2001

Asynchronous Validity Resolution in Sequentially Consistent Shared Virtual Memory

Jonathan Thomas

ASYNCHRONOUS VALIDITY RESOLUTION IN SEQUENTIALLY CONSISTENT SHARED VIRTUAL MEMORY

BY

Jonathan Thomas

B.S. University of Vermont, 1995

M.S. University of Maine, 1998

A THESIS

Submitted in Partial Fulfillment of the

Requirements for the Degree of

Doctor of Philosophy

(in Computer Science)

The Graduate School

The University of Maine

May, 2001

Advisory Committee:

James L. Fastook, Associate Professor of Computer Science, Advisor

Larry Latour, Associate Professor of Computer Science

Thomas Wheeler, Assistant Professor of Computer Science

Thomas Wagner, Assistant Professor of Computer Science

Fei Chai, Assistant Professor of Marine Sciences

LIBRARY RIGHTS STATEMENT

In presenting this thesis in partial fulfillment of the requirements for an advanced degree at The University of Maine, I agree that the Library shall make it freely available for inspection. I further agree that permission for "fair use" copying of this thesis for scholarly purposes may be granted by the Librarian. It is understood that any copying or publication of this thesis for financial gain shall not be allowed without my written permission.

Signature:

Date:

Asynchronous Validity Resolution in Sequentially Consistent Shared Virtual Memory

By Jonathan Thomas

Thesis Advisor: Dr. James L. Fastook

An Abstract of the Thesis Presented
in Partial Fulfillment of the Requirements for the
Degree of Doctor of Philosophy
(in Computer Science)
May, 2001

Shared Virtual Memory (SVM) is an effort to provide a mechanism for a distributed system, such as a cluster, to execute shared memory parallel programs. Unfortunately, SVM has performance problems due to its underlying distributed architecture. Recent developments have increased performance of SVM by reducing communication. Unfortunately this performance gain was only possible by increasing programming complexity and by restricting the types of programs allowed to execute in the system.

Validity resolution is the process of resolving the validity of a memory object such as a page. Current SVM systems use synchronous or deferred validity resolution techniques in which user processing is blocked during the validity resolution process. This is the case even when resolving validity of false shared variables. False-sharing occurs when two or more processes access unrelated variables stored within the same shared block of memory and at least one of the processes is writing. False sharing unnecessarily reduces overall performance of SVM systems

because user processing is blocked during validity resolution although no actual data dependencies exist.

This thesis presents Asynchronous Validity Resolution (AVR), a new approach to SVM which reduces the performance losses associated with false sharing while maintaining the ease of programming found with regular shared memory parallel programming methodology. Asynchronous validity resolution allows concurrent user process execution and data validity resolution. AVR is evaluated by comparing performance of an application suite using both an AVR sequentially consistent SVM system and a traditional sequentially consistent (SC) SVM system. The results show that AVR can increase performance over traditional sequentially consistent SVM for programs which exhibit false sharing. Although AVR outperforms regular SC by as much as 26%, performance of AVR is dependent on the number of false-sharing vs. true-sharing accesses, the number of pages in the program's working set, the amount of user computation that completes per page request, and the internodal round-trip message time in the system. Overall, the results show that AVR could be an important member of the arsenal of tools available to parallel programmers.

ACKNOWLEDGEMENTS

I would like to thank my wife, Kris, for her support of me and my Ph.D efforts. Although I am grateful for her understanding, patience, and tolerance for the many late nights that I worked, I am especially appreciative of her continuous support of my educational goals. I thank her for helping me finish my Bachelor's and for supporting me in the earning of both my Master's and Ph.D. degrees.

I would like to thank my parents, Tom and Judy Thomas, for support of my educational endeavors. Their support, among other things, allowed me to earn a Bachelor's, an educational foothold for my Master's and ultimately my Ph.D.

I would like to thank my advisor, James Fastook, for allowing me to pursue this problem, for supporting my work, and for offering invaluable insights when the problem seemed most difficult. I would like to thank my committee members, Larry Latour, Thomas Wheeler, Thomas Wagner, and Fei Chai for their guidance, willingness to help, and commitment. I thank Larry Latour and Tom Wheeler for their suggestions on transactions and software engineering. I am grateful to Tom Wagner for his help with my proposal and for helping me through the process. I thank Fei Chai for allowing me to be part of the University of Maine Beowulf Project. Without access to the Beowulf machine, this work may not have been possible.

I would like to thank the Computer Science Department for support of my graduate work and for starting the Ph.D. program. Particularly, I would like to thank George Markowsky for 17 years of relentless effort toward the evolution of the department and development of the new program. As the first Computer

Science Ph.D. student at The University of Maine, I am grateful for the efforts of the department in helping me navigate my graduate career.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

Chapter 1

INTRODUCTION

Cluster based computing has become increasingly popular due to the availability of low cost off-the-shelf computer components. Advances in technology have allowed clusters composed of off-the-shelf hardware to perform similarly to expensive modern shared-memory supercomputers. Hence, clusters are attractive platforms for production and research. The problem is that clusters have a distributed-memory architecture and distributed-memory programming, i.e. message passing, is more difficult than shared-memory programming. In the distributed-memory programming paradigm, the programmer must manage data flow across processors by using explicit messaging constructs. In contrast, the shared-memory programming paradigm allows common areas of memory to be accessed by multiple processors, such that shared information can be passed by reference. Shared Virtual Memory (SVM) is an effort to provide a shared memory programming paradigm over a distributed memory architecture.

A memory consistency model of a shared-memory multiprocessor provides a formal specification of how the memory system will appear to the programmer, eliminating the gap between the behavior expected by the programmer and the actual behavior of the system [2]. Caching is a mechanism which serves to provide locality to a processing unit. A cache consistency model, sometimes called a cache coherence model, is a specification which ensures that the view of memory maintained across all caches allows the correct execution of programs. Shared Virtual Memory emulates shared memory cache consistency in a distributed memory system. A SVM employs a memory consistency model, such as Sequential Consistency (SC) [32], to ensure the proper execution of programs in the distributed system. Unfortunately, consistency in a distributed system has considerable performance costs. These costs are related to the number of messages required to synchronize distributed memories [12]. Consistency costs are a barrier to SVM performance since consistency maintenance occurs inline in relation to user processing. This means that a process must block while permission to a memory object is requested and received. The database community, when referring to transaction processing, classifies this process as *synchronous validity checking* if the permission request occurs inline before transaction processing. If the permission request is performed inline after transaction processing, it is called *deferred validity checking.* Synchronous validity checking is a pessimistic approach which has been observed to have a high cost of consistency [12]. Deferred validity checking works well with transactions, but has problems when applied to SVM. It requires extra effort on behalf of the programmer for correct program execution in SVM. Deferred validity checking in SVM requires global synchronization of mem-

ory accesses which introduces a performance bottleneck and limits the amount of parallelism in a program's execution.

This thesis proposes the use of asynchronous *validity resolution* (AVR) in sequentially consistent SVM. Asynchronous validity resolution is a semi-optimistic protocol, analogous to *asynchronous validity checking*, in which computation overlaps the communication request for memory access permission. Asynchronous validity resolution is of interest because it has the capability of reducing the effect of false-sharing. False-sharing occurs when two or more processors access variables located on the same block of memory and at least one of the processors is writing. Typically, the false-shared block of memory is an operating system level page structure. Asynchronous validity resolution allows a process to continue with computation using potentially invalid data while permission to access the memory location is sent and received. When access permission is received and the SVM determines that the data is false-shared, computation can continue. If the SVM determines that the data is not false-shared, the requesting process must rollback to the state at the time of the invalid memory access. Implementation of AVR involves time and storage space overhead. Process checkpointing, memory page comparisons, and rollbacks are required to be performed. Although this approach has substantial overhead, it improves the performance of sequential consistency SVM for some applications by reducing the negative performance effects that are inherent to the SVM design. Specifically, AVR increases performance for applications which have high ratio of false-shared to true-shared memory accesses and applications which perform a large amount of user computation relative to the time required for memory resolution.

1.1 Shared Virtual Memory

Message passing and shared-memory are two parallel programming models that provide mechanisms for data to be shared among processes. The message passing model is the most versatile, because it can be used with both shared-memory multiprocessors and distributed systems. The caveat is that the message passing model is difficult to program, since the communication layer is visible to the programmer. The shared-memory paradigm, an extension of uniprocessor program-

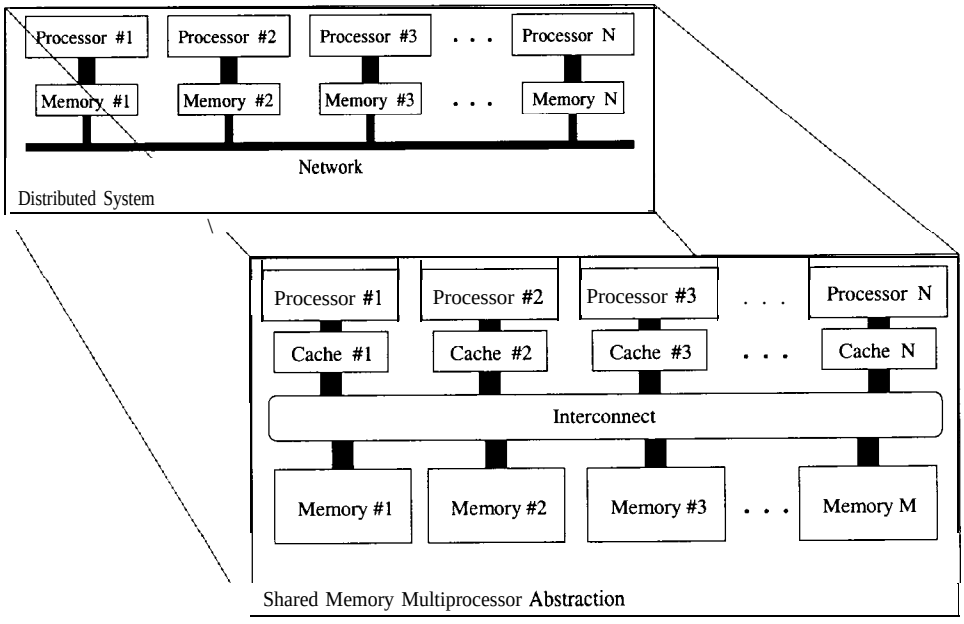

Figure 1.1: Shared Memory Abstraction

ming methodology, provides an intuitive and easy programming model. Unfortunately, the shared-memory model does not extend readily to distributed systems. Shared Virtual Memory is an effort to provide a shared memory programming model with message passing performance characteristics in a distributed environment .

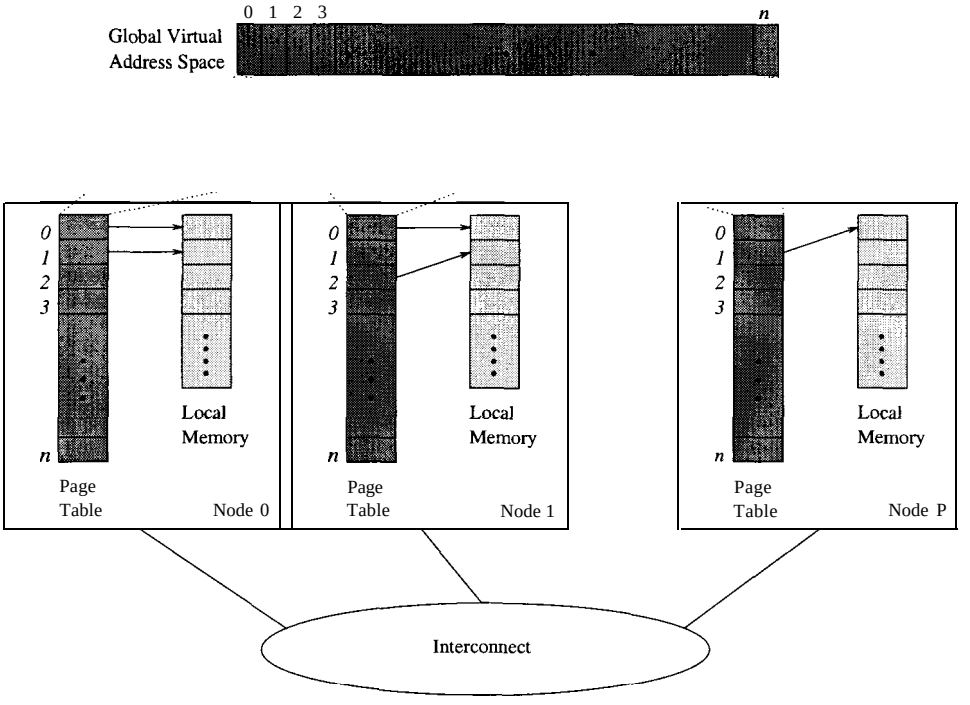

Figure 1.2: Shared Virtual Memory[20]

Shared Virtual Memory was first proposed in 1986 by Kai Li [35]. Shared Virtual Memory provides an abstraction of shared memory architecture (Figure 1.1) and emulates caching in cache-coherent multiprocessors [34]. In the cache-coherent multiprocessor architecture, caches provide local access to a global address space. In SVM, local memory acts as a cache of a global virtual address space. See Figure 1.2. Memory consistency is maintained by using consistency mechanisms, similar to those used in cache-coherent multiprocessors. Each node has a page table with an entry for each page of global virtual address space. Entries can either be valid or invalid. A valid entry signifies that the global page is cached in local memory and maps the global address to the address of the local cached page. An invalid entry signifies that the page has never been cached or

that the cached page is invalid. A page fault occurs when attempting to access an invalid page. The SVM system is responsible for memory consistency and must respond to page faults by fetching new copies of pages from remote sources.

The following sections outline some fundamental concepts of shared virtual memory such as caching and granularity. The later sections, starting with Section 1.4, provide an overview of the major advancements in the SVM field describing research efforts centered on memory consistency models, consistency protocols, architectural support mechanisms, and application support methodologies.

1.2 Cache Consistency

Optimal multiprocessing in a shared memory system requires some mechanism, such as caching, to provide locality. Caching allows multiple processing units to concurrently access replicated shared memory objects. The cache consistency mechanism controls concurrent access to the shared memory objects such that the allowed order of memory operations provides for the correct execution of programs. The database community has dissected cache consistency into four main categories[12]. They are *invalid access prevention*, *write validity checking*, *write permission duration*, and *remote update action*.

Invalid access prevention denotes how the consistency algorithm maintains the caches. In detection based algorithms, the cache is validated before performing an operation. In avoidance based algorithms, the cache is maintained in an up-to-date state. Avoidance based algorithms tend to perform better in small systems. Their performance degrades as system size and number of messages increases. Clearly,

the use of a page table in an SVM is a detection based mechanism, since the entry in the page table is tested for validity. Pre-fetching is a type of avoidance, since it acts to cache pages before they are required for use.

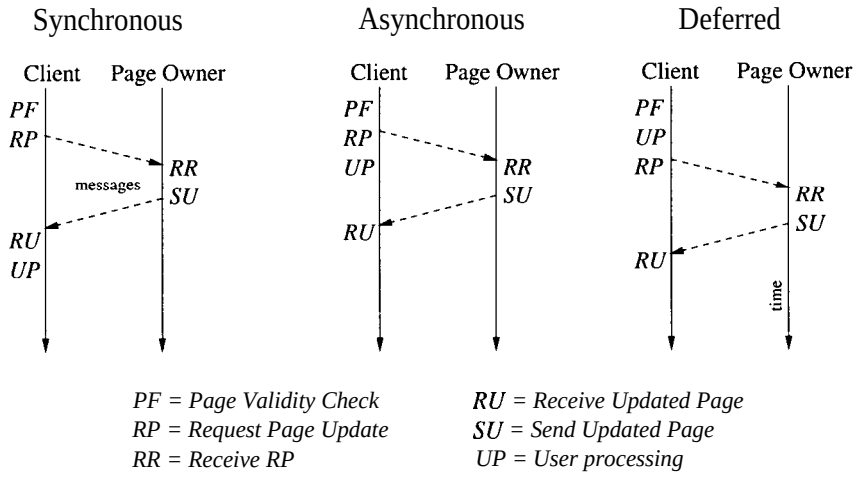

Figure 1.3: Validity Checking

Write validity checking is the process of obtaining write permission from the concurrency control, the server of the memory object. *Write validity checking*, as shown in Figure 1.3, has been subdivided into three categories called *synchronous, asynchronous,* and deferred. *Synchronous Validity Checking* means that the process obtains write permission prior to transaction processing. Deferred *Validity Checking* is the opposite of synchronous write declaration with write declaration occurring at the point when the transaction commits. *Asynchronous Validity Checking* is the process when write validity checking occurs concurrently with transaction processing. Synchronous write validity checking is referred to as pessimistic, while deferred validity checking is referred to as optimistic. The main disadvantage to the pessimistic approach is a higher cost of consistency than that of the optimistic approach [12]. The disadvantage of the optimistic approach is the

late detection of data conflicts which cause a transaction abort. In the database domain, the choice between synchronous and deferred methods is a trade-off in the number of messages sent and the transaction abort rate [12]. Asynchronous validity checking seeks to minimize the cost of consistency and to discover conflicts earlier in the transaction execution. Although programs usually are not described in terms of transactions, nearly all SVM systems use a process similar to synchronous validity checking where write permission is received before the write occurs. The system which this dissertation proposes is asynchronous validity resolution (AVR). AVR is different from asynchronous validity checking because AVR allows a program to use data that is marked as invalid while data validity resolution occurs. Additionally, AVR does not require a program to be broken into distinct transactions.

Write duration is the length of time for which write permission is given to a cached object. Write duration can be inter-transactional or *intratransactional.* Intertransactional write duration means that write permissions, unless revoked by the server, do not have to be reacquired for a series of transactions which use the same cached objects. Intratransactional write duration means that the cached object is only valid for the lifetime of the caching transaction. Write duration in SVM is defined by the pattern of sharing. Write permission usually only extends to the lifetime of the process with some type of writeback occurring at the end of execution.

Remote update action refers to how the updates are handled in the system. Propagation of the update means that the update is installed at each remote site. Invalidation means that old remote copies are marked as invalid. As the number

of nodes in the system increases, the messages required for the propagation of updates degrades overall system performance considerably. In general, invalidation tends to perform much better than propagation, except in a fixed single-writer multiple-reader configuration [9]. Invalidation is typically used in SVM systems because the pattern of memory access in unknown and the cost of communication within the system is high. Invalidation makes page acquisition demand-based and minimizes unnecessary overhead for the writing process.

In distributed client-server transaction processing, avoidance based algorithms using deferred write declaration, intertransactional caching, and a dynamic update action perform better than other algorithms using different combinations of features. Although SVMs present a different environment than database transaction processing, SVMs may benefit from application of some of these cache consistency techniques.

