259	258
256	703	694	519	518	515	514

small number of processors and low value of WF, and it is lower in all the other cases. The total number of messages is the sum of the data transfer and the consistency messages, and its behavior can be explained based on our discussion earlier about its component messages.

Total Objects: The total number of objects for OCDEP and OCRP for various values of WF and the number of processors is shown in Table 6.5. Both schemes use objects to transfer data during consistency and data transfer operations. Both schemes apply the same optimizations to reduce the number of objects and data transfer units and hence, they transfer similar objects during data transfer operations. Moreover, the objects transferred by OCRP during remap operations are similar to those transferred by OCDEP during the consistency operations. Hence, the number of objects for both schemes is almost the same for all cases. In contrast, due to the presence of read-write

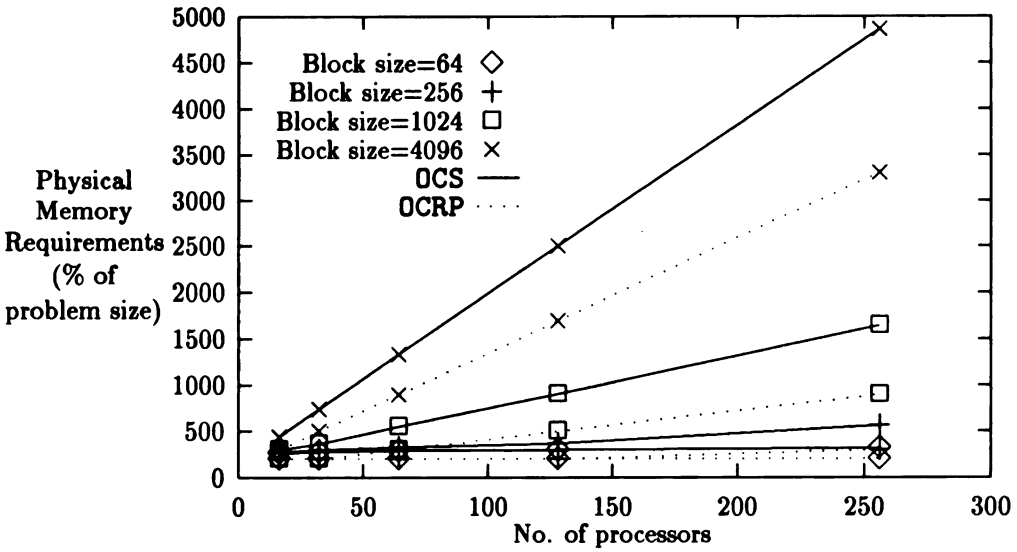

Figure 6.31. Physical memory requirements (% of problem size) vs. Number of processors (**multi-code-segment-array**).

sharing of data, for the application **parallel iterative solver**, OCRP creates more objects than OCDEP. Note that the behavior of this performance measure is similar to that of the total number of messages.

Physical Memory Requirements: All schemes need the same amount of physical memory to allocate the application's data in the server's local memory. In addition, since we consider an infinite number of blocks and do not replace blocks, OCS and OCDEP need physical memory for each of the parallel code segments. OCRP on the other hand needs additional memory equal to the maximum out of that needed for each of the parallel code segments. Hence, OCRP requires less physical memory than OCS as shown in Figure 6.31, which shows how the physical memory requirements for OCS and OCRP varies with the block size and the number of processors. For a given number of processors, both schemes need more physical memory for larger blocks. The additional memory needed is due to the higher amount of fragmentation for OCRP and the larger number of unreferenced elements per block in the case of OCS. For a given block size, both schemes need more physical memory for more processors. The additional memory needed is due to the increased fragmentation of blocks for

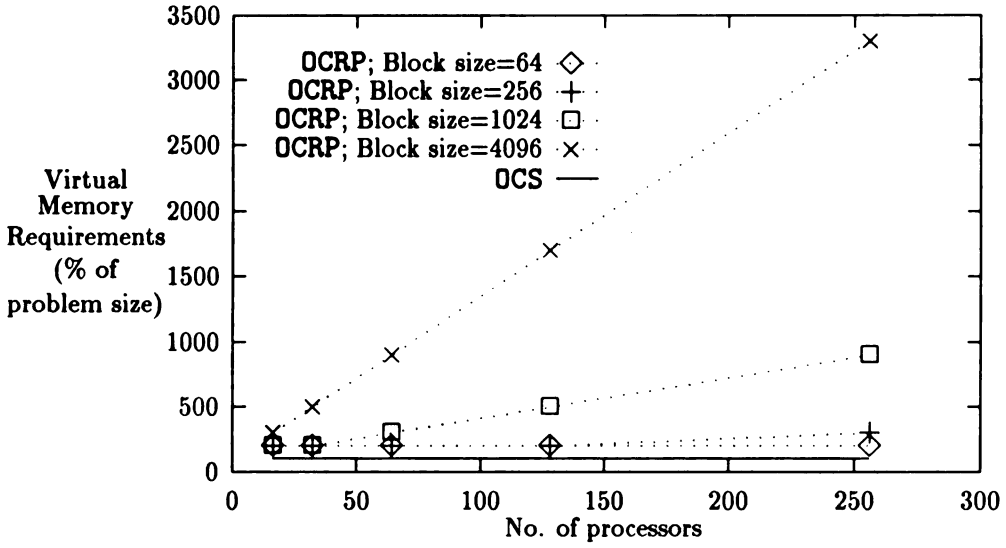

Figure 6.32. Virtual memory requirements (% of problem size) vs. Number of processors (**multi-code-segment-array**).

OCRP, and the higher number of processors referencing each block for OCS.