1.3 Granularity and Fragmentation

Granularity refers to the size of the shared memory blocks used in the system. Common granularities are byte, word, page or complex data type (object). Granularity size has many trade-offs and is an important issue, since it effects the overall performance of the system. Applications have inherent memory sharing patterns and memory access granularity determined by algorithm design. Use of an SVM with a mismatched granularity causes the program to have an induced memory sharing pattern [22]. Small granularity creates additional directory overhead and increased message traffic, but reduces the potential for false-sharing. False-sharing

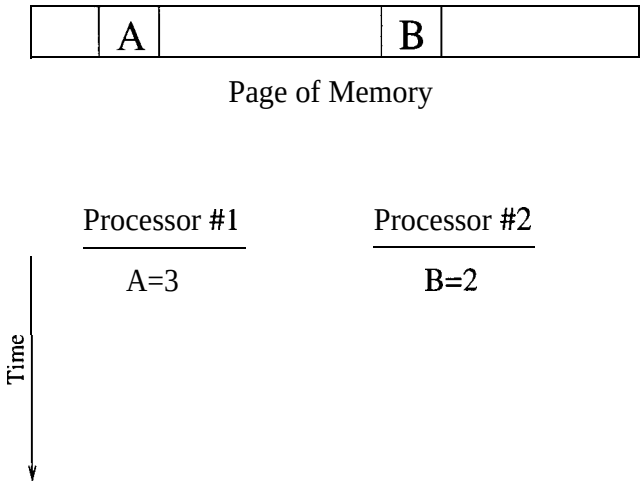

Figure 1.4: False-Sharing

occurs when two or more processes use disjoint sets of variables that are stored in the same block of memory and at least one of the processors is writing (see Figure 1.4). False-sharing is a cause of the ping-pong effect in which a page of memory is repetitively sent from one processor to another with only a minimal amount of work actually being done. False-sharing, in synchronous validity resolution, results in a process having to block on a page fault until an updated page is obtained. This burdens the system with unnecessary performance loss, since parallel processing cannot occur even though no actual sharing is taking place. Fragmentation occurs when a page is fetched in response to a fault on a single word of memory. Fragmentation breaks a processing unit's locality of reference. Locality of reference is the affinity for some working set of data objects. In general, large granularity has reduced directory overhead and provides locality of reference for a processing unit, but introduces a greater potential for false sharing.

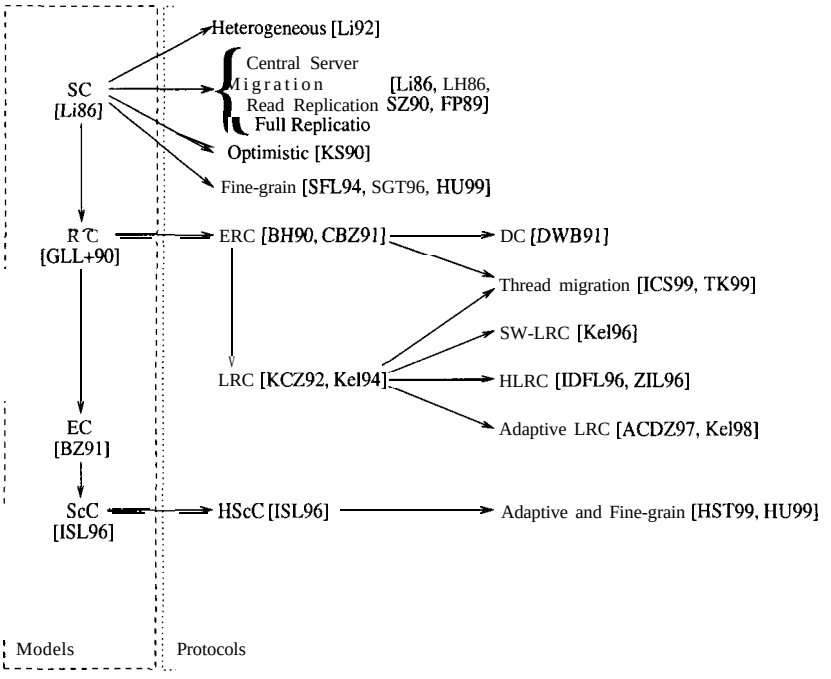

Figure 1.5: Consistency Models and Protocols (adapted from [20])

In practice, the choice of sharing size usually reflects what is implemented on the platform, typically a page of memory. Small granularities, such as words or cache lines, are rarely used due to high overhead except in systems with some hardware support. Many efforts in the SVM community have been aimed at reducing the negative effects of false sharing. These efforts will be discussed in later sections.

1.4 Consistency Models

The consistency model is the underlying specification of how memory operations are handled in the system. It guarantees a consistent view of memory that matches programmer expectations. Figure 1.5 shows an abridged historical view of SVM

research involving memory consistency models and protocols. Some of these models and protocols will be discussed in later sections.

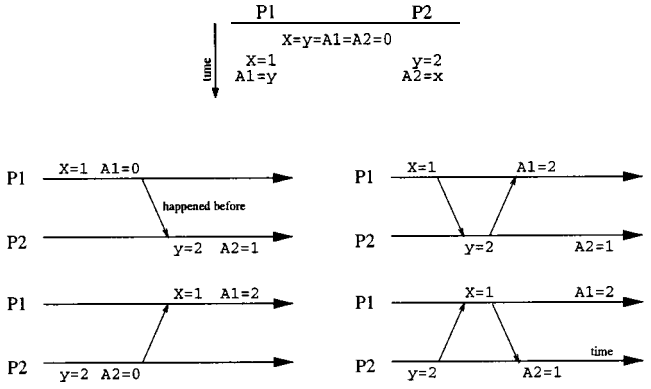

Figure 1.6: Sequential Consistency

1.4.1 Sequential Consistency

Strict consistency is a uni-processor consistency model which is the basis for sequential consistency, a multiprocessor consistency model. Strict consistency defines what most programmers intuitively expect: A read operation returns the value of the most recent write operation. Sequential Consistency (SC) is strict consistency applied to multi-programming. In SC, the result of any execution appears as some interleaving of the memory operations of the individual nodes when executed on a multi-threaded sequential machine. Figure 1.6 shows the possible outcomes in sequential consistency given a set of memory operations.

Sequential Consistency [32] was the first model used for SVM implementation [35]. SC is an extension of strict consistency that offers a simple parallel programming model which follows common uni-processor methodology. SC has barriers to performance, since every write can result in an invalidation and every read can

result in a page fault. This means that the protocol overhead can be high relative to the computation.

1.4.2 Processor Consistency

In processor consistency, writes issued by an individual processor are seen in order by all processors. Writes by two different processors, however, can be seen differently. Processor consistency is weaker than SC because different processors can observe a different ordering of writes.

1.4.3 Weak Consistency

Weak Consistency (WC) methods allow caches to become inconsistent and utilize synchronization operations to define synchronization points within programs. Weak consistency was introduced by Adve and Hill [3] in 1990. Weak Consistency methods aggregate consistency events and perform coherence operations at specific user defined synchronization points within the program. The overall effect is that network traffic can be minimized in WC. Weak Consistency relies on the programmer to properly label programs with synchronization operations. These synchronization operations are fence operations that act to separate conflicting sets of memory operations. Synchronization operations are guaranteed to be sequentially consistent. Thus, by properly labeling a program, sequentially consistent execution can be enforced.

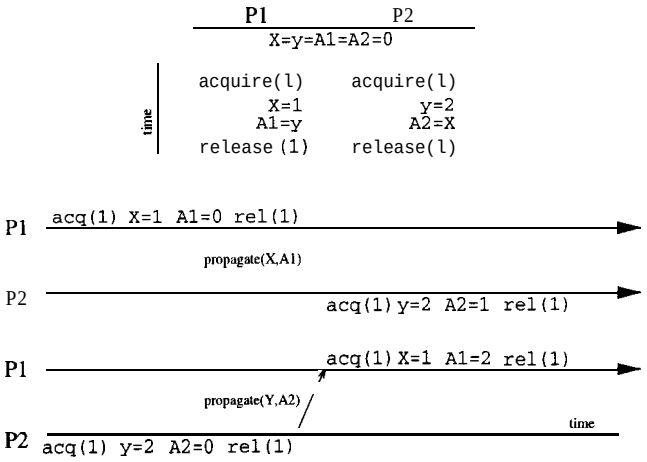

Figure 1.7: Release Consistency

1.4.4 Release Consistency

Release Consistency (RC) [14] is an effort to lower the cost of coherence. Release consistency is an extension of weak consistency which utilizes two types of synchronization operators: Acquire and *Release.* An Acquire operation is performed when a process wants access to a shared memory object. A Release operation is performed when a process "gives up" a shared memory object. These operations have two roles. The first is that of a barrier where synchronization of all shared data objects occurs. The second role is that of a lock. Acquires and Releases are guaranteed to be processor consistent. Figure 1.7 shows the two possible outcomes in a Release Consistent two processor execution. Note that variables in the two nodes are not made consistent unless synchronization operations are used.

RC reduces the frequency of coherence operations beyond that of Weak Consistency, because synchronization in RC only occurs between sharing processors. This reduces the negative effects of false sharing. The drawback is that in order

to use RC, programs must be data-race free and be properly labeled, which means that the programmer must use explicit synchronization.

RC was the basis for Eager Release Consistency (ERC) [10] and Lazy Release Consistency (LRC) [25][26] protocols. ERC and LRC are described in Section 2.4.

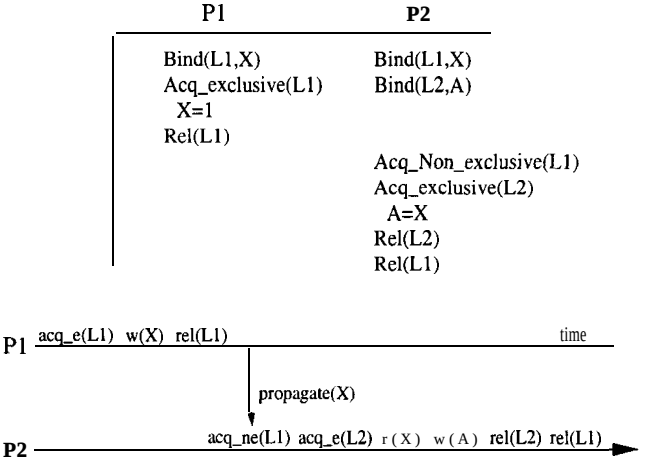

Figure 1.8: Entry Consistency

1.4.5 Entry Consistency

Entry Consistency (EC) [9] is a memory model which is more relaxed than Release Consistency. EC is more relaxed because it limits the data to which the synchronization is applied. In EC, there are two levels of synchronization. The first is identical to that of RC where global synchronization occurs. The second level provides a method for synchronization of a smaller set of data objects. In this level, the programmer explicitly binds data objects to synchronization variables. At the synchronization event, only the data which is bound to the synchronization variable is made consistent. The overall effect is a reduced message traffic which

allows for greater performance with some types of data distributions. See Figure 1.8 for an example of Entry Consistency.

A negative aspect of EC is that it places a greater burden on the programmer than RC, since the programmer must explicitly associate data to synchronization variables.

1.4.6 Scope Consistency

Scope Consistency (ScC) [23][20] is similar to entry consistency. In ScC, however, not all data objects need to be explicitly bound to synchronization variables. ScC

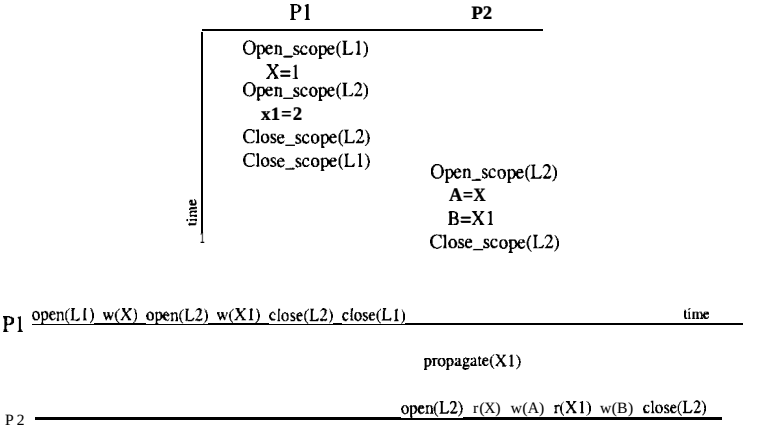

Figure 1.9: Scope Consistency

uses synchronization variables which define a scope, such that a data binding is achieved dynamically when write access to the data occurs within that scope. See Figure 1.9. The ScC model is less complex than EC to program, but is at least as complex as RC and in some cases more complex than LRC [20].

1.5 Protocols

1.5.1 Generalized SC Protocols

A protocol provides a global view of how the SVM system works. Stumm and Zhou [47] describe four general SC algorithms as central server, migration, read replication, and full replication. Although these algorithms were originally designed in the context of a sequentially consistent distributed shared memory model, they have been applied to other consistency models. Choice of an optimal algorithm for an SVM system will depend on a number of architectural and implementation features, such as amount of memory, per message communication costs, and computational ability. One negative aspect of SC protocols is the tendency for a ping-pong effect as pages or permissions migrate repeatedly between sharers. The ping-pong effect can be so severe that memory accesses are never allowed to complete [38]. A solution shown to address this problem is to assign each page a delta value, the minimum amount of time that the page must be resident [13]. The delta value is typically based on the amount of time the hardware requires to perform one memory operation. The four protocols are discussed below.

Central Server

The central server algorithm, shown in Figure 1.10, is based on client-server architecture. In this algorithm, a central node, sometimes called the home node, controls access to globally accessible memory. Caching does not occur in this algorithm. Instead messages that contain a request type and data are sent between the client and server. The server responds to a read request by sending the data

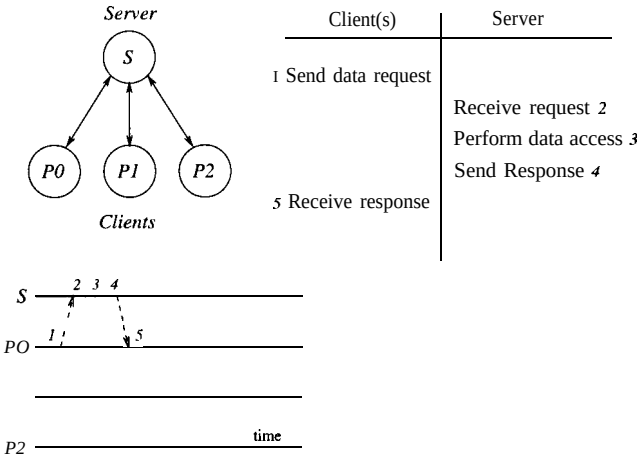

Figure 1.10: Central Server

at the requested location. A server responds to a write request by writing the data contained in the client message to the requested location. The ordering of operations can be provided by the natural sequencing produced by the interconnect or by a system of logical clocks. The central server algorithm has the obvious problem that the server performs all memory operations and can be a bottleneck to system performance.

Migration

In migration, data is shipped to the local memory of the accessing process. See Figure 1.11. Migration is a single-reader/single-writer protocol with at most one copy of each page in the system at any given time. When a process wants to access a page, it must query remote nodes for the location of the page. There are two mechanisms used to minimize the cost of the query. One is to store page directory information on a designated server. The other is to allow sharers to

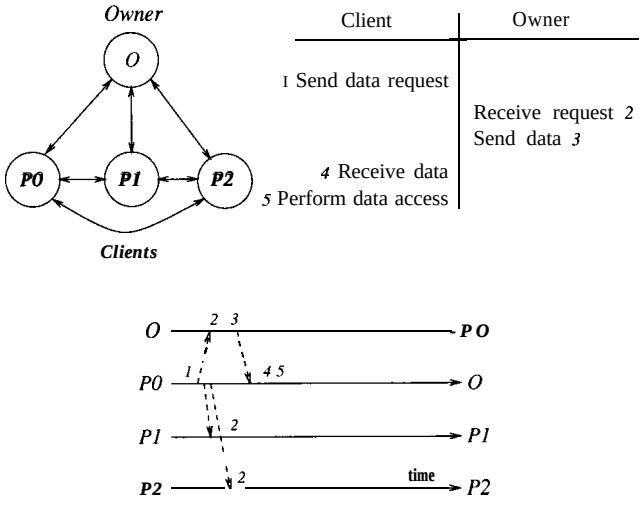

Figure 1.11: Migration

store, as hints, some information about previously seen pages, such as the page's destination address. These hints can help to lower the cost of finding the page.

Read Replication

Read replication, shown in Figure 1.12, is a multiple-readers/single-writer protocol. In read replication, concurrent read access is allowed. Shared pages of memory can be replicated with read-only permission at each process. Only a single writer is allowed in the system. In order to keep the replicated copies consistent, an owning process must keep track of the state of the replicated copies. The owner must ensure that a page with write permission is invalidated before giving read or write permission to another process. Similarly, the owner must invalidate all read-only copies before giving write permission. Systems can either scatter or migrate ownership in the system to alleviate performance limitations.

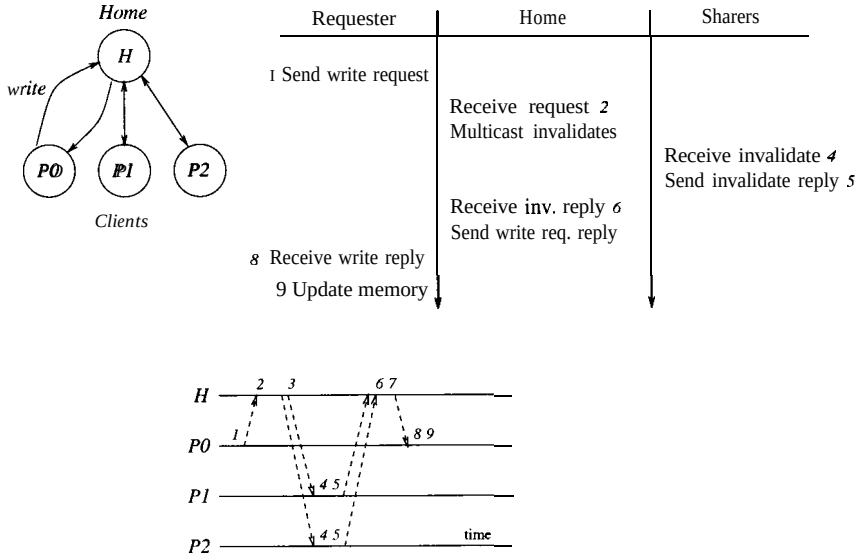

Figure 1.12: Read Replication

Full Replication

Full replication, shown in Figure 1.13, is a multiple-readers/multiple-writers algorithm. It allows concurrent reads and writes. In full replication, a mechanism must be used to ensure a global ordering of operations. This can be accomplished by using an implementation based on system of distributed clocks or by using a server implementation similar to the central server algorithm. With a system of distributed clocks each node maintains its own updates. This can be costly with respect to message traffic. Using a centralized server introduces a bottleneck, since all write operations must be scheduled by the server.

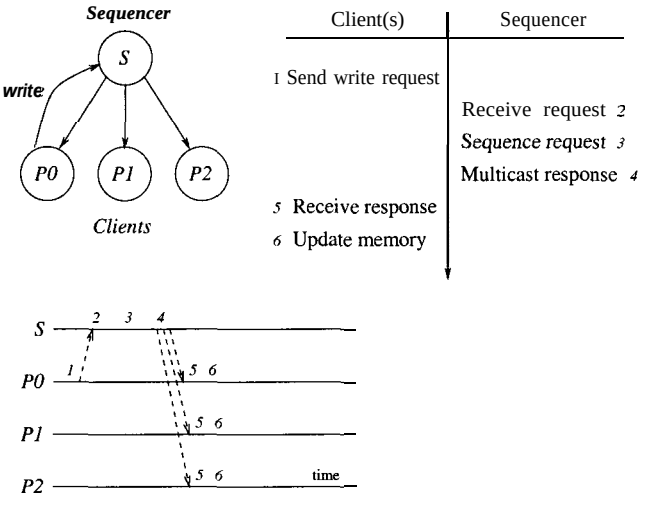

Figure 1.13: Full Replication

1.5.2 Optimistic Protocol

TORiS [31] is an optimistic central-server full replication algorithm. The primary components of TORiS are shown in Figure 1.14. TORiS defines a transaction to be a set of read and write operations bracketed by *begin-transaction* and *end- transaction* operations. Transactions are two-phase with active and commitment phases. A transaction is active after executing *begin- transaction* and before executing *end- transaction*. Memory operations of the transaction are logged during the active phase. The commitment phase begins at the point of the *end- transaction* operation. A sequence number *n* is assigned by the central server at the onset of the commitment phase. A transaction commit or abort marks the end of the commitment phase. A transaction *A* with sequence number n commits when no concurrently executing transaction with a sequence number smaller than *n* has modified any of the data that transaction *A* accessed. Transaction *A* aborts when the opposite occurs.