Virtual Memory Requirements: Figure 6.32 shows how the virtual memory requirements for OCS and OCRP varies with the block size and the number of processors. As expected, the needs for OCS are close to the problem size, while that for OCRP are much higher. For a given number of processors, OCRP requires more virtual memory for larger blocks because of increased fragmentation. For a given block size, it needs more virtual memory for more processors because of the fewer elements in each processor's object and the consequent increase in fragmentation.

Summary: When the consistency overhead due to false sharing is extremely low, the performance of all the three schemes in terms of the total number of messages is comparable. In all other cases, OCDEP and OCRP perform better than OCS. Due to the absence of consistency operations within code segments, the performance of OCDEP and OCRP in terms of the total number of messages and objects is comparable. In terms of the amount of data transferred, OCDEP performs the best, followed by OCRP, and then OCS. OCRP requires more virtual memory than both OCS and OCDEP. For our experimental parameters, its physical memory requirements are less than that for the

Table 6.6. Summary of experimental results.

Application Characteristics				Performance Rank		
False sharing of blocks with OCS?	True sharing of blocks?	Consistency needed within code segment?	Consistency needed between code segments?	OCS	OCRP	OCDEP
No	No	No	No	1	1	1
No	Yes	No	No	2	2	1
No	Yes	Yes	No	1	1	1
Yes	No	No	No	2	1	1
Yes	Yes	No	Yes	3	2	1
Yes	Yes	Yes	Yes	3	2	1
Yes	Yes	Yes	No	2	1	1
Yes	Yes	No	No	2	2	1

other two schemes.

6.4.4 Conclusions

Our experiments highlight the manner in which various characteristics of an application can influence its performance for these data placement schemes. Based on our experimental results, we rank the performance of the placement schemes for various combination of these characteristics as shown in Table 6.6. Our performance rank is based on the total number of messages and the amount of data transferred. OCDEP performs the best of all the schemes, thereby verifying our claim that the compiler can best help in data placement if it is allowed to specify when data placement operations are necessary.

OCDEP performs better than OCRP because it directs rather than assists data placement, and OCRP can also be used in a similar manner. For example, in the case of the application **parallel iterative solver**, OCRP can specify that consistency need not be maintained within the parallel code segments. Between parallel code segments, optimizations can be applied to combine together objects that need to be transferred between a given pair of send and receive processors. The performance rank for this

Table 6.7. Compiler-directed data placement (OCRP vs. OCDEP).

Performance Measure	OCRP	OCDEP
Possible unused physical memory	Only in last block	In all blocks
Virtual memory needs	Problem size + Objects	Problem size
Code generation	New, more complex algorithms	Existing algorithms

application in the case of both OCRP and OCDEP will then be the same. Table 6.7 compares the other performance measures when compiler-directed object-level placement is provided using either OCRP or OCDEP. OCRP requires additional virtual memory and new code generation algorithms. Except when fragmentation is severe, OCRP uses less physical memory than OCDEP because it creates objects with temporal locality. The choice between these schemes for compiler-directed object-level placement therefore depends on the available physical and virtual memory, and the ease of generating code when using OCRP.

In related work, Chen and Veidenbaum [121] use trace-driven simulations to compare two cache-line-level placement schemes. One is a directory-based hardware scheme (without any compiler assistance) and the other is a simple software-assisted scheme [108]. They conclude that both schemes perform the same, and that the hardware scheme suffers from false sharing while the software scheme suffers from unnecessary invalidations. They argue that sophisticated software schemes which reduce the number of invalidations will perform better and at the same time cost less than hardware schemes. In contrast, we compare schemes with varying degrees of compiler control (software assistance) over data placement operations by compiling with various object-creation schemes. Further, while they consider only block-level placement, we study placement in terms of blocks as well as objects.

6.5 Summary

In this chapter, we experimentally compared various placement schemes which differ in the degree to which the compiler controls data placement operations. We established that the placement scheme which entrusts the compiler with the complete responsibility of data placement (compiler-directed object-level placement) performs the best of all the placement schemes. Further, by transferring data in terms of objects rather than blocks, this scheme can potentially perform better than software-assisted, block-level schemes. We conclude that compiler-directed object-level placement shows promise as the approach to automate data placement and make it easier to use shared-virtual-memory NUMA multiprocessors.

CHAPTER 7

CONCLUSIONS

In this thesis, we attempted to develop new and better techniques to place the data of a parallel application in the physical memory of a multiprocessor with a non-uniform memory access (NUMA) time. We assume that the parallel application uses as many processes as the available number of processors, and that each process is statically assigned to a given processor. Further, processes communicate and synchronize by means of a shared virtual memory (SVM). In this chapter, we summarize the major contributions of our thesis, and provide directions for future research.

7.1 Major Contributions

Existing techniques place data in terms of basic data blocks such as cache lines or pages. Using analysis and trace-driven simulations, we studied the relationship among page-level replication, hardware parameters, the layout of data in the SVM, and page reference patterns. We demonstrated the need for page-level replication to be dependent on hardware parameters and the reference pattern of pages. We also showed that proper data layout reduces or, in some cases, even eliminates false sharing of pages. Moreover, it simplifies the reference pattern and the placement of pages. These results apply to other block-level placement strategies as well. In the absence of help from the compiler or the applications programmer however, these strategies incur runtime overhead when being adaptive and further, are unable to optimally place blocks which are falsely-shared and those with rapidly-changing reference patterns.

Motivated by the conclusions of our study, we developed a new approach for compiler-assisted data placement. Our approach helps data placement by creating compile-time objects that contain data of the same variable type and similar reference patterns. We outlined the design of a compiler which uses this approach. Our compiler allows applications to be written either in a sequential or a parallel language. (In the former case, the compiler parallelizes applications using techniques in existing parallelizing compilers [71, 72, 73, 74, 75]). Our compiler assumes static control-partitioning of the iterations of each parallel code segment. It uses partitioning heuristics for well-known loop constructs such as iterative parallel loops [80], but requires the applications programmer to specify the partitioning for the other cases. We also developed new algorithms to determine the data reference pattern and the type of each variable which are required by our object-creation schemes. These schemes are different for various types of data placement.