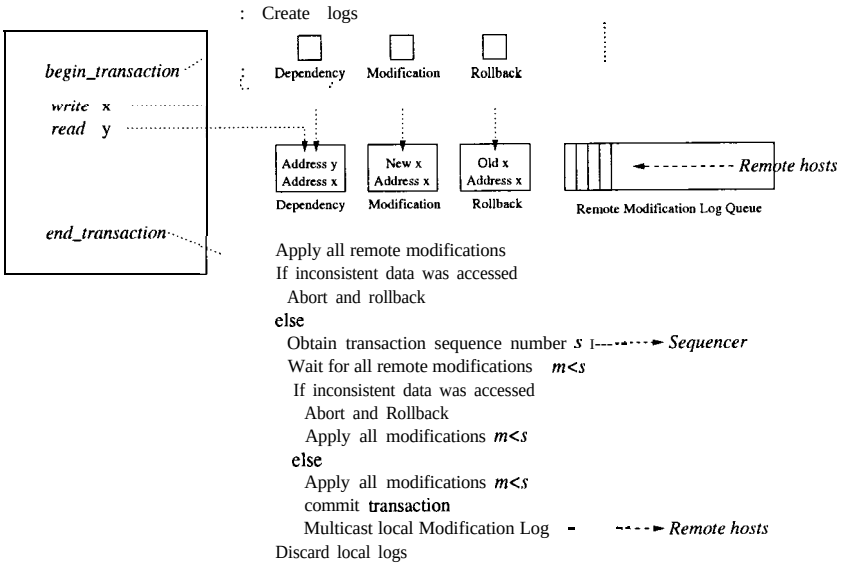

Figure 1.14: ToRiS: Optimistic Full Replication

TORiS differs from regular sequential consistency methods since it determines *a posteriori* if a transaction has accessed invalid data. Hence, TORiS uses deferred validity resolution. TORiS minimizes communication since consistency messages occur at the end of a transaction and not with each memory object access.

1.5.3 Reduced Granularity Protocols

Much work has been done in the area of fine-grain SVM. Fine-grain approaches reduce the size of the shared object in an effort to reduce the amount of false sharing and fragmentation. Blizzard-S [43] and Shasta [45] are two fine-grain SVMs which function by first scanning executables for memory operations and then modifying the executables to include coherence and synchronization functionality. This approach has the advantage that source code is not always available and false

sharing is eliminated, but has the disadvantage that there may be an increase in the number of messages required for synchronization.

Other efforts are aimed at reducing false sharing and message overhead. JIA-JIA [17] divides a shared page into blocks. A faulting processor is only required to fetch the invalid blocks and not the entire shared page. BOPS [41] adapts the shared object granularity to application requirements during program execution. MultiView [21] uses the concept of a view. A view is a region of shared memory. Views of different sizes can be used such that large and small granularity sharing is allowed in the system. Small views can be composed into larger views and large views can be decomposed into multiple small views, thus providing a sharing granularity best suited for the application.

1.5.4 Lazy Protocols

A release based protocol can be lazy in two respects. Lazy propagation means that the invalidations are propagated from one process to another on demand when the second process performs an Acquire operation. Lazy application means that invalidations are queued at the destination node and applied when the node performs its next Acquire operation. Lazy propagation minimizes coherence overhead since updates are sent to only those processes requiring the modifications. Lazy application can reduce the number of page misses which occur between the time invalidations are received and the next Acquire operation.

Eager Release Consistency (ERC) [6] [10] is a release consistent protocol which propagates updates as soon as they are made and applies updates as soon as they are received. ERC is not a lazy protocol, but was a basis for the development of

other lazy protocols. Delayed consistency [11] eagerly propagates modifications, but uses lazy application. Lazy Release Consistency (LRC) [25] is a page-based multiple writer protocol that uses lazy propagation and lazy application. It minimizes network traffic by packaging each update as a *diff*, a comparison between new modified memory and old memory.

1.5.5 Home-Based Protocols

Home-based protocols utilize one or more central nodes for bookkeeping, performing page modifications, and servicing page requests. Home-based protocols have advantages. Home-based protocols minimize the amount of information that needs to be stored, since modifications can be applied immediately and discarded. In some circumstances, home-based protocols require fewer messages since requests from remote nodes can be serviced by one round trip to the home node. Another advantage of home-based protocols is that memory operations performed at the home node do not require remote data operations. There are two main disadvantages to home based systems. One is that home-based systems typically send an entire page even if the faulting process needs to modify a single word. The second is that assignment of homes can yield a wide range of performance.

Home-based Scope Consistency [23] and Home-based LRC [20][50] are two home-based protocols.

1.5.6 Adaptive Protocols

There are two main types of adaptive protocols. The first, Adaptive Writer Protocols, adapt between single writer and multiple writer protocols. Adaptive LRC [1] is a version of LRC which adapts on a per page basis between a single writer and multiple writer protocol. At runtime during a synchronization point, the sharing pattern of each page is examined. If write-write sharing is not detected, then the protocol switches from multiple-writer to single-writer for that page. Single writer LRC [27] eliminates the need for diffs and the extra coherence messages required with regular LRC.

The second type of adaptive protocols are Adaptive Migration Protocols. Adaptive Migration Protocols adaptively migrate homes or threads. These migration methods seek to optimize the amount of local operations, minimize the amount of network traffic, and load balance the system. JIAJIA [15], incorporates home migration into home-based scope consistency. It adaptively migrates home pages according to the application sharing pattern. Orion [39], is an adaptive home-based LRC SVM system which uses home migration and dynamic adaptation between write-invalidation and write-update protocols. Orion's dynamic write protocol adaptation is based on the collection of memory access information at the home nodes and aims at lowering network costs while providing high data availability. Write invalidate methods are generally less costly since messages are shorter. Write update methods have greater network costs, but minimize page misses.

Thread migration is a strategy for load balancing a distributed system. Migration of a thread from an overburdened node to an idle or minimally burdened

node can boost overall performance. On the other hand, it may be beneficial to migrate sharing threads to the same node, such as a multiprocessor node, to take advantage of hardware and operating system memory management. This strategy also reduces the amount of network overhead between the two sharers. How to choose which threads to migrate to a location in a release consistent system is an active field of research. One approach is to minimize the total number of shared pages between any two nodes [48]. This is based on the idea that the amount of network traffic for maintenance is dependent on the number of sharers. Cohesion [33] is a thread-migratory SVM that incorporates this strategy. Additionally, in order to determine if migration of a thread to a remote node is beneficial, Cohesion determines the page-sharing pattern between the thread and all local threads and the page-sharing pattern between the thread and all threads on the remote node. If the number of local sharers is greater than the number of remote sharers, migration will not be beneficial.

1.6 Architectural Support

1.6.1 Broadcast and Multicast Protocols

Use of broadcast or multicast capable network hardware devices and libraries minimizes network traffic because they replace multiple messages with a single message. Brazos [42] is an example of a Scope Consistent SVM which uses both a multicast and point-to-point communication protocol. Multicast, as Speight and Bennett [42] describe, does lead to two problems. The first problem is useless multicast traffic. In this situation, a process receives multicast updates for pages

they are no longer actively accessing. Upon receiving the updates, the process is interrupted, thereby reducing the amount of useful computation. Brazos minimizes the effect of useless multicast traffic with a copyset reduction algorithm. This algorithm involves a counter for each page that is decremented when a useless update is received. If the counter reaches zero, the process is removed from the sender's copyset at the next synchronization point. If the process accesses the data, the counter is reset. The value of the counter is configured based on an application-specific history mechanism.

The second problem is multicast conflicts. A multicast conflict occurs when a multicast update for a page arrives while waiting for a response to a request for updates. In this situation, there is an extra conflicting copy of an update. Brazos' solution to the multicast conflict problem is to store the duplicate updates and at the next barrier transfer the updates to the sending processor. For each page, the sending processor determines which nodes received useless updates. This set of nodes becomes the page's copyset. Subsequent updates are multicast to nodes only within the copyset.

The resolution of the two multicast-based problems helps Brazos provide a dynamic write protocol on a per page basis. When only one process has membership in the copyset of a particular page, the page is handled using a write-invalidation protocol. With two or more sharers, multicast is utilized and the page is handled using a write update protocol.

1.6.2 Network Support and SMP Clusters

Network devices capable of performing memory operations are the basis for research in the area of remote operations. Coupled with high speed interconnects, these types of network interfaces can be used to improve performance of SVM systems. Digital's Memory Channel and the custom network interface of the SHRIMP [4] multicomputer are examples of memory-mapping communication devices. Automatic Update Release Consistency (AURC) [19] is a SVM system which has been used on the SHRIMP multicomputer. In AURC, the memory bus is monitored for write operations. These write operations are then automatically propagated in hardware to remote memory locations. Cashmere [29] is a similar system that uses memory channel. Cashmere does not use an automatic remote write mechanism like AURC. Instead, a remote write is performed on demand for each shared local write.

Use of Symmetric Multi-processor (SMP) nodes with SVM systems has appeared in recent work [7] [S]. These efforts are an attempt to capitalize on hardware coherence and synchronization while maintaining memory consistency across the entire cluster. The results of the work show that SMP nodes perform better than uniprocessor nodes and that increasing the size of the SMP nodes further increases performance.

1.7 Application Support

1.7.1 Compiler Transformations and Instrumentation

Compiler transformations are useful tools for providing transparency, reusabilty, and increased performance. Compiler modifications to source can reduce the amount of false sharing, the amount of synchronization, and overall performance of the system. Jeremiassen and Eggers [24] have used compile time data transformations to reduce the number of false-sharing misses in SVM systems. They have modified Parafrase-2, a source-to-source restructurer, to incorporate algorithms that reduce the amount of false sharing. Specifically, Parafrase-2 analyzes parallel source code, generates information about cross-processor memory reference patterns, identifies data structures that may be false shared at runtime, and transforms these data structures to eliminate false sharing. Two fundamental transformations are used. The first is to cluster together data that is accessed primarily by one processor. This ensures that a high degree of availability is maintained for data with processor locality. The second is to arrange data such that shared data structures with no processor locality do not fall in the same cache lines. This transformation minimizes the cost of coherency overhead for shared data.

There is substantial work in the area of synchronization-reducing compiler transformations. Barrier synchronization can be the worst form of synchronization because barriers are global in nature. Potential computation time is lost as processors already in a barrier become idle and wait for the slowest processor to enter the barrier. Also as computation is spread out across more processors,

the frequency of barriers increases, thus decreasing the amount of useful computation. Han, Tseng, and Keleher [16] researched SVM barrier synchronization elimination for compiler-parallelization. The main thrust of their work is to utilize LRC and replace barriers with nearest-neighbor synchronization translations. Communication of memory coherence information in SVM systems often occurs within a barrier region. In some barriers, communication only takes place between neighboring processors. Nearest neighbor synchronization allows a process to synchronize with only those neighbor processes with which it communicates coherence information. In nearest neighbor replacement, a process sends a notification message to each of its neighbors. After the neighbors receive the notification and after the two processes have transfered any pending coherence information, the neighbors send a message in response to the notification. Receipt of responses from all the neighbors releases the process to continue computational work. Nearest neighbor replacements have the ability to balance the system by decreasing the amount of processor idle time. Another benefit of nearest neighbor replacements is the elimination of the serial bottleneck of global barrier managers.

Compiler Assisted Software DSM (CAS-DSM) [36] is a recent effort aimed at eliminating operating system involvement in segmentation violations. The majority of SVM systems rely on hardware mechanisms to detect local memory faults. These faults, in the form of segmentation violation signals (SEGV), are handled by costly operating system or SVM system SEGV handler routines. CAS-DSM instruments source code in a way that helps avoid the segmentation violation. In essence, CAS-DSM equips the source with aggressively optimized prefetch functionality. This is a sensible approach since the overhead required for testing for

invalid data and possibly prefetching in user code is less than the overhead expense of the operating system or SVM SEGV handlers.

Another approach at the compiler level is the use of Aspect-Oriented Programming (AOP) [37]. Aspect-Oriented Programming [30] is a programming methodology where different aspects of a program, such as synchronization and memory management, are specified separately from the base program. Aspects are woven into a base program to create source that has aggregate functionality. Mentre, Metayer, and Priol's work proposes a base program consisting of a high level SVM abstraction and aspects consisting of different implementation choices. The aspects are transformed into automatons which can be dynamically loaded at runtime in order to optimize SVM performance for a particular application sharing pattern.

1.7.2 Transparency

Shasta [44] is a SVM system which tackles the problem of transparency. Shasta is a fine-grain SVM system that provides the tools to execute applications compiled for hardware shared-memory systems. These applications include a large number of commercial binaries that are available in the software market. Shasta transparently executes binaries by outfitting the binary code with appropriate memory coherence functionality. Shasta scans the binary executable for load and store operations and inserts code that checks during runtime whether the required data is local and in the proper state. Shasta minimizes the overhead of checks by recognizing that groups of memory operations can be satisfied by a single check. Memory checks eliminate operating system page fault resolution. Also,

Shasta incorporates polling as a replacement for interrupt based SVM messaging. Interrupts are costly since they involve hardware and operating system layers. Shasta researchers have noted that two of the largest problems with this type of approach are the correct support of the instruction set architecture and the provision of operating system services, other than memory-management, in the distributed system.

1.8 Thesis

This dissertation is an evaluation of a new protocol called Asynchronous Validity Resolution. It centers around three claims:

Claim 1.1 Asynchronous Validity Resolution (AVR) decreases the negative effects of false-sharing that are found in regular Sequential Consistency.

Claim 1.2 Asynchronous Validity Resolution (AVR) does not require a different programming methodology than that of regular Sequential Consistency.

Claim 1.3 Asynchronous Validity Resolution (AVR) has best performance in loosely coupled systems that have relatively high communication costs.

False-sharing burdens processes with extra induced coherence costs. Reducing false-sharing by decreasing granularity is an obvious solution. Unfortunately, smaller granularity means more shared objects, more directory overhead, and

more message traffic. Also, in order to use hardware and operating system services, many Shared Virtual Memory sharing granularities are matched to the hardware and operating system page size requirements. Thus, use of small granularity may not be optimal, if hardware and operating system support is lost.

Some relaxed consistency protocols eliminate false sharing, but not without limitations. Programs with data races, such as chaotic programs, cannot be run on relaxed memory model SVM systems. Also, programs destined for execution on relaxed memory model systems must be written to include synchronization primitives. In some protocols, explicit binding of synchronization to variables is required. In other words, the implementation details of the memory consistency model have bubbled-up from lower levels into user application space. The programming methodology required by relaxed memory models is middle-ground between the traditional sequentially consistent shared memory parallel programming paradigm and the message passing programming paradigm. Sequentially Consistent shared memory parallel programming is an intuitive step from regular uni-processor programming methodology. Thus, it makes sense to study shared virtual memory in the context of sequentially consistent shared memory programming.

The traditional shared memory parallel programming paradigm is only available on sequentially consistent SVM systems and sequential consistency protocols suffer from high communication overhead and false sharing. Kreiger and Stumm [31] proposed an approach to this problem with TORiS. TORiS uses optimistic write declaration in a multiple-writer sequentially consistent protocol, requires the programmer to define transactions within the program, and sequences memory

operations on a per transaction basis. TORiS uses transactions to define consistency events. This has two caveats. The first is that transactions larger than one memory operation disallow certain orderings of memory operations. The second is that TORiS suffers from the same programming complexity problem that is found with relaxed consistency models. Thus, TORiS doesn't offer a reduction in communication costs and false sharing while maintaining a sequentially consistent shared memory parallel programming paradigm.

SVM protocols, with the exception of TORiS, have used synchronous validity resolution techniques. Synchronous validity resolution techniques ensure that a page is valid before performing memory operations, but leave the faulting process in a blocked state while the page fault is resolved. Unfortunately if the page is false-shared, the process loses valuable computational time waiting for completion of indirectly related coherence operations. These coherence operations, such as write permission requests, can have considerable costs since they may involve a large number of sharers and a large number of messages. Current trends in hardware show that processor speeds are increasing relative to communication speeds, which suggests that the amount of computation lost due to blocking is also increasing. Overall, the amount of lost computation due to synchronous validity resolution can be substantial. Asynchronous Validity Resolution (AVR) is a new approach to the problem. AVR reduces the effects of false sharing by overlapping communication with computation. AVR is based on a read-replication sequential consistency protocol [40] and does not require explicit communication primitives, transaction boundary operations, or any other protocol programming constructs.

The base sequentially consistent protocol is a read-replication SVM protocol that is similar to an SGI Origin 2000 cache consistency protocol. As with all read-replication protocols, this protocol has a relatively high overhead on certain operations, such as a write request on a shared read-only page. In general, AVR provides the greatest benefit for write request operations on pages that are cached in read-only state on multiple remote nodes. These types of requests involve the greatest number of messages and highest overall network costs. High network costs allow more user operations to complete asynchronously. AVR also increases performance for write request operations on a pages cached in read-write state on a remote node. The success of AVR in an application is a function of the amount of false sharing in the application, the cost of SVM overhead, and the cost of coherence overhead.

1.9 Contributions

The contributions of this dissertation are the design, implementation, and evaluation of asynchronous validity resolution protocol. The claims made in Section **1.8** are validated.

Claim 1.2 states that AVR does not require specialized programming methodology. This claim is validated by designing and implementing an AVR system which uses regular shared-memory programming methodology. A programming suite is then implemented using the AVR protocol.

Claim 1.1 and Claim 1.3 address the performance characteristics of AVR. Claim 1.1 states that AVR decreases the negative effects of false sharing that

are found in regular sequential consistency while claim 1.3 states that AVR will have best performance in loosely coupled systems that have relatively high communication costs. These two claims are validated through a set of experiments and analysis of those experiments. AVR and SC are implemented in a modularized SVM, called CVM, which provides a control environment for testing protocols. A suite of applications is chosen for use as a test bed for performance analysis of AVR and SC.

The results show that four components influence the performance of AVR. They are the number of false-sharing vs. true-sharing accesses, the number of pages in the program's working set, the amount of user computation that completes per page access, and the round-trip message time. In order for AVR to outperform SC, the average round-trip message time must be greater than the average amount of time required to checkpoint and perform user computation. Additionally, the average amount of time to perform user computation must be longer than the average amount of time required to resolve the checkpoint. The amount of false-sharing and true-sharing, characteristics of program memory access patterns, also are factors of performance. True-sharing reduces performance in all SVM systems because it reduces the amount of parallelism. In AVR, however, true-sharing causes additional performance loss in the form of protocol overhead from processor rollbacks. False-sharing is the basis for performance gains in AVR, since the net effect of AVR is to allow different processors simultaneous access to false-shared pages. AVR performs well if the false-shared gains outweigh the true-shared losses. Overall, AVR met the expectations and satisfied the claims of the dissertation.

Chapter 2

ASYNCHRONOUS VALIDITY RESOLUTION

2.1 Motivation and Background

Sequential consistency (SC) was the basis for the first work in the SVM field. Sequential consistency is defined as follows:

> *A system is sequentially consistent if the global* ordering of op- *erations is such that each processor executes its operations in some sequential order and each processor views the operations of the other processors in some global sequential order as specified by the program.*

In SC, any read operation returns the value of the most recent write operation as long as it doesn't violate program order. This global ordering requirement creates

considerable consistency overhead in SVM systems. The problem is compounded by false sharing. False sharing occurs when unrelated memory accesses fall on a shared memory object, such as a page. False sharing means that consistency events are required for the memory object although there is no actual data sharing occurring.

Newer relaxed consistency models, such as Lazy Release Consistency, Entry Consistency, and Scope Consistency, have been used to provide improved performance over SC by reducing consistency overhead. Unfortunately, these newer memory models have two caveats: they cannot be used with programs that contain data races and they require more programming effort than SC.

In relaxed consistency (RC), synchronization defines consistency points. Memories are not consistent unless synchronization constructs (i.e. locks and barriers) are used. Hence, the requirement in RC that programs be data race free. Data races are often considered bugs, but are used for some programs including chaotic programs. The use of inconsistent memories in RC, allows RC protocols to reduce or eliminate false-sharing by performing page comparisons in a multiple writer configuration. In contrast to RC, SC allows programs with data races, but suffers from false sharing.

SC is an extension of the memory architecture and cache coherence mechanisms found in regular uni-processor machines. Likewise, the SC parallel programming paradigm is an extension of the programming methodology of regular uni-processor machines. SC programming is an intuitive step from traditional regular uniprocessor programming. In contrast, RC programming varies in difficulty. At a minimum, the RC programmer must use proper synchronization to

provide consistency for shared variables. This is not substantially different from SC programming, except for some programs, such as those using task queues [26]. Recent RC methods require significantly more programming effort than SC because they require variables to be bound to synchronization variables. This programming methodology approaches the complexity required by message passing programming interfaces in which the programmer must specify the communication parameters.

Asynchronous Validity Resolution (AVR) is an extension of SC designed to improve performance by overlapping computation with communication. In AVR, a memory object is used by the user program while the SVM system simultaneously verifies the memory object's validity. AVR reduces the effects of consistency overhead, especially overhead related to false-sharing. AVR utilizes the programming methodology of regular SC. It allows data races and does not require explicit coherence operations.

2.2 Protocol

Asynchronous Validity Resolution is an extension of Sequential Consistency. AVR provides overlapped communication and computation while preserving Sequential Consistency. Overlapping communication and computation means that potentially invalid data is used while data validity is resolved. Data validity resolution occurs in shared virtual memory systems when a page fault occurs. From the application viewpoint, all memory is accessible. Use of potentially invalid memory by the application triggers a page fault by the operating systems and/or underly-

ing hardware. Subsequently, a SVM fault handler routine is called which resolves the invalid access. Validity resolution is the process of sending a page request and receiving an updated page from another SVM node. SVM nodes may be either remote hosts or local threads in a SMP configuration. Current SVM systems use synchronous data validity resolution. In synchronous validity resolution a page request is sent by the SVM fault handler in response to a page fault, the SVM system waits until the reply to the page request is received, and then copies the newly received updated page into local memory. The SVM marks the page as valid and returns from the fault handler. User processing in synchronous validity resolution is blocked during the entire data validity resolution interval. AVR differs by allowing the fault handler to return and user processing to continue while data validity is resolved.

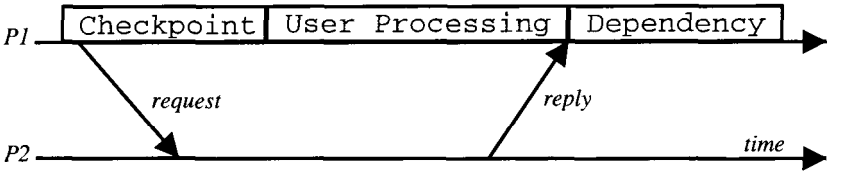

Figure 2.1: Validity Resolution Interval

In AVR, the validity resolution interval has three phases as shown in Figure 2.1. The first phase includes the initial page fault, sending of the request message, and process checkpointing. The second phase is the phase in which user process-

ing on potentially invalid data occurs. The final phase is the period where user processing stops, the reply is received, and dependency checking occurs. The AVR configuration allows user processing to occur while messages are sent and received across the network. Dependency checking, the process of determining if valid data was used during the second phase of the resolution interval, is performed once an updated page is received. Dependency checking has two possible consequences: The first is that valid data was used which means that all operations that were performed are valid and the process can continue with user processing. The second is that invalid data was used which means that all operations performed are invalid and the user process must return, or rollback, to its state at the point of the original page fault. Valid data must be used in order to provide sequential consistency in the SVM system. Hence, the requirement for a process thread to rollback to a previous point in execution.

2.3 System

The AVR concept was implemented using a modularized application layer SVM system called the Coherent Virtual Machine (CVM) [28]. CVM is a library of functions which provide the functionality of a SVM. The structure of CVM allows SC and AVR to be tested in the same environment. The AVR protocol was incorporated into CVM and was developed for use with the Linux operating system on Intel hardware. The AVR implementation was based on an existing sequentially consistent protocol.

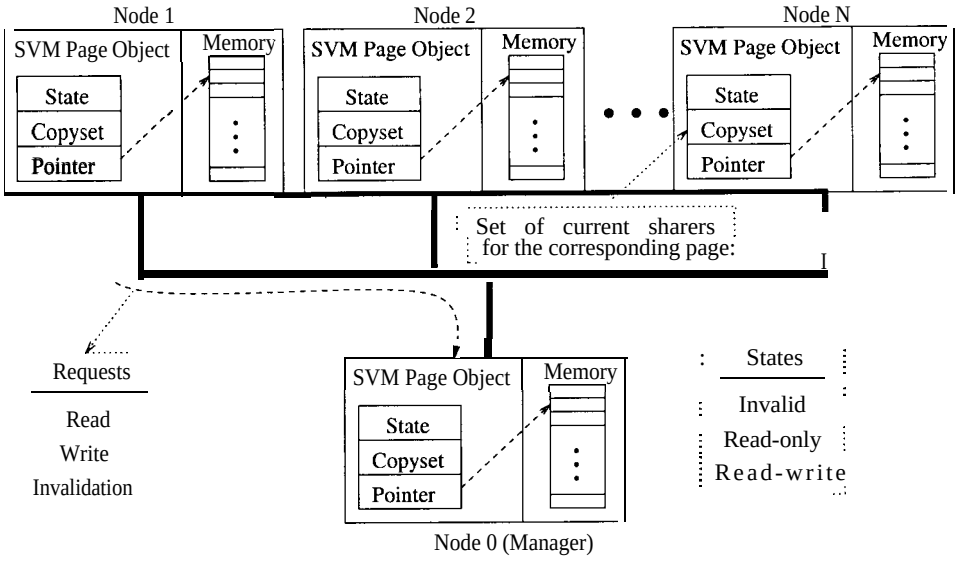

Figure 2.2: SC System

2.3.1 Sequential Consistency

A memory page is an operating system level memory management structure. A page is the smallest sized memory block for which access can be controlled. Page size is platform dependent. The default for the Intel/Linux platform is 4096 bytes. SVM systems that are page based, such as CVM, must use page sizes that are a multiple of the platform dependent page size. Page based SVM systems mimic the cache architecture and mechanisms that are present in shared memory multiprocessors. Memory consistency in the distributed SVM system is analogous to cache coherency in the shared memory architecture. In CVM, memory consistency and local access permissions for shared memory are controlled by a memory consistency protocol module. The protocol module provides the functionality required to make the distributed memories behave as a single physical memory. The CVM SC consistency module allows sequential execution of the user program in the

distributed system. It controls caching of shared pages and retrieval of updated pages from remote hosts. See Figure 2.2.

Pages are objects in the CVM implementation. Every node has a page object that corresponds to each page in the global shared memory address space. Page objects provide the additional implementation features necessary for physical pages in local memory to be consistent with their corresponding pages in remote nodes. A page object includes a pointer to the actual page in local memory, a copyset, an owner, a manager, and a state. The copyset is the set of nodes which have valid copies of the page in memory. The owner is the node that has the most recent valid copy of the page in local memory. The manager is the node that keeps track of the page's current owner and is determined in the setup routines before user processing. A valid copy of each page is present in the system at all times.

In the SC module implementation, each page exists in memory in a particular state which dictates the type of access allowed on the page. As shown in Figure 2.2, these states are invalid, read-only, or read-write. The read-only state marks the page as readable, but not writable. The page is readable and writable in the read-write state. In the invalid state, the page is not accessible. The invalid state implies that either the page has never been in local memory, has been written over with a different page due to space limitations, or was demoted from another state as a result of an incoming write request from a remote host. The SC system is a multiple-reader/single-writer system which means that multiple processes can simultaneously read from the same shared page, but only one process is allowed to write to a page at any given time.

A read or write operation performed by the user process triggers a page fault, if it operates on a memory location that does not have the required access permission. The SVM system responds to the page fault by sending a page request message to either the owner or the manager of the page. The faulting node is the requester. The requester can be the page's owner if and only if the page fault is a write fault and the page is cached in read-only state. If the requester is the manager of the page and not the owner, the request is sent to the owner. If the requester is not the manager, the request is sent to the manager. If the manager is not the owner, it forwards the request to the current owner of the page. The manager also changes the copyset for the page to show that the requester is now the owner of the page. This ensures that the manager always has the current copyset for the page and always knows which node is the owner. The current owner receives the request from the manager. It changes the copyset for the page and changes the local permissions for the page to reflect the request. The current owner replies to the requester by sending the page and a copy of the copyset. The current owner is now no longer the owner. The requester, once receiving the reply, copies the new page and the new copyset into memory and marks itself as the new owner of the page. The requester always becomes the owner, even when the page is read-only cached in multiple nodes. If the initial request is a read request, then the new owner returns to user processing with the page in a read-only state. If the initial request is a write request, the new owner sends invalidation messages to all nodes, excluding itself, listed in the page's copyset. The receivers of invalidations, mark the page as invalid, and reply with acknowledgement messages. Once all acknowledgment messages are received, the new owner changes the copyset to reflect

the invalidations, marks the page as read-write and returns to user processing. If the initial page fault is a write fault and the faulting node is currently the owner of the page, the faulting node needs to only perform the invalidation phase of the fault resolution. Waiting for all invalidation replies ensures that all nodes have a consistent view of memory.

Messages are assigned sequence numbers to allow reliable communication such that messages sent between any two processes are guaranteed to arrive in the order that they are sent. Each page has a *delta* value and a message queue. The delta value is used to reduce the ping-pong effect that occurs in page-based SVM systems. It is the minimum amount of time that a page is required to be resident in memory and is used to ensure that at least one operation completes before a remote node can obtain access to the page. The message queue is used to delay incoming messages that request pages that are in their *delta* interval.

Data validity resolution in the SC implementation is pessimistic. User processing is blocked until the new page is copied into memory and the faulting node has received the appropriate reply messages to guarantee consistency.

2.3.2 A V R

The AVR implementation is based on the SC implementation. The main difference between the two protocols is that data validity resolution in AVR is optimistic and in SC it is pessimistic. AVR is optimistic because the user process is allowed to continue execution immediately after the request message is sent. It optimistically assumes that the data is valid. The benefit of an optimistic approach is enhanced performance due to overlapped computation and data validity resolution. The

caveat to the approach is that the user process may use data which is invalid. Use of potentially invalid data requires mechanisms that allow for process checkpointing, process rollback, memory operation logging, message queueing, and page twin comparison. The AVR implementation utilizes these mechanisms in the SC framework. The sections below describe the implementation issues of AVR.

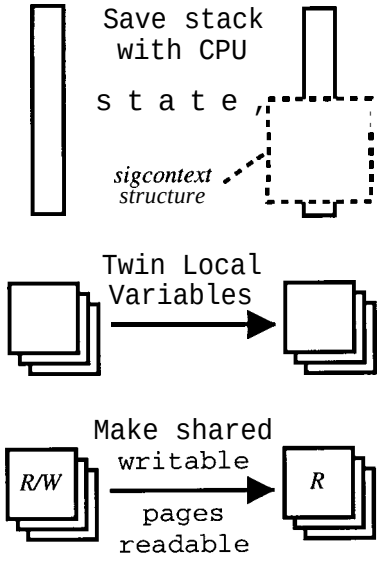

Figure 2.3: Checkpointing

2.3.3 Checkpointing

The use of potentially invalid data and the possibility of the user process having to return to a previous point in execution means that the protocol must checkpoint the user process state at the point of the initial invalid data access. The checkpoint is the saved process state which includes all the necessary information for the process to return from a point later in execution to a point previous in execution. See Figure 2.3. The necessary process state structures are the CPU registers, the

stack, the local variables, and the shared global variables (the SVM pages). The CPU registers are saved by saving the stack, and thus the *sigcontext* structure, from within the fault handler. The stack is copied into a static structure that grows to accommodate different size stacks. The local variables are saved by making twins. Shared global variables are contained in SVM pages. The process of making twins of SVM pages is not performed initially, but is deferred to the point of first access within the checkpointed interval. Initially each SVM page that currently has write permission is made readable such that writing to the page triggers a page fault. The page fault results in the execution of the segv handler which makes a twin of the page and sets the permission of the page to its original writable state. Thus, twins are only made for pages which are written during the checkpointed state. Twins do not need to be made for pages that have read-only permission, since reads do not modify variables. Handling of shared pages in this fashion means that the working set of pages can be minimized to only those which are modified, hence allowing remote access to unmodified pages to be granted.

2.3.4 Memory Mechanisms

The checkpoint page is a page for which the process does not have appropriate permission. Access of this page generates a page fault and causes delivery of a segv signal which is handled by the system's fault handler. The fault handler calls routines which send the update request message and perform checkpointing. After the request message is sent and the process is checkpointed, the handler uses *mprotect()* to set page access permissions for the checkpoint page. Access

permissions are set based whether the initial page fault was a read fault or a write fault. Before returning for user processing, the handler sets the trap flag by altering the *sigcontext* structure in the stack. The trap flag is a processor mechanism used for debugging. It is used in AVR to help provide operation logging. When the trap flag is set, the processor has a debug exception after each operation. The operating system responds to this exception by sending a trap signal to the SVM. The trap handler in the SVM receives the trap signal and performs memory operation logging. The trap handler and *segv* handler of the SVM work in conjunction to provide operation logging. The checkpoint page is kept in an unreadable state. A fault on this page triggers a *segv* signal which results in the execution of the handler. The handler sets the page permission to either read or write accessible depending on the original page fault that marked the beginning of the checkpointed interval. It also sets the trap flag and returns to user processing. Since the trap flag is set, the user process has a debug exception immediately after the re-execution of the faulting operation. The debug exception causes execution of the trap handler which logs the type and location of the memory access. The trap handler sets the page permission to unreadable to allow for future logging and returns to user processing. The page is kept in an unreadable state thereby allowing multiple accesses to be logged.

Logging memory locations for read and write operations is required for sequential consistency. Read logging is necessary to determine if a dependency exists between the local read operation and any remote write operation. A dependency signals that the memory location is true-shared. In sequential consistency, a read must return the value of the most recent write, hence a dependency indicates that

the read used invalid data and that a rollback is required. Writes are logged for two reasons: To determine dependencies in large atomic operations and to make sure correct values are copied from the remote page. Large atomic operations are atomic operations which have two memory operations, such as add x,mem. There are three phases to add x,mem: The memory location mem is read, x is added to the value of mem, and the resulting sum is written to memory location mem. The Intel specification [18] cites that if mem is unreadable, then a write fault occurs even though the read phase of the operation precedes the write phase of the operation. Hence, write operations must be logged in order to determine if read dependencies exist for these types of operations. Write operation logs are also necessary to ensure that values from the remote page are copied into memory correctly. The checkpoint page is the only page for which data validity is uncertain, hence it is the only page for which logging is required. The checkpoint page has an outstanding fault and an outstanding request during the checkpoint-resolution interval.

2.3.5 Message Handling

AVR uses two message queues: The delta queue and the protocol queue. The *delta* queue is used in conjunction with the *delta* value to avoid ping-pong behavior. The protocol queue is used during data validity resolution to delay request messages that would otherwise place the system into an inconsistent state. These messages are either requests for the checkpoint page or requests for modified SVM pages. Messages are dequeued from the protocol queue once validity of the data has

been determined. It is possible that a message can be dequeued from the protocol queue and subsequently queued in the delta queue.

Polling and interrupt handling are two strategies for handling incoming messages. Generally interrupt handling is the more expensive method because interrupts require context switches and execution of a message handler. CVM uses polling for replies. A reply is a message that is a response to a request. Examples of requests are page requests, lock-acquire requests, and barrier-entry messages. Polling as the method for handling replies is required in the CVM. Use of interrupt based handling gives rise to reentrancy problems, which can only be solved by separating the SVM and user process threads of execution. Regardless of the implementation problem, polling for replies is ideal for both SC and AVR protocols. It is ideal for SC because the user process is blocked until the reply is received. In AVR, the same logic can be applied to all replies except page request replies. Intuitively, an incoming request reply signals the end of the second phase of the resolution interval. In practice however, the interrupt tends to occur while already in the third phase of resolution thereby adding unnecessary overhead to message handling. All the programs studied performed at par or better when polling was used to handle replies. Nodes use the *select()* function call to poll for reply messages.

Incoming request messages from remote nodes in CVM are handled differently than replies. These messages are handled using the I/O interrupt mechanisms of the operating system. Use of interrupt based handling allows for the immediate response to incoming requests. Use of polling means that the system avoids costly interrupt overhead. Unfortunately polling can limit performance because requests

can only can be answered when the user process has a page fault or timer event. If a node receives a request then there is a page-level dependency between the requesting node and the receiving node. This is a page-level dependency because the page may not have an actual data dependency, but a current copy of the page is required at the requesting node to ensure memory consistency. Validity resolution of the data on the requesting node cannot complete without resolution of the dependency, hence delay in the reply can dilate the resolution interval on the requesting node. Tests show for all the programs studied, polling as a method for handling requests dilates resolution intervals and severely degrades performance.

The communication module of CVM provides network services. User Datagram Protocol (UDP), a component of TCP/IP, is the transport layer. Reliability is assured by the communication module which tracks sequence numbers for requests and replies. CVM does not use a global message sequencing system. Each node uses two sockets for each other node in the system. One socket is for incoming requests and outgoing replies, the second is for outgoing requests and incoming replies. Each node is responsible for generating sequence numbers for messages that use the second socket, the outgoing request and incoming reply socket. Hence, each node is responsible for maintaining sequence numbers for $n - 1$ nodes, where n is number of nodes in the system. Sequence numbers are gap-free. Using a timer and sequence numbers ensures communication reliability.

2.3.6 Timers, Locks, and Barriers

Timers are used in message reliability and to aid in the implementation of the delta value. The communication timer aids in message reliability by triggering

resends when replies are not received. The *delta* value is the length of time that a newly received page is forced to stay resident in memory. It is used to reduce the ping-pong effect and is typically set to be the amount of time to perform a single memory operation. It is set to 15 microseconds in CVM. This is approximately the amount of time it takes for the *segv* handler to return to user processing and the user process to perform the operation. All incoming request messages for the newly received page are queued during the *delta* interval. The *delta* timer is set when the page is copied into memory. It expires when the page has been resident for the length of time described by the *delta* value. Expiration of the timer results in the execution of a timer handler which dequeues and forwards the queued messages to the appropriate message handler.