We first developed a new object-creation method that assists block-level placement by creating objects that are not falsely-shared and have temporal locality. We demonstrated how these objects can be used to eliminate false sharing and reduce the runtime overhead of adaptive placement through hints. We also developed new code generation algorithms that are required when applying our scheme. Further, we proposed solutions which can be used when our method fragments virtual and physical memory. We derived the time and space overhead involved in compiling applications using our method. We identified the factors that affect the performance offered by our scheme. We concluded that the additional runtime overhead incurred by our method is small when compared to the improvement in performance it achieves by addressing the limitations of block-level placement schemes. Moreover, the performance improvement is limited because block-level placement schemes invoke placement operations unnecessarily without allowing the compiler to specify when they are required.

We then considered object-level placement schemes which require the compiler or the applications programmer to specify data placement operations. In order to direct object-level placement, we developed a new object-creation scheme which creates objects based on the data exchange pattern among the processes of the parallel

application. As in the case of block-level placement, we derived the time and space overhead involved in compiling applications using this method. We discussed why compiler-directed object-level placement performs better than block-level placement. By modeling the execution of an application when object-level placement is provided, we identified the performance optimizations possible when transferring objects.

Finally, we conducted a detailed performance study using experimental simulations to quantitatively compare various placement schemes, each of which allow a different degree of compiler control over data placement. We chose a workload that highlights the cases when a higher degree of compiler control improves performance and when it does not. Our results clearly demonstrate that block-level placement with compiler assistance performs significantly better than that without compiler assistance. In one application for instance, the number of consistency messages in the absence of compiler assistance is several orders of magnitude higher than that in the presence of such help. The main reason for this performance improvement is the ability of our object-creation scheme that assists block-level placement to eliminate the consistency overhead due to false sharing. Our study also showed that compiler-directed object-level placement performs as well as or better than compiler-assisted block-level placement. For example, in the case of another application in our workload, the number of consistency messages for the former is an order of magnitude less than that for the latter. This performance improvement is achieved because our object-creation method (that directs object-level placement) avoids unnecessary data placement operations. By identifying the application characteristics of our workload, we concluded that these results are true for applications with various combinations of these characteristics.

7.2 Future Directions

The results of our thesis can be extended as described below.

Our object-creation schemes use the reference pattern of variables in each code segment. In Chapter 3, we developed algorithms to determine the reference pat-

tern when the parallelism in a code segment is specified using the DOALL and the DOACROSS programming constructs, assuming that the subscripts are linear. Algorithms need to be developed for other parallel constructs and for non-linear subscripts. The basic concept behind our existing algorithms can be applied for these cases as well, provided the work done by each processor is known at compile-time. More efficient algorithms than those proposed here will help further reduce the compilation overhead in applying our schemes.

Our approach to compiler-assisted data placement assumes static control-partitioning of the work in each code segment. Our compiler uses heuristics to control-partition well-known parallel constructs. Since it requires the applications programmer to control-partition other cases, it is important to make this task easy. This objective can be achieved by incorporating our object-creation algorithms in a programming environment. Such an environment can display the objects created for each code segment, provide an estimate of the fragmentation of virtual and physical memory for various block sizes, and provide an estimate of the performance. Further, partitioning heuristics for various loop constructs which are developed using such an environment can be incorporated in our compiler.

When compiling applications using our object-creation scheme for block-level placement, new algorithms are needed to address the issues of fragmentation and code generation for static array references. In Chapter 4, we developed code generation algorithms for the case of linear subscripts. Algorithms need to be developed for the other cases, and the solutions we proposed for fragmentation need to be implemented.

Our object-creation scheme for object-level placement outlined in Chapter 5 assumes that at most one processor writes a data element within each code segment. In those applications for which this condition is not true, it is necessary to develop algorithms that use the data dependence graph to establish an order on the writes within each code segment.

In our experimental comparison of different placement schemes, we considered placement at a single level of the memory hierarchy with an infinite number of physical

blocks. This assumption is typical of studies on memory performance e.g., [51]. Our object-creation schemes, however, are designed to assist placement at all levels of the memory hierarchy. Further, due to price/performance reasons, we expect future multiprocessors to have several levels of the memory hierarchy. It would be interesting to extend our experimental study to the case of data placement at all levels of the memory hierarchy, assuming that each level has only a finite number of physical blocks. We expect the placement scheme that provides the compiler with the highest control over placement operations to perform the best even under these conditions.

Our object-creation schemes for both the block-level and the object-level types of placement need to be incorporated in a compiler, the design of which was outlined in Chapter 3. The runtime primitives required by our schemes also need to be implemented. Such an implementation will allow estimation of the additional time required for compilation when applying our object-creation schemes. It will also allow measurements of execution time for various applications when using the different placement schemes. We expect the results of these measurements to concur with the conclusions of our experimental study.

Compiler-assisted data placement has also been studied by researchers for message-passing NUMA multiprocessors, also referred to as distributed-memory message-passing multiprocessors [96, 97, 98, 99, 74, 83, 100, 81, 82]. In these studies, data is partitioned among the local memories of the various processors, and then the work to be done by each processor is decided. In contrast, our approach is to control-partition the work and to create objects which are then carefully placed to minimize the time spent in memory references. These approaches differ in the process-interaction paradigm, but attempt to use compiler help to achieve the same objective of reducing the performance degradation due to non-local data accesses. It would be interesting to develop a methodology to compare these two approaches and determine which is appropriate under what conditions.