In Linux, each process is allowed one real-time timer. The timer in CVM is currently implemented as an object which manages the timer events for the delta and the communication timers. Unfortunately, this configuration results in additional overhead in the form of conditionals, math operations, and system calls that would not be present if multiple timers were allowed.

CVM provides mechanisms for locks and barriers. Locks and barriers are synchronization operations that imply true sharing. Locks provide mutual exclusion. Barriers are fence operations which separate memory operations occurring prior to the barrier from memory operations occurring after the barrier. Due to the relationship of these synchronization operations to memory consistency, AVR performs any outstanding validity resolution actions prior to a lock acquire or barrier entry operation. Asynchronous validity resolution mechanisms are disabled until the lock release or barrier exit operation.

2.3.7 Dependency Checking and Rollbacks

Dependency checking occurs in the third phase of the resolution interval. The transition from the second phase to the third phase is marked by three possible events. The first event is a page fault resulting from the user process attempting to access a page that does not exist in memory with the required access permissions. This also includes write access to a checkpoint page that has read-only permission. The second possible event is that the deadlock avoidance counter limit is reached. The deadlock avoidance counter is a tool used to monitor and detect the condition when multiple processors are waiting for messages from each other. The deadlock avoidance counter is set to zero when the process is checkpointed and is incremented each time there is a page fault on the checkpoint page. When the counter reaches its maximum value and there is a queued request message, the third phase begins. The maximum value for the counter can be set by a command line parameter, but defaults to 30 page faults. This value provided the overall lowest execution times for all the programs tested. A timer-based approach to deadlock avoidance was tested, but was shown to degrade overall performance. Degraded performance was caused by timer object overhead and the overhead resulting from additional interrupts. The final possible event leading to entry into the third phase of the resolution interval is the case where a new request has been received that indicates that the reply is in the requester's buffer. One of these three events must occur for the transition to the third phase of the resolution interval. Experimentation shows that, in all the applications tested, the primary influence on the length of the user processing interval is the occurrence of the first event, a fault on an inaccessible page, thus indicating that pre-fetching may

increase performance. Pre-fetching, however, is not studied in this work, but is indicated as a possible avenue of future research.

If the third phase of the resolution interval is reached by the occurrence of the second situation and the reply is not in the incoming message buffer, a rollback is automatically performed. The second situation indicates that the system may be in a deadlock state, such that its request is queued by some other remote process. The system must rollback in order to allow the queued request message(s) to dequeue. Once the user process has returned to its original state, it will page fault on invalid data and checkpoint again since a reply to its request was not received. If a reply is found in the incoming message buffer, resolution will proceed in the manner required by the first and third situations.

In the first and third events of the second phase of resolution, the system will poll until it receives the reply. The reply will, in most cases, include the most recent copy of the page. Comparison of this page with the active checkpoint page, the checkpoint page's twin, and the operation log will determine if a rollback is required. A rollback is required if a dependency exists between incoming data and data that was used. Conflicts with the read entries in the log and the new page indicate that a rollback is necessary. The new page is written over the active checkpoint page except in locations marked by write entries in the log. If a rollback is not required, the process checkpoint, operation log, and page twins are deleted and the system returns to user processing. Memory objects, which include SVM pages and local storage, are returned to their original access permissions. If a rollback is required, active memory objects are replaced with the previously saved and unaltered twins. Page permissions are returned to their

original states. The process is returned to its original point in execution by copying the saved stack over the current stack and then changing the stack pointer. The stack is a process object that shrinks and grows. Its size at any given point depends on the process's function call depth and the number of local variables. The previously saved stack may be larger than the stack existing immediately prior to rollback. Since functions use the stack for storage of local variables and as storage of reference information for other variables, it is necessary to change how the stack is handled for the function call which performs the rollback. If the older stack is larger, the stack pointer must be set to point at a position that is above the incoming older stack. This modification along with use of global variables ensures that the rollback function operates correctly and the process can be returned to its previous state in program execution.

2.4 Correctness

In terms of the AVR system, correctness is considered to be the ability to provide sequentially consistent execution of user programs. There are two central issues important to correctness in AVR. They are message sequencing and page resolution. With regard to messaging, the AVR system is correct in its execution. The AVR module was adapted from the SC module. The SC module has been shown to be sequentially consistent because the order of messages is guaranteed[26]. AVR does not disturb this ordering. Messages, from any given remote node, are delivered in the order they are received. Thus relative to messaging, AVR executes correctly.

Page resolution is an important consideration to the idea of correctness. The mechanisms of AVR ensure that, upon receipt of a reply containing an updated page, data dependencies are identified and that new data elements are copied over old data elements. The updated page is compared to the page twin and the operation log. This comparison identifies invalid data which has been accessed during user program execution. Access of invalid data indicates that the process must rollback, since information regarding intra-process dependencies is not present. If no invalid accesses are identified, the updated page is copied over the active page except in locations that were modified during the resolution interval. The page resolution mechanisms ensure that data dependencies are identified and that a rollback is performed when dependencies exist between local and corresponding remote memory locations. These mechanisms coupled with message sequencing mechanisms guarantee sequential consistency and correctness of AVR.

2.5 Limitations

There are two groups of limitations relevant to the AVR implementation. They are limitations with respect to the types of programs that can be used and limitations with respect to the implementation itself. The AVR implementation does not allow for proper handling of program structures such as those related to sockets and file I/O. Programs which use these types of structures must use AVR function calls which disable and re-enable the asynchronous validity resolution capabilities of the system. These issues have not been addressed in the AVR implementation since they are not critical to AVR research. Moving the AVR im-

plementation into the operating system layer and adding consistency functionality to the operating system resource functions would eliminate the need for the programmer to explicitly use the AVR functions required by the current middleware implementation.

The performance of the implementation is limited by the logging and timer mechanisms. The performance of the logging mechanisms are limited because the size of memory which a memory operation operates on cannot be readily determined. This is viewed as a result of a processor architecture limitation since adequate debug registers could provide this information when a page fault occurs. Without size information, AVR must assume that the faulting reference is for the largest memory object specified by the processor architecture. The largest memory object is the largest amount of memory that can be referenced by one operation. In Intel architecture the largest memory object is 80 bits. Without the ability to determine the size of a referenced location, AVR must assume that the reference is for the largest possible object. Hence, the operation log may not represent the true granularity of memory references and may incorrectly indicate that there is true-sharing during validity resolution. AVR is currently implemented on Intel architecture and uses eight bytes as the minimum size of log entry. Intel's extended 80 bit memory operations are rarely generated by compilers and because of this were not addressed in the AVR system implementation. Slight modification of the logging mechanisms would be required to include these operations in the system.

The timer mechanism limitation has already been partially discussed. This is a limitation imposed by the Linux operating system. Each process in Linux is allowed three timers. Only one, however, functions as a traditional wall-clock

timer. In order to use this one timer to time multiple events, a large amount of overhead in the form of interrupts, math operations, and system calls is added to the system. Timer object overhead degrades performance of the AVR system. The timer limitation could be removed if either the SVM system was moved into the kernel or if Linux processes were allowed additional timers.

2.6 Summary

AVR is a protocol for providing concurrent validity resolution and user processing. The AVR implementation is an extension of SC that provides concurrency while maintaining sequentially consistent execution. The AVR implementation does have some limitations related to the test platform. In general, however, the AVR CVM implementation correctly captures the AVR concept and is suitable for testing a wide range of programs.

Chapter 3

PERFORMANCE

This chapter presents an evaluation of the Asynchronous Validity Resolution (AVR) protocol. AVR was implemented in the Coherent Virtual Machine (CVM) system. Ten programs were used in the evaluation: Two molecular dynamics simulations (Water, Spatial), a lower-upper block matrix decomposition (LU), a volume renderer (Volrend), a ray tracing program (Raytrace), a Fast Fourier Transform (FFT) , an eddy current simulation (Ocean), a radix sort (Radix), a matrix multiply (Matmul), a Jacobi relaxation (Jac), and a Gauss-Seidel relaxation (Gauss).

The evaluation of AVR has two elements: a direct comparison of execution times, an analysis of AVR in the context of application sharing patterns. AVR is compared to a traditional sequentially consistent protocol (SC) such as that used on the SGI Origin2000 supercomputer. The evaluation of AVR does not include a comparison to the relaxed SVM protocols such as Release Consistency or to explicit message passing protocols. These protocols either follow different

programming paradigms or impose restrictions on the types of programs allowed to execute in the SVM system. Both SC and AVR use the traditional parallel programming paradigm that is found in shared memory architectures such as the SGI Origin2000. This style of programming is an intuitive extension of regular uniprocessor programming which does not require explicit cache synchronization operations. The design of AVR targets performance limiting problems of SC while maintaining ease of programming. The comparison of AVR to SC is a measure of performance gain that can be achieved by altering sequential consistency protocol design. Four of the eleven programs perform better using AVR.

The behavior of AVR is analyzed with regard to application sharing patterns. Application sharing patterns are related to the frequency of checkpointing and necessity of process rollback. Sharing patterns also are related to the type of checkpoints made in the system and the overall resolution interval times.

3.1 Experimental Environment

3.1.1 Hardware Platform

The experimental platform is a 32 processor / 16 node Beowulf [46] cluster. Each node has dual 600 MHz Pentium III processors and 512 MB of main memory. Nodes are connected with 100 Mb switched Ethernet.

3.1.2 Basic Operation Costs

The basic system level costs relevant to the evaluation are the times required to perform memory protection, page twining, operation logging, and to send a peer to peer message in the network. The cost of performing an *mprotect* system call is 20 microseconds. *Mprotect* system calls are used to provide operation logging and for protection of pages which have write permission during the resolution interval. The cost of making a twin of a page is 43 microseconds. Twins are made for the checkpoint page during the first phase of the resolution interval. Twins are also made for read-write pages that are accessed during the checkpoint interval. The cost of trapping a memory operation is 78 microseconds per memory operation. The cost for logging a memory operation is the cost of the trap in addition to the cost of one *mprotect* system call. These three base operating costs influence the percent of user work that can be accomplished during the resolution interval and the overall length of time of the resolution interval. The peer-to-peer network time for a 4192 byte message, a typical message size in CVM, is 1.4 milliseconds. The network time is the available time for asynchronous user processing.

3.2 Application Suite

Eleven programs were selected for this study: Water, Spatial, LU, Raytrace, Volrend, FFT, Ocean, Radix, Matmul, Jacobi, and Gauss. Water, Spatial, LU, Raytrace, Volrend, FFT, Ocean, and Radix were taken from the SPLASH-2[49] suite of applications. The SPLASH-2 program suite is a set of scientific, engineering, and graphics programs that exhibit a wide range of data sharing patterns

and were written for hardware cache-coherent machines with cache-line granularity. These programs were selected because they are well studied and provide a diverse pattern of memory sharing. The other programs were written explicitly for this work and were selected because they comprise two computational cores of examples of scientific code which needs to be parallelized by researchers in many fields. Together, the two groups of programs provide an application suite which is representative of common parallel programs. Use of this suite adds validity to the evaluation of AVR. Table 3.1 gives a brief description of each program along with its inherent memory sharing pattern.

Application	Description	Inherent Sharing Pattern
Water N^2	System of water molecules, n/p per processor	Migratory & 1P-MC
Water Spatial	Water with spatial directory, cell assignment	1P-MC
LU	LU blocked factorization of dense matrix	1P-MC
Raytrace	Scene rendering	Migratory
Volrend	Renders 3-D volume data into image	Migratory
FFT	High-performance FFT kernel	1P-1C
Ocean	Eddy current simulation, iterative near neighbor	1P-1C
Radix	Ascending sort of integer keys	1P-1C
Matmul	NxN Matrix Multiply	1P-0C
Jacobi	Iterative Matrix Solver	1P-MC
Gauss	Iterative Matrix Solver	1P-MC

Table 3.1: Application Sharing Patterns

The experimental suite of programs shows a diverse pattern of memory access. This is desirable for testing a page-based shared virtual memory system. Each application has an inherent sharing pattern, the pattern specified by the programmer. A page based SVM often will impose a different sharing pattern because a page is the smallest memory object that can be shared between processors. This is called the induced sharing pattern. The induced and inherent patterns of sharing

for an application can be the same in some circumstances such as when problem decompositions fall on page boundaries or when application objects are individually *malloced*. Typical patterns of sharing are 1P-1C (one producer with one consumer), 1P-MC (one producer with multiple consumers), and Migratory [22]. 1P-1C is a pattern in which one processor writes a location that is read by another single processor. Migratory access is usually 1P-1C access that occurs consecutively across the processors in the system. Variables requiring mutual exclusion are typically accessed in a migratory fashion.

False-sharing and fragmentation are characteristics of SVM systems. False-sharing occurs when two or more processes use logically unrelated data that are stored on a common page of memory and at least one processor performs a write operation on that data. When a page is false-shared, it is sent back and forth between participating processors. This ping-pong effect degrades performance of the SVM system. Fragmentation occurs when a processor fetches an entire page, but only requires access to part of the page. An application's pattern of memory access determines whether the application will suffer from false-sharing and fragmentation imposed by the page granularity of the SVM. In regard to the size of a page, memory access can be considered to be fine-grained, medium-grained, or coarse-grained [22]. Fine-grained access is a set of read or write accesses that operate on a region of memory much smaller than a page. Medium-grained access is a set of read or write accesses that operate on a region of memory which is larger than that of fine-grained access, but is also smaller than the size of a page. Access that operates on entire pages is considered coarse-grained. Fine-grained and medium-grained write access signify false-sharing while fine-grained

and medium-grained read access signifies fragmentation. Coarse-grained access is page sized and, therefore, does not result in fragmentation or false-sharing. In some programs, the access granularity is dependent on the size of the input and the number of processors used in computation.

The applications in the experimental suite provide a wide range of application sharing patterns and access granularities. They are ideal tools for evaluation of a shared virtual memory system.

3.2.1 Water

Water is a molecule simulation which uses an array for storage of the individual molecules. The molecules are partitioned in a simple n/p block format. The majority of computation in Water occurs in its force calculation phase, an $O(n^2)$ algorithm, which displays a course-grained 1P-MC access pattern. Water also has a less dominant migratory memory access pattern in which each processor must update molecules assigned to other processors. Updates are made using locks leading to a 1P-1C access pattern. The amount of false sharing and fragmentation depends on the alignment of partitions to page boundaries.

3.2.2 Spatial

Spatial solves the same problem as water, but uses a different partitioning scheme. Spatial is *O(n)* and is a more efficient algorithm [49]. The simulation domain is broken into 3-D cells which are assigned to individual processors. The molecules in each of the cells are stored one per page and are elements in a linked-list

data structure. The simulation requires communication from each molecule to all other molecules that are within a cut-off radius. The domain decomposition is advantageous since processors only need to communicate with processors assigned with adjacent cells. During the simulation, molecules can migrate between cells thus requiring the use of a lock to protect the linked list structures. For large problems the access pattern is mostly 1P-1C. For small problems, it is mostly 1P-MC [22].

3.2.3 LU Decomposition

LU decomposition is a blocked factorization of a dense matrix. It operates by creating a NxN array of BxB blocks for a nxn matrix such that n = NB. Each block is page aligned and allocated contiguously in memory eliminating false-sharing and fragmentation. Blocks are assigned to processors in a 2-D scatter decomposition. Memory accesses in LU are coarse grained and follow a 1P-MC scheme.

3.2.4 Raytrace

Raytrace is a computer graphics application which renders a three dimensional scene using ray tracing. A ray is traced from a central point, the eye, through each pixel of the image plane into the image scene where it may strike a scene object. The process is recursive such that each contact with an object generates new rays which are stored in a ray tree. The result is that each pixel in the plane has a ray tree. The pixels in the plane are partitioned among processors in contiguous

blocks. Distributed task queues which employ task stealing are implemented. The data accesses are very unpredictable in this application [49]. Access to scene data in Raytrace is fragmented (OP-MC). Task queue and pixel writing are fine-grained with frequent false-sharing. Raytrace is also marked by page level fragmentation (MP-MC)[22].

3.2.5 Volrend

Volrend is a three dimensional computer graphics rendering program. It is similar to Raytrace and uses a ray casting technique. It works from an image plane and includes distributed task queues. A ray is shot through a pixel in the image plane into a 3-D scene. Unlike Raytrace, the rays do not reflect. Instead, color for a pixel in the image plane is determined by sampling and interpolating along the ray's linear path. Memory accesses are irregular and input-dependent [49]. Volrend has a relatively small set of read-only data compared to that in Raytrace. False sharing and fragmentation occur in Volrend.

3.2.6 FFT

FFT is a Fast Fourier Transform algorithm. Two $\sqrt{n}$ x $\sqrt{n}$ matrices are used to store n points to be transformed and n *roots of unity* points. Each processor is assigned contiguous sets of rows in each matrix. Communication in the kernel occurs as every processor transposes a $\sqrt{n/p} \times \sqrt{n/p}$ matrix from every processor including itself. Write access is coarse. Read access granularity depends on the

size of ***n*** and ***p.*** Fragmentation can occur, but only when $\sqrt{n/p}$ is not a multiple of page size. False-sharing does not occur.

3.2.7 Ocean

Ocean is a simulation of ocean movements based on eddy and boundary currents. The domain is partitioned into grids. Read operations on borders of adjacent partitions represent the communication. Write access is coarse since each processor writes only to its own partition. Read granularity is fine if occurring on a column border, medium or coarse if occurring on a row border. The access pattern is generally 1P-1C. Ocean suffers from fragmentation, but not false sharing.

3.2.8 Radix

Radix is an iterative sorting algorithm which performs one iteration for every digit in the keys. The set of keys is block partitioned to the processors. Each processor generates a histogram based on its ***n/p*** keys. Each processor's histogram is merged into a global histogram which is then used by all the processors to permute the keys into a global destination array. The permutation of the keys is irregular and accesses are fine to medium granularity thereby leading to false-sharing. Read access is coarse grained.

3.2.9 Matmul

Matmul is the textbook matrix multiply program. It performs multiplication of two n x n matrices using a row based partition. Matrix multiply is a central com-

ponent in many matrix operations. Matmul uses coarse-grained read accesses and medium-grained write accesses. It exhibits false sharing, but not fragmentation.

3.2.10 Jacobi

Jacobi is an iterative matrix solution method for Ax = B. In each iteration a new x_i is computed based on values of the old x_{i-1}. At the end of each iteration, the new values are copied into the old array. The number of iterations is controlled by monitoring the difference between new values and old values. If the difference is less than some tolerance, iteration is halted. The number of iterations required for the method to converge to a solution is dependent upon the data. The Jacobi routine used for the study was modified to produce ten iterations regardless of convergence. Jacobi exhibits false sharing.

3.2.11 Gauss

Gauss is the Gauss-Seidel method, a modification of Jacobi iteration. In Gauss-Seidel, the new values of x_i are used as they are computed. Unlike Jacobi, array x is read and written as new values are computed resulting in a heightened ping-pong effect. Gauss suffers from false sharing and fragmentation depending on the size of the ***n/p*** partitions. Gauss was modified for this study to produce 10 iterations.

3.2.12 Diversity

These eleven applications are diverse in their memory access patterns and granularity. Since the design of AVR focuses on the reduction of costs associated with false-sharing in SC, only programs which suffer from false-sharing are expected to perform better using AVR. Volrend, Raytrace, Jacobi, Gauss, and Matmul produce false sharing and are expected to perform well. Radix also produces false sharing, but is known to perform poorly in SVM due to the need for frequent communication[22]. FFT and Ocean exhibit fragmentation, but not false sharing. Water, Spatial, and LU do not exhibit fragmentation or false sharing and hence would not be expected to benefit from AVR. FFT, Ocean, Water, Spatial, and LU have been included in this study for completeness.