In summary, in this thesis, we have identified the factors that limit the performance of existing block-level data placement strategies. We have developed a methodology for compiler-assisted block-level placement which improves performance by address-

ing these limitations. We have concluded that the performance can be improved even further if the compiler is allowed to direct rather than assist data placement, as provided by compiler-directed object-level placement. We have developed a method to assist object-level placement, and demonstrated that this method indeed provides the best performance. Due to price/performance reasons, we expect future multiprocessors to belong to the NUMA class. Results such as ours on compiler-directed data placement are therefore important steps towards improving the performance of applications and the programmability of NUMA multiprocessors.

APPENDICES

APPENDIX A

Glossary

Array reference: A reference to an array variable in a program statement.

Block: Unit of transfer in a given level of the memory hierarchy, e.g., cache line or page.

Block-level migration: A data placement scheme that migrates the virtual block containing a data element.

Block-level placement: A data placement scheme that allocates physical memory for the virtual block containing a data element.

Block-level replication: A data placement scheme that replicates the virtual block containing a data element.

Code segment: A set of statements in the program of an application.

Combine operation: An operation that combines several objects into a single object as per specifications.

Consistency model: Definition of the relationship among the various physical copies of a given data element or block.

Control partitioning: Division of the work in a parallel code segment among available processors.

Data dependence: The relationship between two program statements that read or write the same variable.

Data dependence analysis: A method to extract information about the data dependencies of an application.

Data exchange pattern: Pattern of communication and transfer of data among the processors executing a single parallel application.

Data layout: Map of data to locations in the shared virtual memory.

Data partitioning: Distribution of data among the local memories of available processors.

Data placement: Allocation of a physical memory location for a data element.

Data reference: A reference to any data element in a program statement.

Data reference pattern: A temporal order of the read and the write references to a data element originating from different processors.

Falsely-shared block: A block which contains data elements with different reference patterns.

Internal fragmentation: Condition in which blocks are only partially used.

Layout of data: See **data layout**.

Linear subscript: A subscript of the form $ai+b$, where a and b are integer constants and i is a loop induction variable.

Migration: Movement of a single physical copy of the data among the local memories of the processors that reference it.

Multiprocessor: A parallel computer which consists of multiple processors and memory modules connected together by one or more interconnects.

NUMA multiprocessor: A multiprocessor in which the time to access different memory locations is not the same.

Object: A collection of data elements, with similar reference patterns, and treated as a unit.

Object-level placement: A data placement scheme that allocates physical memory for an object.

Page padding: A method of allocating virtual memory for an object in units of a virtual page, possibly with internal fragmentation in the last page.

Parallel code segment: A code segment which is executed in parallel by several processors.

Parallelizing compiler: A compiler that automatically extracts the parallelism available in a given application's program and generates code that can run concurrently on different processors.

Placement: See **data placement.**

Placement information: Information that assists data placement schemes.

Reference pattern: See **data reference pattern.**

Release consistency: A consistency model that makes the several physical copies of the same data consistent with one another only when a release operation on a synchronization variable is executed.

Release operation: A hardware instruction that is used to implement the release consistency model.

Remap operation: An operation that changes the layout of data in the shared virtual memory e.g., combine and split operations.

Replication: Creation of multiple, co-existing physical copies of the same data.

Sequential code segment: A code segment executed by a single processor.

Sequential consistency: The strictest consistency model which requires the execution of the parallel application to appear as some interleaving of the execution of the parallel processes on a sequential machine. For example, when sequential consistency and the write-update scheme is provided, a write to any data element is not complete unless all physical copies of the data element have been updated.

Shared virtual memory (SVM): A programming model in which the processors executing a parallel application communicate and synchronize by means of data that is mapped to a region of the virtual memory shared by all processors.

Split operation: An operation that splits a single object into several smaller objects as per specifications.

Sub-page: A contiguous portion within a virtual page.

Write-invalidate (WI) A scheme that maintains consistency of multiple physical copies of the same data on a write operation by invalidating all copies except the one in the local memory of the processor issuing the write request.

Write-update (WU) A scheme that maintains consistency of multiple physical copies of the same data by updating each of them on a write or a release operation.

BIBLIOGRAPHY

BIBLIOGRAPHY

[1] BBN Advanced Computers Inc., Cambridge, MA, *Inside the TC2000 Computer*, February 1990.

[2] R. Perron and C. Mundie, "The Architecture of the Alliant FX/8 Computer," in *Spring COMPCON '86*, pp. 390–393, March 1986.

[3] T. Lovett and S. Thakkar, "The Symmetry Multiprocessor System," in *Proc. Intl. Conf. of Parallel Processing*, pp. 303–310, August 1988.

[4] G. Pfister *et al.*, "The IBM Research Parallel Processor Prototype," in *Proc. Intl. Conf. on Parallel Processing*, pp. 764–772, August 1985.

[5] Intel Corporation, *A Touchstone DELTA System Description*, 1991.

[6] Kendall Square Research, *KSR1*, March 1992.

[7] D. Lenoski *et al.*, "The directory-based cache coherence protocol for the DASH multiprocessor," in *Proc. 17th Intl. Symp. on Computer Architecture*, pp. 148–159, May 1990.

[8] D. Gustavson, "The Scalable Coherent Interface and Related Standards Projects," *IEEE Micro*, pp. 10–22, February 1992.

[9] J. Kuehn and B. Smith, "The Horizon Supercomputing System: Architecture and Software," in *Supercomputing 1988*, pp. 28–34, November 1988.

[10] C. Seitz *et al.*, "The Architecture and Programming of the Ametek Series 2010 Multicomputer," in *Proc. 3rd Conf. on Hypercube Concurrent Computers and Applications Volume I*, pp. 33–36, ACM, January 1988.

[11] A. Garcia, D. Foster, and R. Freitas, "The Advanced Computing Environment Multiprocessor Workstation," Tech. Rep. RC 14491, IBM T.J. Watson Research Center, March 1989.

[12] J. Hayes, *Computer Architecture and Organization.* New York: McGraw-Hill, 1988.

[13] J. Peterson and A. Silberschatz, *Operating System Concepts.* Reading, MA: Addison-Wesley, 1985.

[14] W. Delaney, D. Cohn, K. Tracey, and M. Casey, "Data Units: A Process Interaction Paradigm," Tech. Rep. 91-3, Personal Computer Laboratory, Dept. of Computer Science and Engg., University of Notre Dame, March 1991.

[15] A. Cox and R. Fowler, "The Implementation of a Coherent Memory Abstraction on a NUMA Multiprocessor: Experiences with PLATINUM," in *Proc. 12th ACM Symp. on Operating System Principles*, pp. 32–44, December 1989.

[16] W. Bolosky, R. Fitzgerald, and M. Scott, "Simple But Effective Techniques for NUMA Memory Management," in *Proc. 12th ACM Symp. on Operating System Principles*, pp. 19–31, December 1989.

[17] R. Bisiani and M. Ravishankar, "PLUS: A Distributed Shared-Memory System," in *Proc. 17th Intl. Symp. on Computer Architecture*, pp. 115–124, May 1990.

[18] R.P. LaRowe Jr. and C. Ellis, "Experimental Comparison of Memory Management Policies for NUMA Multiprocessors," in *ACM Transactions on Computer Systems*, vol. 9, pp. 319–363, November 1991.

[19] K. Li and R. Schaefer, "A Hypercube Shared Virtual Memory System," in *Proc. Intl. Conf. on Parallel Processing*, pp. 125–131, August 1989.

[20] K. Li and P. Hudak, "Memory Coherence in Shared Virtual Memory Systems," in *ACM Transactions on Computer Systems*, pp. 321–359, November 1989.

[21] U. Ramachandran, M. Ahamad, and M. Khalil, "Coherence of distributed shared memory: Unifying synchronization and transfer of data," in *Proc. Intl. Conf. on Parallel Processing*, vol. II, pp. 160–169, August 1989.

[22] B. Fleisch and G. Popek, "Mirage: A Coherent Distributed Shared Memory Design," in *Proc. 12th ACM Symp. on Operating System Principles*, pp. 211–223, December 1989.

[23] R. Bryant, P. Carini, H. Chang, and B. Rosenburg, "Supporting Structured Shared Virtual Memory under Mach," in *Proc. USENIX Mach Symposium*, November 1991.

[24] J. Carter, J. Bennett, and W. Zwaenepoel, "Implementation and performance of Munin," in *Proc. 13th ACM Symp. on Operating System Principles*, pp. 152–164, October 1991.

[25] A.W. Wilson Jr., "Hierarchical cache/bus architecture for shared-memory multiprocessors," in *Proc. 14th Intl. Symp. on Computer Architecture*, June 1987.

[26] J. Goodman and P. Woest, "The Wisconsin Multicube: A new large-scale cache-coherent multiprocessor," in *Proc. 15th Intl. Symp. on Computer Architecture*, pp. 422–431, May 1988.

[27] D. Cheriton, H. Goosen, and P. Boyle, "Multi-level shared caching techniques for scalability in VMP-MC," in *Proc. 16th Intl. Symp. on Computer Architecture*, pp. 16–24, June 1989.

[28] K. Li, "IVY: A Shared Virtual Memory System for Parallel Computing," in *Proc. Intl. Conf. on Parallel Processing*, pp. 94–101, August 1988.

[29] J. Bennett, J. Carter, and W. Zwaenepoel, "Adaptive Software Cache Management for Distributed Shared Memory Architectures," in *Proc. 17th Intl. Symp. on Computer Architecture*, pp. 125–134, May 1990.

[30] L. Lamport, "How To Make a Multiprocessor Computer That Correctly Executes Multiprocess Programs," *IEEE Transactions on Computers*, pp. 690–691, September 1979.

[31] K. Gharachorloo *et al.*, "Memory Consistency and Event Ordering in Scalable Shared-Memory Multiprocessors," in *Proc. 17th Intl. Symp. on Computer Architecture*, pp. 15–26, May 1990.

[32] S. Adve and M. Hill, "Weak Ordering - A New Definition," in *Proc. 17th Intl. Symp. on Computer Architecture*, pp. 2–14, May 1990.

[33] M. Dubois and C. Scheurich, "Memory access dependencies in shared-memory multiprocessors," *IEEE Transactions on Software Engineering*, vol. 16, pp. 660–673, June 1990.

[34] P. Teller, R. Kenner, and M. Snir, "TLB Consistency on Highly Parallel Shared Memory Multiprocessors," in *Proc. 21st Annual Hawaii Intl. Conf. on System Sciences*, pp. 184–192, IEEE Computer Soceity, 1988.

[35] D. Black *et al.*, "Translation Lookaside Buffer Consistency: A Software Approach," in *Proc. 3rd Intl. Conf. on Architectural Support for Programming Languages and Operating Systems*, pp. 113–122, April 1989.

[36] B. Rosenburg, "Low-Synchronization Translation Lookaside Buffer Consistency in Large-Scale Shared-Memory Multiprocessors," in *Proc. 12th ACM Symp. on Operating System Principles*, pp. 137–146, December 1989.

[37] M. Holliday, "Page Table Management in Local/Remote Architectures," in *Proc. 1988 Intl. Conf. on Supercomputing*, pp. 1–8, July 1988.