The size of the inputs used in the experimentation are listed in Table 2.

Program	Input
Water N^2	512 molecules
Water Spatial	512 molecules
LU	512-by-512
Raytrace	teapot
Volrend	256^3head
FFT	256K points
Ocean	256-by-256
Radix	256K keys
Matmul	1024-by-1024
Jacobi	2500-by-2500
Gauss	2500-by-2500

Table 3.2: Application Input Sizes

3.3 Results

3.3.1 Timings

Experimental Procedure

The application suite was tested using a Beowulf computational cluster. Although each node in the cluster offered dual CPUs, only one CPU per node was used in the experiments in order to minimize the effects of operating system context switching and the effects attributable to assignment of specific tasks to processors located on common nodes. Each program in the application suite was run thirty times using SC in 2, 4, 8, and 16 CPU configurations and thirty times using AVR in 2, 4, 8, and 16 CPU configurations. Measurements such as execution time, lengths of resolution intervals, and number of sigios were collected and averaged for each configuration of program, protocol, and processor count. For each program in the application suite, the 2 processor SC measurements were compared with the 2 processor AVR measurements, the 4 processor SC measurements were compared with the 4 processor AVR measurements, the 8 processor SC measurements were compared with the 8 processor AVR measurements, and the 16 processor SC measurements were compared with the 16 processor AVR measurements. These comparisons were performed using a two-tailed t-test. The null hypothesis of equivalence was rejected at the 0.05 level for these comparisons. The results of the comparisons are not shown here.

AVR Average	SC Average	AVR Variance	SC Variance	T-Test
5.42	5.52	0.03	0.05	0.0005

Table 3.3: Water 4 CPU 100 Run Test Case Data

An additional 4 processor test case experiment of 100 runs of Water using SC and 100 runs of Water using AVR was also performed. The null hypothesis of equivalence for average execution times for SC and AVR for this 100 run experiment was tested, and subsequently rejected, with the two-tailed t-test. Results are reported in Table 3.3. Based on the statistical significance testing of the Water application, and the similarly low variance in the performance of all the algorithms, we believe that a direct comparison between the 30 run averages in the other algorithms is sufficient.

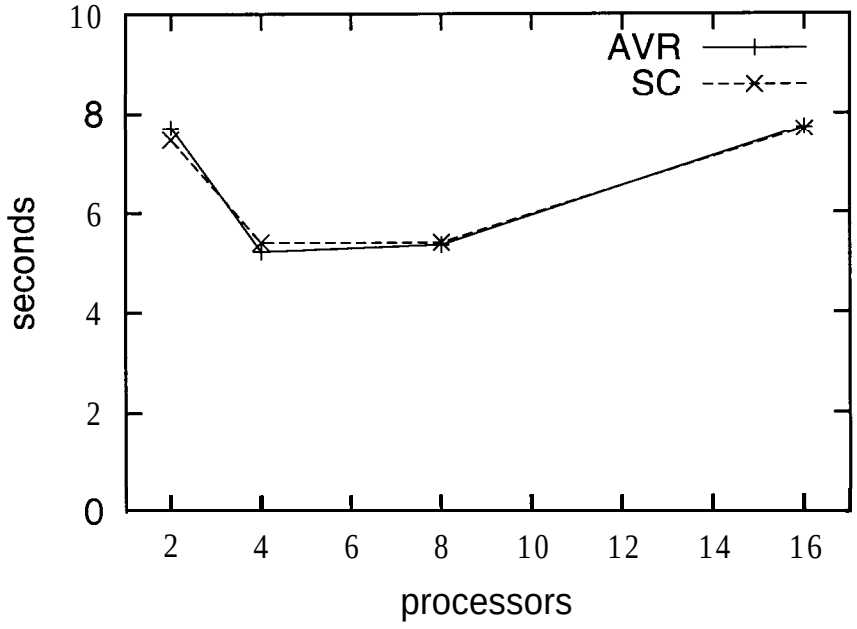

Figure 3.1: Water time vs. #processors

Water

AVR performs 3% better than SC for the Water application. See Figure 3.1. This was unpredicted since true-sharing occurs on all SVM pages. AVR performance

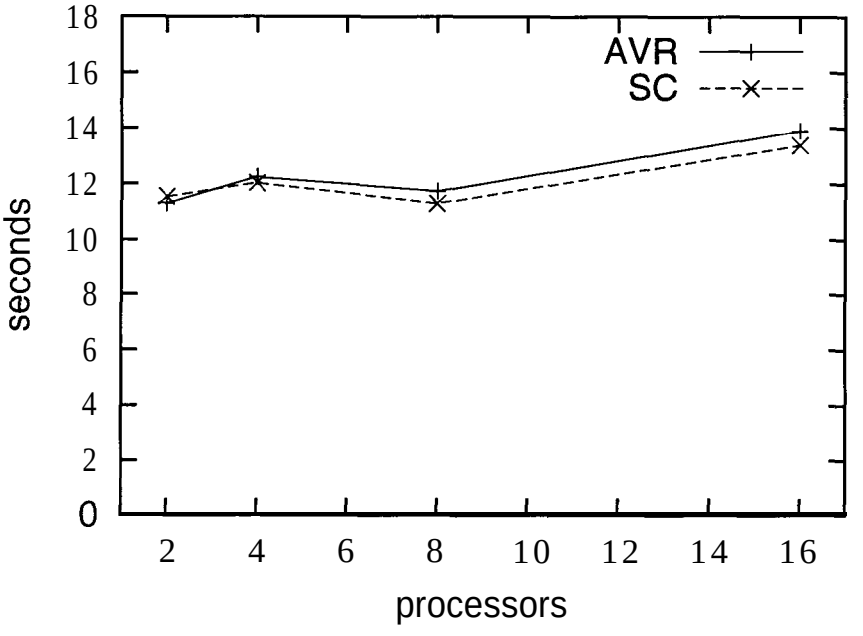

Figure 3.2: Spatial time vs. #processors

is due to overlapped write validity resolution with computation and the ability of the protocol to revert to regular SC when true-sharing is detected. In the AVR four processor configuration, invalidation checkpoint and resolution costs were 452 microseconds. SC invalidation costs were 723 microseconds. This means that in contrast to SC, AVR provides an additional 271 microseconds per invalidation for user computation. Performance gains due to invalidations offset the costs of true-sharing rollbacks in the 4 and 8 processor configurations, but not in the 2 and 16 processor configurations. In the two processor configuration, the number of rollbacks is greater due to larger working sets, thus negating the invalidation cost savings. In the 16 processor configuration, the number of per processor invalidations decreases by 75% while the number of rollbacks only decreases by

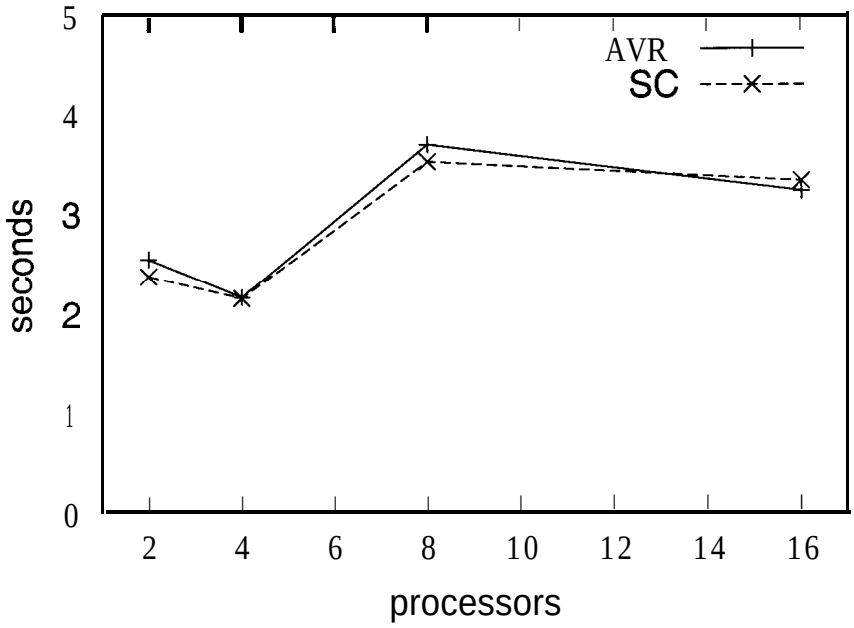

Figure 3.3: LU time vs. #processors

23%. The cost savings from the total number of invalidations is not large enough to recoup the losses from rollbacks.

Spatial

AVR performs on par with SC for Spatial (Figure 3.2). Performance is due to overlapped computation with invalidation message handling. In the spatial implementation of the water simulation, each molecule is allocated on a separate page. Spatial does not suffer from false sharing due to method of storage.

LU Decomposition

Figure 3.3 shows execution times for LU. AVR performed on par with SC for the LU kernel. Although 93% of all checkpoints required a rollback in the four

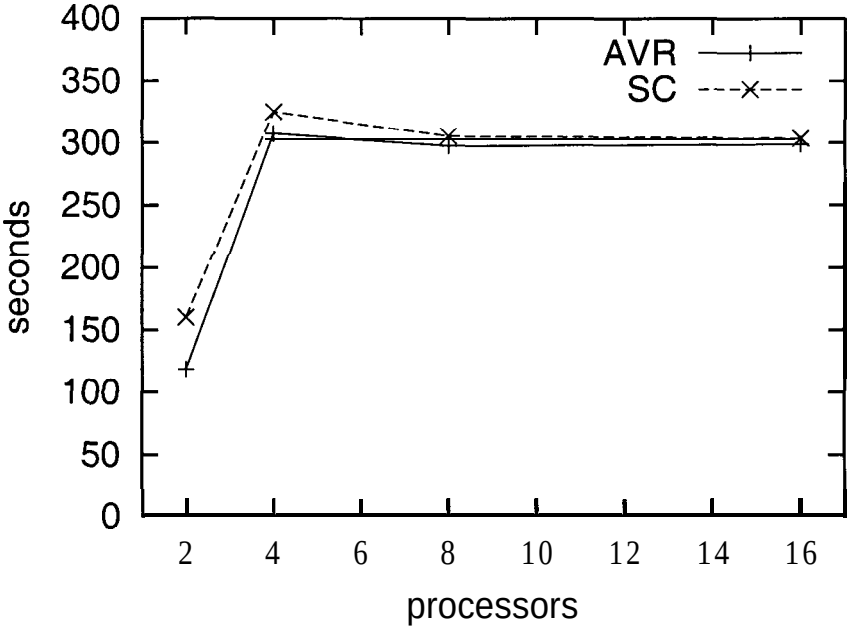

Figure 3.4: Raytrace time vs. #processors

processor runs, AVR produced an execution time equivalent to SC. AVR performance is attributed to a relatively low cost to checkpoint and performance gains in invalidation handling. The cost of performing a checkpoint can be measured by examining the number of memory protection system calls required to protect writable pages and the number of page twins that need to be made for the set of protected pages. In LU, 36 pages per checkpoint, not including the checkpoint page, required access protection. Out of 1017 checkpoints only 429 pages were twin-ed. The total number of invalidations in both the SC and AVR systems was 1178. Invalidation resolution intervals in SC were measured to be 318 microseconds. The protocol overhead for an invalidation in both SC and AVR is 61 microseconds. The remaining time, 257 microseconds, is the sum of the wire time for the request, the wire time for the reply, and the time spent handling the

invalidation request on the remote node. AVR capitalizes on this wait time by allowing user processing, while SC blocks.

Raytrace

AVR performed better then SC for all processor counts for both the data sets. Execution times for SC were up to 35% longer than those using the AVR protocol. See Figure 3.4 for the teapot data. Fastest times were produced using two processors. Both protocols slowed at processor counts greater than two due to an increase by 50% in the frequency of locking. In the two processor runs, where the difference in execution times was greatest, AVR used only 66% of the messages and *sigios* required by SC. Additionally, the length of time spent in sigio events in AVR is 43% less than the amount of time spent in sigio events in SC. Raytrace has little true-sharing, explaining why less than 2% of checkpoints resulted in a rollback. AVR performance is due to a larger number of operations being completed per page fault, which reduces the number of messages that need to be sent and increases the proportion of local invalidations. The overall result is a reduction in the ping pong effect that is found in SC.

Volrend

Volrend performs well using AVR. The fastest times were 4.5% better than SC in the 16 processor configuration. See Figure 3.5. Overlapped communication and computation reduces the performance penalty of increased contention for shared memory that is found in SC for runs using 8 or more processors. Average validity resolution times in AVR are half the length of those in SC. Volrend is a task

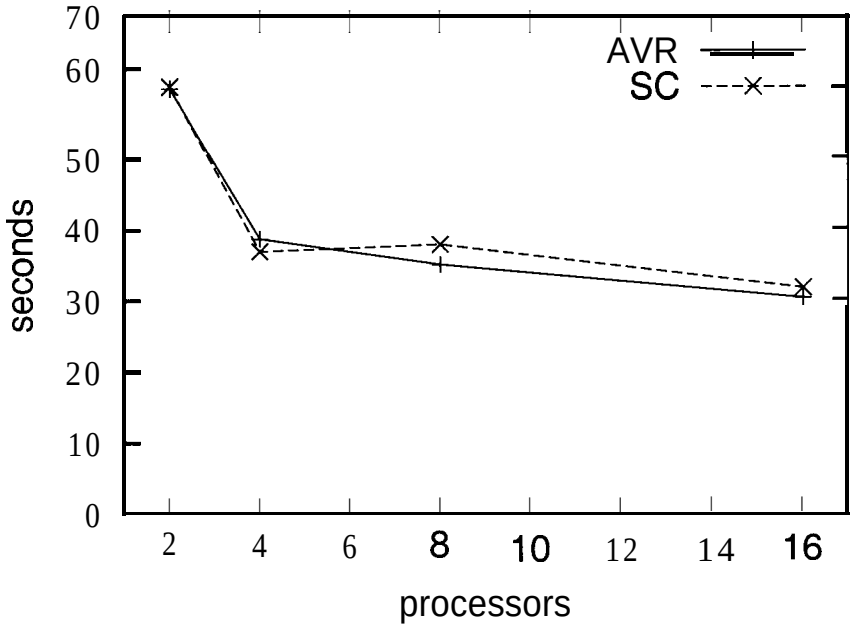

Figure 3.5: Volrend time vs. #processors

queue based program, like Raytrace, that utilizes locks to protect the queues. AVR requires 30% fewer lock messages and 11% fewer total messages than SC. At 16 processors, AVR averages 23 *mprotects* and 0.2 twins per checkpoint. One percent of all checkpoints result in rollbacks.

FFT

The best execution times for FFT were observed in the 16 processor runs (Figure 3.6). The execution time for SC was 50% of that for AVR. The average number of rollbacks was 16813 or 56% of the number of checkpoints. The average time to create a checkpoint in AVR is 2583 microseconds while the average time to resolve a page fault in SC is 1859 microseconds. The performance of FFT in AVR is limited due to the number of *mprotects* required to protect the working

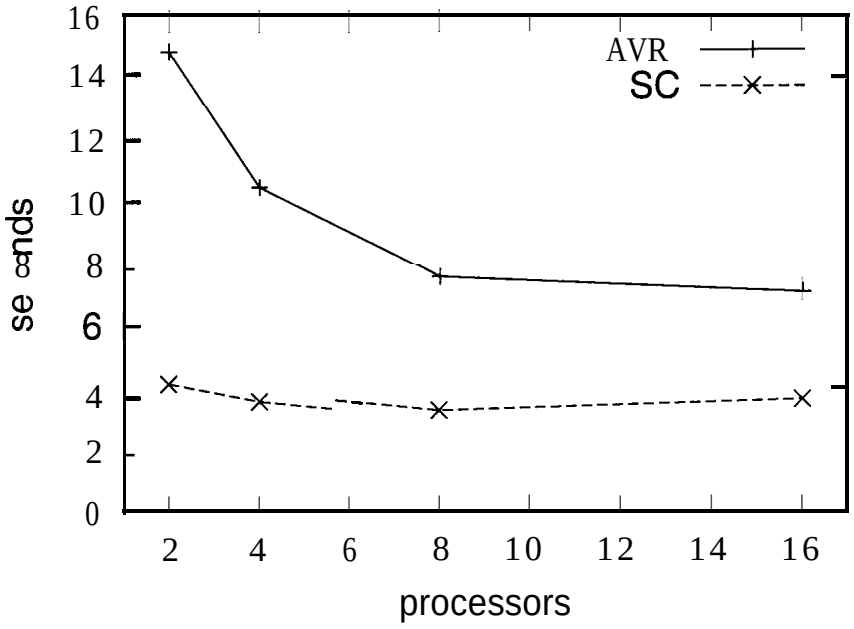

Figure 3.6: FFT time vs. #processors

set of pages. On average, 57 pages per checkpoint in FFT are needed to be protected from modification. The memory protection system call, *mprotect*, *was* called 101 times per checkpoint to perform the protection operation. Twelve of the protected pages are written and twin-ed during the resolution interval. The cost of executing *mprotects* and twin-ing causes resolution intervals to be long. Long resolution intervals lead to long barrier events since it takes longer for the distributed memories to become consistent. AVR spent 4 times longer than SC in each barrier event. Overall, the costs required to maintain consistency in FFT outweighs any performance gains provided by overlapped communication and computation.

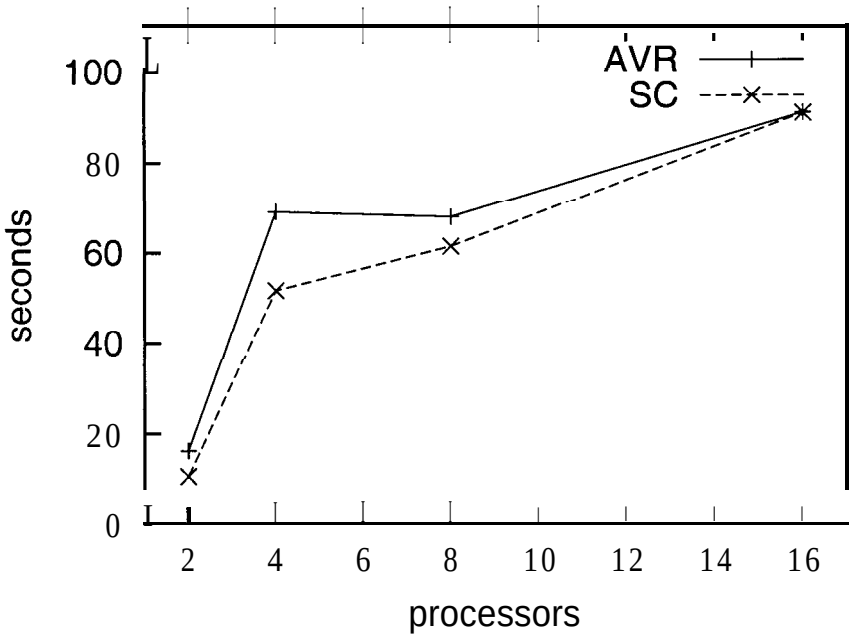

Figure 3.7: Ocean time vs #processors

Ocean

As shown in Figure 3.7, SC execution times for Ocean were 4.5% better than those for AVR. AVR performance is poor due to the number of pages that need to be protected from access during each checkpoint. In the two processor run, there are an average of 1250 pages which require protection, 83 *mprotect* system calls, and 2.3 pages which require twins for each checkpoint. The page protection overhead in AVR is greater than the average validity resolution interval in SC.

Radix

Execution times for Radix are shown in Figure 3.8. SC outperforms AVR in Radix. False-sharing occurs in Radix, but true-sharing dominates. AVR performance is

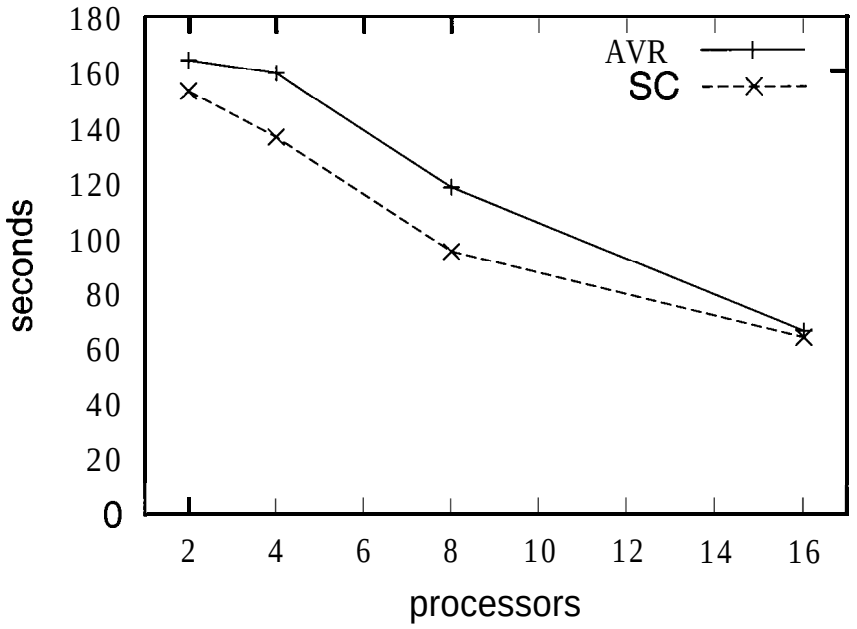

Figure 3.8: Radix time vs. #processors

low due to the pattern of sharing, the low computational complexity per false-shared memory operation, the size of logging granularity requirement and to a large number of write accesses being identified as true-sharing.

In radix, a global array of keys is divided up across processors. Each processor works on its own array segment and develops a histogram describing how to arrange its assigned keys. The arrangement is an ordering based on a particular digit in the keys. Each processor then writes its keys to locations in the global array, by comparing its locally generated histogram to the histograms of the other processors. This process iterates until all digits have been examined. True-sharing and false-sharing exist in Radix. True-sharing occurs as each processor reads its segment from the global array. AVR cannot improve on SC for these operations other than to overlap invalidation with computation. The true sharing pattern

ensures that a processor will need to perform one or more rollbacks, if its assigned segment in the global array is written to by some other processor. In the 16 processor runs AVR performs rollbacks for 6.5% of the number of checkpoints. This number is substantially smaller than it could be, since AVR is configured to revert to SC on pages with a true-shared history. False-sharing occurs when the updated keys are written to the global array. AVR is designed to work well in this type of access, but cannot live up to its potential because the computational complexity of the write operations is low. The operations complete successfully and rollbacks are not required, but the overhead of resolving the data validity outweighs the gain generated by performing the operations. The average number of operations is 4 for every checkpoint and the average AVR validity resolution interval is 2340 microseconds. Each checkpoint in Radix requires the protection of 35 pages and the generation of 1.1 twins. The average validity resolution interval in SC is 2060 microseconds. The difference in resolution intervals is 280 microseconds, yet a write operation takes less than 1 microsecond to complete.

The logging granularity requirement is a factor for the performance of AVR and is discussed in an earlier section. The requirement is basically that for any memory operation 8 bytes need to be logged as used regardless of the number of bytes the operation actually used. Radix is an integer based program and uses only 4 bytes per access on the Intel architecture. Hence, some memory locations in Radix are incorrectly being marked as having a data dependency, thereby increasing the probability of a page being treated as true-shared rather than false-shared. The requirement to handle write accesses as a read-write pair increases the number of accesses identified as true-sharing.

The operations which write ordered keys into the global array are on pages which are false-shared in the system and are the cause of 30494 dirty/write faults and 7655 dirty/write checkpoints in the 16 processor runs. As discussed in the memory mechanisms section, write operations are logged as read operations due to the existence of instructions which perform atomic read and write operation pairs on memory. Since, the writes are treated as reads and the memory locations on which the write operations act indicate that there are data dependencies, the pages are incorrectly treated as true-shared. Overall, performance of AVR for the Radix program is limited by implementation requirements.

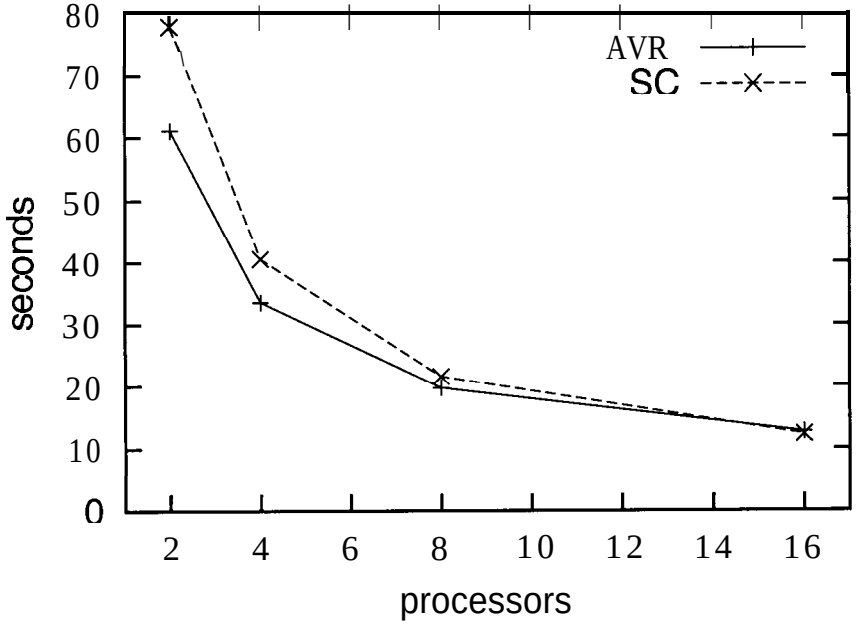

Figure 3.9: Matmul time vs. #processors

Matmul

AVR outperforms SC for Matmul. See Figure 3.9. The amount of false-sharing in Matmul depends on the size of the problem and the number of processors used. Two data sizes were tested: 1024~1024 and 101321013. See Table 3.4 and Table 3.5 for those results.

Processors	2	4	8	16
AVR	61.06s	33.57s	19.69s	12.69s
SC	77.90s	40.56s	21.54s	12.24s
% gain for AVR	21.61	17.23	8.58	-3.67
False-shared pages	0	0	0	0

Table 3.4: Matmul 1024x1024 Timing Data

Processors	2	4	8	16
AVR	111.61s	60.28s	32.89s	19.33s
SC	127.33s	66.83s	34.67s	18.84s
% gain for AVR	12.34	9.80	5.13	-2.60
False-shared pages	1	3	7	15

Table 3.5: Matmul 1013x1013 Timing Data

The larger data set, 1024~1024, does not exhibit any false-sharing for the chosen processor counts. The 101321013 size data set exhibits false-sharing as shown in the Table 3.5. The performance gains of AVR for the larger data set is due to overlapped validity resolution with computation as each processor acquires its working set of pages from the page manager. False-sharing occurs in the smaller data set and is a cause for contention in the system. The smaller data set has longer times due to false-sharing and the number of read shared pages in the system. Runs using these two data sets show that the smaller data set has a larger number of queued messages, a greater number of twins per checkpoint, a

greater amount of time spent in *sigio*, and **2.5%** more pages which need protection per checkpoint. These differences are a result of the added contention placed on the system due to the data partitions not being page aligned.

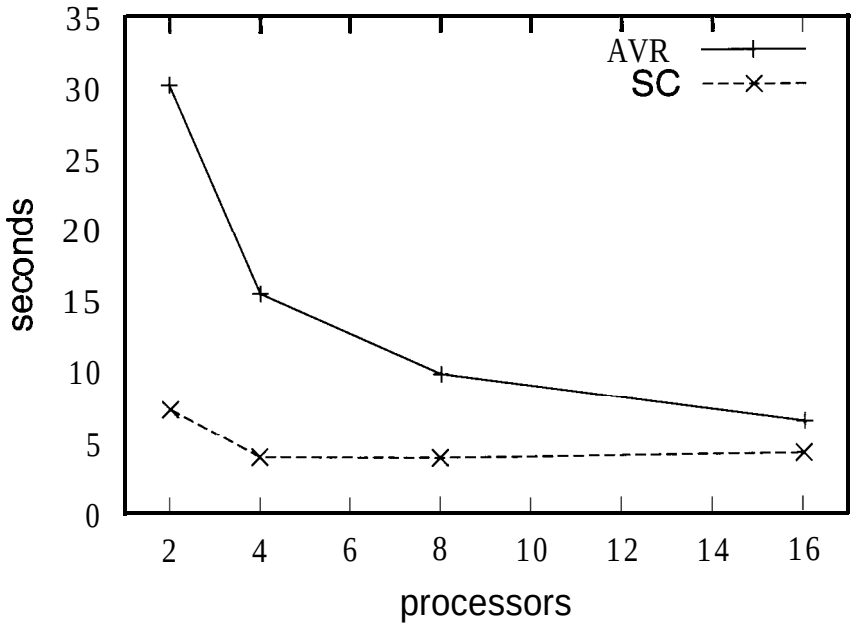

Figure 3.10: Jacobi time vs. #processors

Jacobi

AVR did not provide any improvement in performance over SC for Jacobi Iteration (Figure 3.10**).** The design of Jacobi ensures that the majority of working-set pages are cached in read-only state. The pages used to store the new unknown array are write shared and not read-write shared, avoiding large message traffic from invalidation messages. SC validity resolution intervals for Jacobi are up to 63% shorter than that of AVR for runs using less than 16 processors. Less than 2% of checkpoints resulted in rollbacks in runs up to 16 processors. False sharing

occurs in the new unknown array as new values are calculated and updates to each page are written by more than one processor. The pages containing the new and old arrays of unknowns are true-shared as the new values are used in the next iteration.

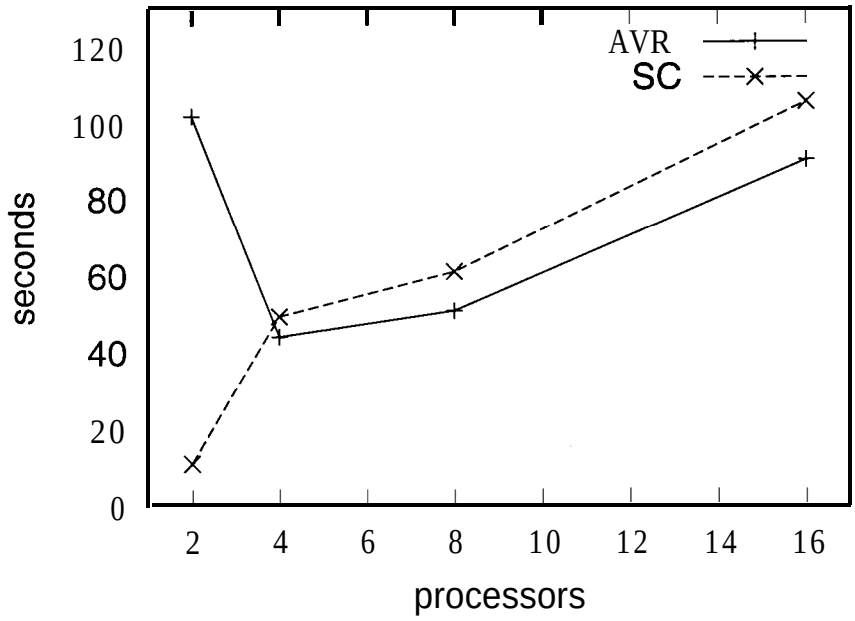

Figure 3.11: Gauss time vs. #processors

Gauss

Gauss-Seidel is similar to Jacobi, except that only one array of unknowns is used and it is both read and written during computation. As shown in Figure 3.11, the best time for Gauss was 11.35 seconds with two processors using SC. AVR took 101 seconds. AVR does not perform well in the two processor configuration because the checkpointing overhead exceeds the time required to send and receive a message from a remote host. The checkpoint overhead is large because

the working set is large. The time to send and receive a message is relatively small because copysets are small and message forwards do not occur. The overall effect is long resolution intervals, a decrease in the amount of computation per interval, an increase in the number of messages, and an increase in the number of queued messages. Read resolution intervals in AVR were 4959 microseconds in length. Read resolution intervals in SC were only 1245 microseconds. The number of messages sent in AVR was 39204, while the number of messages sent in SC was 7557. The number of messages in AVR is greater because the amount of computation per page fault is small. The number of queued messages in AVR was 9587, which indicates that the receipt of incoming messages was the cause of entry into the ending phase for 25% of resolution intervals. The effects of long resolution intervals, a decrease in amount of computation per interval, the increase in the number of messages, and the increase in the number of queued messages is cyclic. Long intervals result in the queuing of messages, which in turn results in the decrease of the length of the second phase of the resolution interval. Less computation per interval gives rise to additional messaging, which leads to extensive sigio interrupts and message queueing.

AVR performed better than SC for all processor counts greater than two. SC performance was lower due to extremely long resolution intervals and a high number of messages sent. In some cases SC resolution intervals were 2 times longer than those in AVR. AVR required 20000 fewer messages then SC. False-sharing occurs on pages which store the unknown array. AVR performs well because the overlapped communication and computation allows different processors to simultaneous use shared pages. Rollbacks totaled less than 1% of all checkpoints. The

results from Gauss exemplify the false-sharing problem that occurs with SC. SC performance is poor because the false-shared pages ping-pong between processors such that multiple page faults occur as read operations are made in the calculation of the new values of the unknown array. This does not occur in AVR, since the read operations occur optimistically and are validated upon receipt of the updated page. The net result for AVR is that, because the pages are false-shared, more operations complete per page fault than in SC. Hence, better performance of the AVR protocol. True-sharing occurs in Gauss in both the calculation of new array values and in the tolerance comparison phase which is protected by a lock. Although all pages which contain the array of unknowns are true-shared, only 1 page per processor is marked as a true-shared page. This is because accesses are staggered and data resolution has pre-fetching capability where values not used yet are updated to reflect operations which have been performed remotely. AVR uses regular SC to perform consistency resolution inside locks. Thus, AVR avoids the checkpoint and rollback performance penalty that would otherwise occur as a result of the true shared memory access.

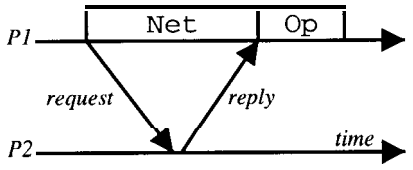

Figure 3.12: SC Resolution Interval

3.3.2 Discussion

The analysis of AVR would not be complete without a discussion of validity resolution interval properties. A resolution interval is the time it takes for invalid data to become valid. Figure 3.12 shows a resolution interval for sequential consistency. Figures 3.13 and 3.14 show resolution intervals for non-rolling back AVR and rolling-back AVR, respectively. In SC user processing is blocked while messages traverse the network and the remote node performs the computation necessary for the remote request. In SC, Figure 3.12, this interval is marked as Net. The *Op* segment, also shown in Figure 3.12, is the time required to perform the operations dependent on the data resolution.

The non-rollback AVR interval (Figure 3.13) includes three segments relevant to this discussion. The Cp segment is the amount of time required to save state. The Op segment is the same as the Op segment in SC. The *Res* segment is the resolution time required to undo the changes necessary to protect the process state.

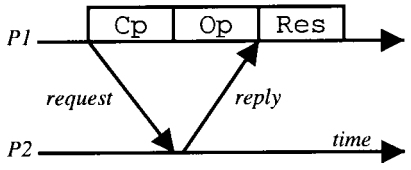

Figure 3.13: AVR Resolution Interval Without Rollback

Comparing the two intervals gives the relationship that in order to improve performance using AVR for resolution intervals that do not end in a rollback it must be true that Cp + Op + *Res* < Net + Op. Hence, *Cp* + Res < Net and Res < *Op*. The relationship indicates that the time used for checkpointing must be less than the time required for messaging and remote processing and the time required to resolve the checkpoint must be less than the the time spent performing user computation. The performance gain of AVR, the length of time g, can be described as g = *Net* + *Op* − *(Cp* + *Op* + *Res)* or g = *Op* − *Res.*

The rollback AVR interval (Figure 3.14) is different from the non-rollback interval because there are two Op segments, a *Rol* segment, and no *Res* segment. The Cp segment is the same as that shown in the non-rollback interval(Figure 3.13). The *Rol* segment is similar to the Res segment, but is longer because the system needs to rollback the process and copy twins over active pages. The first Op segment, Opl, is the length of time allowed for user processing. It is different than *Op2* because the actual amount of computation depends on the validity of the data and program flow. The second Op segment, *Op2*, is equivalent to the Op segments in the SC and non-rollback AVR intervals.

The rollback AVR interval can be approached in the same manner as the non-rollback interval with the relationship given as *Cp* + Opl + *Rol* + *Op* < *Net* + *Op*. Note the replacement of *Op2* with Op. Hence, *Cp* + Opl + *Rol* < *Net*, *Rol* + *Op* < Op, and *Rol* < 0 must hold for AVR to perform better than SC. Obviously, performance improvement over SC cannot occur in the AVR rollback interval. The performance loss in the AVR rollback interval, the length of time *l*, can be described as 1 = *Net* + *Op* − *(Cp* + Opl + *Rol* + *Op)* or *l* = −*Rol*.

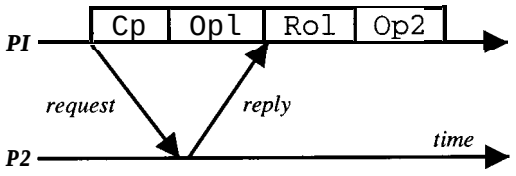

Figure 3.14: AVR Resolution Interval with Rollback

Given the two AVR interval types, performance (P) for a sequence of intervals relative to SC is $P = G+L$ where G $= \sum g$ and $L = \sum l$. Hence, $P = \sum g + \sum l$ and $P = \sum(Op-Res) - \sum(Rol)$. This analysis alone is not suitable for describing performance of different parallel programs, but is suitable for describing identical programs in the same environment.

Performance for AVR is varied. The test suite of programs can be separated into four different groups each of which provides a characterization of AVR. The first group is the set of programs for which there is no false-sharing or fragmentation. These are Water, Spatial, and LU. In these programs, *Res* is sufficiently small that *Res* < Op and g > 0 in invalidation intervals. Losses are minimized due to the ability of AVR to use SC for page faults on true-shared pages. These programs show that the optimistic approach to invalidation that AVR uses does improve performance.

The second group of programs are FFT and Ocean. These programs do not have false-sharing, but do have fragmentation. In these programs, *Cp* > *Net*. The time required to save state information is greater than the message turn-around

time. The programs have no false sharing and, therefore, have no opportunity for performance gains from read and write resolution intervals. These programs also have a high occurrence of rollbacks resulting in $|L| > |G|$.

The third group of programs are Raytrace and Volrend. The programs have false-sharing and fragmentation. In these programs, *Cp* + Res < Net and g > 0. A large amount of computation is performed relative to the length of time required for overhead. Performance loss due to AVR rollback intervals is small since fewer than 2% of resolution intervals terminate with a rollback.