[38] M. Holliday, "Reference History, Page Size, and Migration Daemons in Local/Remote Architectures," in *Proc. 3rd Intl. Conf. on Architectural Support for Programming Languages and Operating Systems*, pp. 104–112, 1989.

[39] D. Black, A. Gupta, and W. Weber, "Competitive Management of Distributed Shared Memory," in *Proc. Spring COMPCON*, 1989.

[40] C. Scheurich and M. Dubois, "Dynamic Page Migration in Multiprocessors with Distributed Global Memory," *IEEE Transactions on Computers*, pp. 1154–1163, August 1989.

[41] J. Ramanathan and L. Ni, "Critical Factors in NUMA Memory Management," Tech. Rep. MSU-CPS-ACS-34, Michigan State University, December 1990.

[42] J. Ramanathan and L. Ni, "Critical Factors in NUMA Memory Management," in *Proc. 11th Intl. Conf. on Distributed Computing Systems*, pp. 500–507, May 1991.

[43] W. Bolosky *et al.*, "NUMA Policies and their Relation to Memory Architecture," in *Proc. 4th Intl. Conf. on Architectural Support for Programming Languages and Operating Systems*, pp. 212–221, April 1991.

[44] M. Stumm and S. Zhou, "Algorithms implementing distributed shared memory," *IEEE Computer*, pp. 54–64, May 1990.

[45] B. Nitzberg and V. Lo, "Distributed Shared Memory: A Survey of Issues and Algorithms," *IEEE Computer*, pp. 52–60, August 1991.

[46] J. Bennett, J. Carter, and W. Zwaenepoel, "Munin: Distributed shared memory based on type-specific memory coherence," in *Proc. 2nd Symp. on Principles and Practice of Parallel Programming*, pp. 168–175, March 1990.

[47] J. Torrellas, M. Lam, and J. Hennessy, "Shared data placement optimizations to reduce multiprocessor cache miss rates," in *Proc. Intl. Conf. on Parallel Processing*, vol. II, pp. 266–270, August 1990.

[48] D. Cheriton, A. Gupta, P. Boyle, and H. Goosen, "The VMP Multiprocessor: Initial Experience, Refinements and Performance Evaluation," in *Proc. 15th Intl. Symp. on Computer Architecture*, pp. 410–421, May 1988.

[49] S. Eggers and R. Katz, "The Effect of Sharing on the Cache and Bus Performance of Parallel Programs," in *Proc. 3rd Intl. Conf. on Architectural Support for Programming Languages and Operating Systems*, pp. 257–270, April 1989.

[50] M. Dubois *et al.*, "Delayed Consistency and Its Effects on the Miss Rate of Parallel Programs," in *Supercomputing 1991*, pp. 197–206, November 1991.

[51] P. Keleher, A. Cox, and W. Zwaenepoel, "Lazy Release Consistency for Software Distributed Shared Memory," in *Proc. 19th Intl. Symp. on Computer Architecture*, pp. 13–21, May 1992.

[52] S. Eggers and T. Jeremiassen, "Eliminating False Sharing," in *Proc. Intl. Conf. on Parallel Processing*, vol. I, pp. 377–381, August 1991.

[53] R.P. LaRowe Jr., C. Ellis, and L. Kaplan, "The Robustness of NUMA Memory Management," in *Proc. 13th ACM Symp. on Operating System Principles*, pp. 137–151, October 1991.

[54] J. Ramanathan and L. Ni, "Compiler-Directed Object Creation for Shared Virtual Memory Systems," Tech. Rep. MSU-CPS-ACS-48, Michigan State University, October 1991.

[55] J. Ramanathan and L. Ni, "Exploiting Data Exchange Patterns in Creating Objects for NUMA Shared Virtual Memory Systems," Tech. Rep. MSU-CPS-ACS-50, Michigan State University, January 1992.

[56] J. Ramanathan and L. Ni, "Exploiting Data Exchange Patterns in Creating Objects for NUMA Shared Virtual Memory Systems," in *Proc. Intl. Conf. on Parallel Processing*, August 1992 (to appear).

[57] R. Rashid *et al.*, "Machine Independent Virtual Memory Management for Paged Uniprocessor and Multiprocesor Architectures," *IEEE Transactions on Computers*, pp. 896–908, August 1988.

[58] A. Forin *et al.*, "Design, Implementation and Performance Evaluation of a Distributed Shared Memory Server for Mach," Tech. Rep. CMU-CS-88-165, Carnegie Mellon University, August 1988.

[59] R.P. LaRowe Jr. and C. Ellis, "Experimental Comparison of Memory Management Policies for NUMA Multiprocessors," Tech. Rep. CS-1990-10, Duke University, April 1990.

[60] BBN Laboratories, Cambridge, MA, *Overview of THE BUTTERFLY GP1000*, November 1988.

[61] R. Bisiani, A. Nowatzyk, and M. Ravishankar, "Coherent Shared Memory on a Message Passing Machine," Tech. Rep. CMU-CS-88-204, Carnegie Mellon University, December 1988.

[62] H. Mizrahi *et al.*, "Extending the Memory Hierarchy into Multiprocesor Interconnection Networks: A Performance Analysis," in *Proc. Intl. Conf. on Parallel Processing*, pp. I41–I50, August 1989.

[63] P. Clancey and J. Francioni, "Distribution of Pages in a Distributed Virtual Memory," in *Proc. Intl. Conf. on Parallel Processing*, vol. II, pp. 258–265, 1990.

[64] S. Leffler *et al.*, *The Design And Implementation of the 4.3BSD UNIX Operating System.* New York: Addison-Wesley, 1988.

[65] C. Stunkel and W. Fuchs, "TRAPEDS: Producing Traces for Multicomputers Via Execution Driven Simulation," in *Performance Evaluation Review*, vol. 17, pp. 70–78, ACM, 1989.

[66] S. Eggers *et al.*, "Techniques for efficient inline tracing on a shared-memory multiprocessor," in *Proc. Intl. Conf. on Measurement and Modeling of Computer Systems*, pp. 37–46, May 1990.

[67] H.-B. Lim, "Characterization of Parallel Program Behavior on a Shared-Memory Multiprocessor," M.S. dissertation, CSRD Technical Report No. 1138, Department of Electrical and Computer Engineering, Univ. of Illinois at Urbana-Champaign, August 1991.

[68] D. Reed, L. Adams, and M. Patrick, "Stencils and Problem Partitionings: Their Influence on the Performance of Multiple Processor Systems," *IEEE Transactions on Computers*, vol. C-36, pp. 845 – 858, July 1987.

[69] R.P. LaRowe Jr., J. Wilkes, and C. Ellis, "Exploiting Operating System Support for Dynamic Page Placement on a NUMA Shared Memory Multiprocessor," in *Proc. 3rd ACM SIGPLAN Symp. on Principles and Practice of Parallel Programming*, pp. 122–132, March 1991.

[70] IBM Corporation, Austin, Texas, *IBM RISC System/6000 Technology*, 1990.

[71] F. Allen *et al.*, "An Overview of the PTRAN Analysis System for Multiprocessing," *Journal of Parallel and Distributed Computing*, vol. 5, pp. 617–640, 1988.

[72] H. Zima, H. Bast, and H. Gerndt, "SUPERB: A tool for semi-automatic MIMD-SIMD parallelization," *Parallel Computing*, vol. 6, pp. 1–18, 1988.

[73] M. Wolfe, *Optimizing Supercompilers for Supercomputers.* Cambridge, Massachusetts: The MIT Press, 1989.

[74] H. Zima and B. Chapman, *Supercompilers for Parallel and Vector Computers.* New York: Addison-Wesley, 1990.

[75] M. Wolfe and M. Lam, "A Loop Transformation Theory and an Algorithm to Maximize Parallelism," *IEEE Transactions on Parallel and Distributed Systems*, vol. 2, pp. 452–471, October 1991.

[76] Z. Fang, P. Yew, P. Tang, and C. Zhu, "Dynamic Processor Self-Scheduling for General Parallel Nested Loops," in *Proc. Intl. Conf. on Parallel Processing*, pp. 1–10, 1987.

[77] C. Polychronopoulos and D. Kuck, "Guided Self-Scheduling: A Practical Scheduling Scheme for Parallel Ccomputers," *IEEE Transactions on Computers*, pp. 1425–1439, December 1987.

[78] T. Tzen and L. Ni, "Dynamic Loop Scheduling for Shared-Memory Multiprocessors," in *Proc. Intl. Conf. on Parallel Processing*, vol. 2, pp. 247–250, 1991.

[79] S. Hummel, E. Schonberg, and L. Flynn, "Factoring: A Practical and Robust Method for Scheduling Parallel Loops," in *Supercomputing 1991*, pp. 610–619, November 1991.

[80] S. Abraham and D. Hudak, "Compile-Time Partitioning of Iterative Parallel Loops to Reduce Cache Coherency Traffic," *IEEE Transactions on Parallel and Distributed Systems*, vol. 2, pp. 318–328, July 1991.

[81] J. Ramanujam and P. Sadayappan, "Compile-Time Techniques for Data Distribution on Distributed Memory Machines," *IEEE Transactions on Parallel and Distributed Systems*, vol. 2, pp. 472–482, October 1991.

[82] M. Gupta and P. Banerjee, "Demonstration of Automatic Data Partitioning Techniques for Parallelizing Compilers on Multicomputers," *IEEE Transactions on Parallel and Distributed Systems*, April 1992.

[83] G. Fox *et al.*, "Fortran D Language Specification," Tech. Rep. Rice COMP TR90-141, Rice University, December 1990.

[84] D. Vrsalovic *et al.*, "The Influence of Parallel Decomposition Strategies on the Performance of Multiprocessor Systems," in *Proc. 12th Intl. Symp. on Computer Architecture*, pp. 396–405, 1985.

[85] P. Havlak and K. Kennedy, "An Implementation of Interprocedural Bounded Regular Section Analysis," *IEEE Transactions on Parallel and Distributed Systems*, vol. 2, pp. 350–360, July 1991.

[86] T. Jeremiassen and S. Eggers, "Computing Per-Process Summary Side-Effect Information," Tech. Rep. 91-11-03, Dept. of Comp. Sci. & Engg., University of Washington, 1991.

[87] P. Chang *et al.*, "IMPACT: An Architectural Framework for Multiple-Instruction-Issue Processors," in *Proc. 18th Intl. Symp. on Computer Architecture*, pp. 266–275, 1991.

[88] D. Hudak and S. Abraham, "Beyond Loop Partitioning: Data Assignment and Overlap to Reduce Communication Overhead," in *Proc. 1991 Intl. Conf. on Supercomputing*, pp. 172–182, June 1991.

[89] D. Hudak and S. Abraham, "Compile-Time Optimization of Near-Neighbor Communication for Scalable Shared-Memory Multiprocessors," *Journal of Parallel and Distributed Computing*, August 1992 (to appear).

[90] S. Bokhari, "On the Mapping Problem," *IEEE Transactions on Computers*, pp. 207–214, March 1981.

[91] M. Garey and D. Johnson, *Computers and Intractability: A Guide to the Theory of NP-Completeness.* San Francisco: Freeman, 1979.

[92] P. V. Laarhoven and E. Aarts, *Simulated Annealing: Theory and Applications.* Eindhoven, The Netherlands: Philips Research Lab, 1986.

[93] J. Ramanathan and L. Ni, "The Mapping Problem: Revisited," Tech. Rep. MSU-CPS-ACS-10, Michigan State University, December 1988.

[94] J. Ramanathan and L. Ni, "The Mapping Problem - In the context of new communication paradigms in multicomputers," in *Proc. Fourth Conf. on Hypercube Concurrent Computers and Applications*, March 1989.

[95] L. Brochard and A. Freau, "Designing Algorithms on Hierarchical Memory Multiprocessors," in *Proc. 1990 Intl. Conf. on Supercomputing*, pp. 414–427, June 1990.

[96] D. Callahan and K. Kennedy, "Compiling programs for distributed-memory multiprocessors," *Journal of Supercomputing*, pp. 2:151–169, October 1988.

[97] A. Rogers and K. Pingali, "Process decomposition through locality of reference," in *Proc. ACM SIGPLAN '89 Conf. on Programming Language Design and Implementation*, pp. 69–80, June 1989.

[98] R. Ruhl and M. Annaratone, "Parallelization of FORTRAN code on Distributed-memory Parallel Processors," in *Proc. 1990 Intl. Conf. on Supercomputing*, pp. 342–353, June 1990.

[99] C. Koelbel, P. Mehrotra, and J. Rosendale, "Supporting shared data structures on distributed memory architectures," in *Proc. 2nd ACM SIGPLAN Symp. on Principles and Practice of Parallel Programming*, pp. 177–186, March 1990.

[100] V. Balasundaram *et al.*, "A Static Performance Estimator to Guide Data Partitioning Decisions," in *Proc. 3rd ACM SIGPLAN Symp. on Principles and Practice of Parallel Programming*, pp. 213–223, March 1991.

[101] V. Balasundaram, "Translating Control Parallelism to Data Parallelism," in *Proc. Fifth SIAM Conference on Parallel Processing for Scientific Computing*, March 1991.

[102] W. Abu-Sufah, D. Kuck, and D. Lawrie, "On the Performance Enhancement of Paging Systems through Program Analysis and Program Transformation," *IEEE Transactions on Computers*, pp. 341–356, May 1981.

[103] D. Gannon, W. Jalby, and K. Gallivan, "Strategies for Cache and Local Memory Management by Global Program Transformations," *Journal of Parallel and Distributed Computing*, vol. 5, pp. 587–616, October 1988.

[104] K. Gallivan, W. Jalby, and D. Gannon, "On the Problem of Optimizing Data Transfers for Complex Memory Systems," in *Proc. 1988 Intl. Conf. on Supercomputing*, pp. 238–253, July 1988.

[105] M. Wolfe, "More Iteration Space Tiling," in *Supercomputing 1989*, pp. 655–664, November 1989.

[106] M. Wolfe and M. Lam, "A Data Locality Optimizing Algorithm," in *Proc. 3rd ACM SIGPLAN Conf. on Programming Language Design and Implementation*, pp. 30–44, June 1991.

[107] E. Gornish, E. Granston, and A. Veidenbaum, "Compiler-Directed Data Prefetching in Multiprocessors with Memory Hierarchies," in *Proc. 1990 Intl. Conf. on Supercomputing*, pp. 354–368, June 1990.

[108] A. Veidenbaum, "A Compiler-Assisted Cache Coherence Solution for Multiprocessors," in *Proc. Intl. Conf. on Parallel Processing*, pp. 1029–1036, August 1986.

[109] H. Cheong and A. Veidenbaum, "A Cache Coherence Scheme with Fast Selective Invalidation," in *Proc. 15th Intl. Symp. on Computer Architecture*, pp. 299–306, June 1988.

[110] H. Cheong and A. Veidenbaum, "Stale Data Detection and Coherence Enforcement using Flow Analysis," in *Proc. Intl. Conf. on Parallel Processing*, vol. I, pp. 138–145, August 1988.

[111] H. Cheong and A. Veidenbaum, "A Version Control Approach to Cache Coherence," in *Proc. 1989 Intl. Conf. on Supercomputing*, pp. 322–330, June 1989.

[112] R. Cytron, S. Karlovsky, and K. McAuliffe, "Automatic Management of Programmable Caches," in *Proc. Intl. Conf. on Parallel Processing*, vol. 2, pp. 229–238, August 1988.

[113] M. Lam, E. Rothberg, and M. Wolf, "The cache performance and optimizations of blocked algorithms," in *Proc. 4th Intl. Conf. on Architectural Support for Programming Languages and Operating Systems*, pp. 63–74, April 1991.

[114] M. Wolfe, "Multiprocessor Synchronization for Concurrent Loops," *IEEE Software*, pp. 34–42, January 1988.

[115] MIPS Computer Systems, Inc., Sunnyvale, California, *MIPS R4000 Microprocessor User's Manual*, 1991.

[116] A. Aho and J. Ullman, *Principles of Compiler Design.* New York: Addison-Wesley, 1985.

[117] R. Cytron, "Doacross: Beyond Vectorization for Multiprocessors," in *Proc. Intl. Conf. on Parallel Processing*, pp. 836–844, August 1986.

[118] S. Midkiff and D. Padua, "Compiler-Generated Synchronization for Do Loops," in *Proc. Intl. Conf. on Parallel Processing*, pp. 544–551, August 1986.

[119] B. Schunk, "Image Flow Segmentation and Estimation by Constraint Line Clustering," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 11(10), pp. 1010–1027, October 1989.

[120] D. Rover, V. Tsai, Y. Chow, and J. Gustafson, "Signal-Processing Algorithms on Parallel Architectures: A Performance Update," *Journal of Parallel and Distributed Computing*, vol. 13, pp. 237–245, 1991.

[121] Y. Chen and A. Veidenbaum, "Comparison and Analysis of Software and Directory Coherence Schemes," in *Supercomputing 1991*, pp. 818–829, November 1991.