The fourth group of programs includes Radix, Matmul, Jacobi, and Gauss. The programs exhibit a false-sharing pattern in which each processor writes to a global object. These programs have performance gains from invalidation intervals where Res < *Op*. These gains are particularly evident in Matmul because of the large number of write accessed pages. Unfortunately, each of these programs suffers when performing false-shared accesses because the accesses are computationally small relative to the overhead. Hence, Res > Op for false-shared accesses. The programs are characterized by large working sets which require extensive protection calls and twin-ing. The high overhead results in *Cp* + Res > Net. This, combined with poor false-shared access performance, outweighs the performance gains of invalidation intervals in Radix, Jacobi and Gauss. AVR performs well in Matmul using low processor counts, since Matmul has considerably more invalidation intervals and less false-shared accesses in those configurations. For this group of programs, the number of false-shared accesses and the length of Net increases with the number of processors, but *Res* gets smaller due to the decrease in working set size resulting in improved performance of AVR relative to SC.

The program suite is comprehensive in its memory use patterns. Experimental results show that good performance of the AVR protocol occurs when the proportion of false-sharing is high in the application, when the overhead cost of the checkpoint is low relative to the round-trip message cost, and when the amount of user processing is high relative to the round-trip message cost. The round-trip message cost includes the time to send a request, the time for the remote node to process the request, and the time for the reply to reach the original sender.

The proportion of false sharing relative to the amount of true sharing is indicative of AVR performance. Although, the cost to perform a checkpoint for false-shared and true-shared accesses can be the same, the cost to resolve the checkpoint is less expensive for false-sharing processes which do not need to rollback. The exact costs are problem dependent and are related primarily to the number of the working set of pages. Each rollback degrades performance by the cost to resolve the checkpoint. Programs which have a relatively low proportion of rollbacks will perform better than programs that have a high proportion of rollbacks.

The number of working-set of pages is the primary factor with regard to the amount of time required to checkpoint and resolve a checkpoint. Read-write cached working pages must be protected from spurious writes. Protection could involve as many system calls as there are pages to protect, depending on whether the pages exist contiguously in memory. Pages which are protected during the checkpoint will need to be unprotected at the end of the checkpoint. If the time required to checkpoint is high relative to the round-trip message cost, the amount of useful computation time is minimized and less work is accomplished per check-

point. If the time to resolve the checkpoint is high and approaches the user computation time, the benefit of optimism is lost. If both checkpoint and checkpoint resolution times are long, the loss of performance is compounded. If working sets are small, the opposite occurs. More user computational work can be performed and AVR achieves larger performance gains as the time to resolve the checkpoint decreases in proportion to the amount of user work. Overall, programs which have small working sets perform better than programs that have large working sets.

The amount of user processing relative to the round trip message time is directly related to the performance of asynchronous validity resolution. The relationship of user processing to the size of the working set is discussed above. There are other influences on the amount of user computation. They are the number of interrupts and the execution flow pattern. Interrupts diminish the amount of user processing by causing the operating system to preempt the user process to run the interrupt handler. Sigios are interrupts caused by incoming messages. Programs that require a large number of messages have a reduction in the amount of user computation per checkpoint. False sharing is a direct cause for increased messages since false sharing leads to the ping-pong effect. The flow of the user program execution determines the amount of user computation. Program flow is bounded by the locality of memory objects. A checkpointed process that attempts to access an inaccessible page must resolve its current checkpoint before performing any validity resolution actions on the new page. A process which has the task of writing a small number of bytes to a number of inaccessible pages may not have any performance gain from AVR unless the overhead of resolving the checkpointing is small. In this situation the amount of user work

is bounded and growth of working set size will determine overall performance. In general, programs that operate contiguously in memory perform better than programs that have scattered accesses and programs which have a high level computational complexity per memory access perform better than those which have low computational complexity per memory access.

3.3.3 Validation of Claims

The claims stated in Section 3 are validated by experimentation. The claims are reiterated here:

> **Claim 1.1** Asynchronous Validity Resolution (AVR) decreases the performance loss that is associated with false sharing in regular sequential consistency.

The experiments show that AVR does reduce the performance loss of false-sharing when overhead costs are low relative to the round-trip message time. AVR provides up to 26% better performance for some applications which exhibit false sharing. However, AVR does not provide faster execution than SC for false-shared applications that have a large number of true-shared accesses or a large number of working-set pages per execution thread. The overall performance trend, however, is that as the amount of false sharing increases in the system by the use of additional processors, so does the performance of AVR relative to SC.

> **Claim 1.2** Asynchronous Validity Resolution (AVR) does not require a different programming methodology than that of regular sequential consistency.

Claim 1.2 addresses the programming requirements necessary for consistency. AVR does not require any additional methodology for programming to ensure memory consistency. The implementation of AVR used for this dissertation is a user space implementation which requires the user to insert special commands around I/O functions. With additional work, not relevant to this thesis, AVR could be implemented in the operating system layers and would not require explicit programming of these instructions.

> **Claim 1.3** Asynchronous Validity Resolution (AVR) has best performance in loosely coupled systems that have relatively high communication costs.

AVR performs best in applications which are loosely coupled and have relatively high communication costs. Loosely coupled applications are those which have little true sharing or few synchronization operations such as locks and barriers. Locks and barriers indicate the presence of true sharing. Locks provide mutual exclusion and sequentialize memory accesses. AVR reverts to SC within locks, since use of optimism is unfounded in this context. Barriers are fence instructions used to separate memory accesses occurring prior to the barrier from memory accesses occurring after the barrier. The results of the experiments show that AVR performs well when a program has a greater amount of false sharing relative to true sharing. AVR does not perform well when true sharing dominates.

Experimentation shows that communication costs are important to the performance of AVR. Communications costs are the round-trip message costs which include the wire-time and the time for processing on the remote node. The ex-

perimental results show that if the round trip message cost is high relative to the protocol overhead costs, best performance of AVR is obtained.

Chapter 4

Conclusions and Future Work

4.1 Conclusions

This dissertation addressed barriers of performance in sequential consistency. Its goal was to develop a method to reduce the effects of false sharing in sequential consistency while maintaining ease of programming. This has been accomplished. This thesis presents and evaluates a new SVM protocol, Asynchronous Validity Resolution (AVR).

AVR was implemented in a modularized SVM system. AVR is based on regular SC and its use does not require additional programming complexity beyond the intuitive shared-memory programming methodology. AVR is capable of handling common synchronization mechanisms such as locks and barriers.

A modularized SVM system was used in this study as a control for experimentation with an existing SC protocol. AVR and SC were tested using a comprehensive suite of programs. Runtime statistics for AVR and SC were generated

and analyzed. The results show that there are four components that are directly related to the performance of AVR. They are the number of false-sharing vs. true-sharing accesses, the number of pages in the program's working set, the amount of user computation that completes per access, and the round-trip message time. In order for AVR to outperform SC, the average round-trip message time must be greater than the average amount of time required to checkpoint and perform user computation. Additionally, the average amount of time to perform user computation must be longer than the average amount of time required to resolve the checkpoint. Performance of AVR is dependent on the nature of the programs execution, since that is what determines the amount of true sharing and the amount of false sharing. True sharing incurs a performance penalty in AVR. False sharing is the basis for performance gains. AVR can only have good performance if the false-shared gains outweigh the true-shared losses.

Overall, AVR met the expectations and satisfied the claims of the dissertation. The results show that AVR could be an important member of the arsenal of tools available to parallel programmers.

4.2 Future Work

AVR is a potential foundation block for future research in areas such as protocol modification, overhead reduction, fault tolerance, and integration. Potential exploits in the protocol modifications of AVR are including multiple and simultaneous resolution actions. Simultaneous resolution events could result in the increase of the overall amount of asynchronous user computation by amortizing

the overhead costs. This protocol enhancement may also lead to prefetching effects which could reduce communication.

Practical overhead reduction is a possible avenue for work. It is expected to be worthwhile to move the implementation of AVR closer to the hardware. An operating system layer AVR protocol would be able to capitalize on lower overhead, kernel threading, access to all CPU registers, and use of multiple timers. This would minimize the protocol overhead component and increase the available time for user computation. Access to all CPU registers may allow for a more precise logging mechanism which would reduce the number of rollbacks.

Fault tolerance is a field of interest in SVM research. Checkpointing is a mechanism used to provide fault tolerance. Hence, fault tolerance would be a natural extension of AVR. Fault tolerance is the ability for an execution to survive loss of resources such as network connectivity. Adding fault tolerance to AVR would allow threads to migrate or be reborn on different hosts to allow the program to complete.

Another possible field of study is the integration of AVR concepts into programming tools such that compiler-generated code can be tooled to have constructs which signal how a particular access should be handled. A true-shared access might be handled using regular SC, while a false-shared access depending on its complexity, might be handled by AVR. Applications can be generated that tell the system how to handle the accesses, rather than the system choosing a solution that is not optimal.

The research potential of AVR and AVR concepts is great. Pursuit of any of the avenues listed will likely improve AVR performance while maintaining an

intuitive programming style that does not limit types of applications which can be used.

BIBLIOGRAPHY

[1] C. Amza, A.L. Cox, S. Dwarkadas, and W. Zwaenepoel. Software DSM Protocols that Adapt between Single Writer and Multiple Writer. In Proceedings of the 3rd IEEE Symposium on High-Performance Computer Architecture, 1997.

[2] S. V. Adve and K. Gharachorloo. Shared Memory Consistency Models: A Tutorial. Technical Report ECE 9512 DEC95/7, Rice University and Western Research Laboratory, September 1995.

[3] S.V. Adve and M.D. Hill. Weak Ordering-A New Definition. In Proceedings of the 17th Annual Symposium on Computer Architecture, pages 2-14, June 1990.

[4] M.A. Blumrich, R.D. Alpert, A. Bilas, Y. Chen, D.W. Clark, S. Damianakis, C. Dubnicki, E.W. Felten, L. Iftode, K. Li, M. Martonosi, and R.A. Shillner. Design Choices in the SHRIMP System: An Empirical Study. In Proceedings of the 25th Annual Symposium on Computer Architecture, June 1998.

[5] N. J. Boden, D. Cohen, R. E. Felderman, A. E. Kulawik, C. L. Seitz, J. N. Seizovic, and W.-K. Su. Myrinet: A Gigabitper-Second Local Area Network. IEEE Micro, 15(1):29-36, February 1995.

[6] L. Borrmann and M. Herdieckerhoff. A Coherency Model for Virtually Shared Memory. In 1990 International Conference on Parallel Processing, August 1990.

[7] A. Bilas, L. Iftode, D. Martin, and J.P. Singh. Shared Virtual Memory Across SMP Nodes Using Automatic Update: Protocols and Performance. Technical Report TR-517-96, Princeton, NJ, March 1996.

[8] A. Bilas, L. Iftode, R. Samanta, and J. P. Singh. Supporting a coherent shared address space across SMP nodes: An application-driven investigation. In IMA Volumes in Mathematics and its Applications, Volume 105, Algorithms for Parallel Processing, 1998.

[9] B.N. Bershad and M.J. Zekauskas. Midway: Shared Memory Parallel Programming with Entry Consistency for Distributed Memory Multiprocessors. Technical Report CMU-CS-91-170, Carnegie Mellon University, September 1991.

[10] J.B. Carter, J.K. Bennett, and W. Zwaenepoel. Implementation and Performance of Munin. In Proceedings of the Thirteenth Symposium on Operating Systems Principles, pages 152-164, October 1991.

[11] M. Dubois, J.C. Wang, L.A. Barroso, K. Lee, and Y-S Chen. Delayed Consistency and Its Effects on the Miss Rate of Parallel Programs. In Proceedings of Supercomputing '91, pages 197-206, 1991.

[12] M. J. Franklin, M. J Carey, and M. Livny. Transactional Client-Server Cache Consistency: Alternatives and Performance. In ACM Transactions on Database Systems, 22(3):315-363, September 1997.

[13] B. Fleisch and G. Popek. Mirage: a coherent distributed shared memory design. In Proceedings of the 12th Annual ACM Symposium on Operating System Principles, pages 211-223, December 1989.

[14] K. Gharachorloo, D. Lenoski, J. Laudon, P. Gibbons, A. Gupta, and J. Hennessy. Memory Consistency and Event Ordering in Scalable Shared Memory Multiprocessors. In Proceedings of the 17th Annual Symposium on Computer Architecture, pages 15-26, May 1990.

[15] W. Hu, W. Shi, and Z. Tang. Home Migration in Home-Based Software DSMs. In Proc. of the 1st Workshop on Software Distributed Shared Memory (WSDSM'99), June 1999.

[16] H. Han, C. Tseng, and P. Keleher. Eliminating Barrier Synchronization for Compiler-Parallelized Codes on Software DSMs. International Journal of Parallel Programming, 26(5):591-612, October 1998.

[17] W. Hu. Reducing Message Overhead in Home-Based Software DSMs. In Proc. of the 1st Workshop on Software Distributed Shared Memory (WSDSM'99), June 1999.

[18] Intel Architecture Software Developer's Manual. Intel Corporation. 1999.

[19] L. Iftode, C. Dubnicki, E. W. Felten, and Kai Li. Improving Release-Consistent Shared Virtual Memory using Automatic Update. In Proceedings of the 2nd IEEE Symposium on High-Performance Computer Architecture, February 1996.

[20] L. Iftode, Home-Based Shared Virtual Memory. PhD thesis, Princeton University, June 1998.

[21] A. Itzkovitz and A. Schuster. Distributed Shared Memory: Bridging the Granularity Gap. In Proc. of the 1st Workshop on Software Distributed Shared Memory (WSDSM'99), June 1999.

[22] L. Iftode, J. P. Singh, and Kai Li. Understanding Application Performance on Shared Virtual Memory. In Proceedings of the 23rd Annual Symposium on Computer Architecture, May 1996.

[23] L. Iftode, J.P. Singh, and K. Li. Scope Consistency: a Bridge Between Release Consistency and Entry Consistency. In Proceedings of the 8th Annual ACM Symposium on Parallel Algorithms and Architectures, June 1996.

[24] T. Jeremiassen and S. Eggers. Reducing False Sharing on Shared Memory Multiprocessors through Compile Time Data Transformations. Technical Report UW-CSE-94-09-05, Dept. of Computer Science and Engineering, University of Washington, 1994.

[25] P. Keleher, A.L. Cox, and W. Zwaenepoel. Lazy Consistency for Software Distributed Shared Memory. In Proceedings of the 19th Annual Symposium on Computer Architecture, pages 13-21, May 1992

[26] P. Keleher. Lazy Release Consistency for Distributed Shared Memory. PhD thesis, Rice University, January 1994.

[27] P. Keleher. The Relative Importance of Concurrent Writers and Weak Consistency Models. At the 16th International Conference on Distributed Computing Systems, May 28, 1996.

[28] P. Keleher. The CVM Manual. Available as Maryland Tech Report. 1997.

[29] L. Kontothanassis, G. Hunt, R. Stets, N. Hardavellas, M. Ciernak, S. Parthasarathy, W. Meira Jr., S. Dwarkadas, and M. Scott. Vm-based shared memory on low-latency, remote-memory-access networks. In Proceedings of the 24th Annual Symposium on Computer Architecture, 1997.

[30] G. Kiczales, J. lamping, A. Mendhekar, C. Maeda, C. Lopes, J.-M. Loingtier, and J. Irwin. Aspect-oriented programming. In Lecture notes in Computer Science, ECOOP'97- The 11th European Conference on Object-Oriented Porgramming. Springer, Vol 1241, pages 220-242, June 1997.

[31] 0. Kreiger and M Stumm. An Optomistic Algorithm for Consistent Replicated Shared Data. In Proc. of the Twenty-third Annual Hawaii International Conference on System Sciences, Vol 2, pages 367-75, June 1990.

[32] L. Lamport. How to Make a Multiprocessor Computer That Correctly Executes Multiprocessor Programs. IEEE Transactions on Computers, C28(9):690-691, 1979.

[33] T. Liang, D. Chuang, and C. Shieh. Thread Selection in Software DSM systems. In Proc. of the 1st Workshop on Software Distributed Shared Memory (WSDSM'99), June 1999.

[34] K. Li and P. Hudak. Memory Coherence in Shared Virtual Memory Systems. In Proceedings of the 5th Annual ACM Symposium on Principles of Distributed Computing, pages 229-239, August 1986.

[35] K. Li. Shared Virtual Memory on Loosely-coupled Multiprocessors. PhD thesis, Yale University, October 1986. Tech Report YALEU-RR-492.

[36] N. P. Manoj and R. Govindarajan. CAS-DSM: A Compiler Assisted Software DSM. In Proc. of the 1st Workshop on Software Distributed Shared Memory (WSDSM'99), June 1999.

[37] D. Mentre, D. Le Metayer, and T. Priol. Towards designing SVM coherence protocols using high-level specifications and aspect-oriented translations. In Proc. of the 1st Workshop on Software Distributed Shared Memory (WSDSM'99), June 1999.

[38] B. Nitzberg and V. Lo, Distributed Shared Memory: A Survey of Issues and Algorithms. In Computer, 24(8):52-60, August 1991.

[39] M.C. Ng and W.F. Wong. Adaptive Schemes for Home-based DSM Systems. In Proc. of the 1st Workshop on Software Distributed Shared Memory (WS-DSM'99), June 1999.

[40] M. Plakal., D. Sorin, A. Condon, and M. Hill. Lmaport Clocks: Verifying a Directory Cache-Coherence Protocol. In Proc. of the 10th ACM Symposium on Parallel Algorithms and Arichitectures, 1998.

[41] C. Rehn and M. Pizka. BOPS - Balancing Objects and Pages in a Shared Space. In Proc. of the 1st Workshop on Software Distributed Shared Memory (WSDSM'99), June 1999.

[42] E. Speight and J. Bennett. Using Multicast and Multithreading to Reduce Communication in Software DSM Systems. In Proceedings of the 4th IEEE Symposium on High-Performance Computer Architecture, 1998.

[43] I. Schoinas, B. Falsafi, A.R. Lebeck, S.K. Reinhardt, J.R. Larus, and D.A. Wood. Fine-grain Access for Distributed Shared Memory. In Proceedings of the 6th International Conference on Architectural Support for Programming Languages and Operating Systems, pages 297-306, October 1994.

[44] D. Scales and K. Gharachorloo. Towards Transparent and Efficient Software Distributed Shared Memory. In Proceedings of the Sixteenth Symposium on Operating Systems Principles, 1997.

[45] D.J. Scales, K. Gharachorloo, and C.A. Thekkath. Shasta: A Low Overhead, Software-Only Approach for Supporting Fine-Grain Shared Memory. In Pro-

ceedings of the 7th International Conference on Architectural Support for Programming Languages and Operating Systems, October 1996.

[46] T. Sterling, D. Becker, D. Savarese, J.E. Dorband, U.A. Ranawake, and C.V Packer. BEOWULF: A Parallel Workstation for Scientific Computation. In Proceedings of the International Conference on Parallel Processing. 1995.

[47] M. Stumm and S. Zhou. Algorithms Implementing Distributed Shared Memory. IEEE Computer, 23(5), May 1990, pages 54-64.

[48] K. Thitikamol and P. Keleher. Thread migration and communication minimization in DSM systems. In the Proceedings of the IEEE, March 1999.

[49] S. Woo, M. Ohara, E. Torrie, J. P. Singh, and A. Gupta. The SPLASH-2 Programs: Characterization and Methodological Considerations. In Proceedings of the 21st Annual International Symposium on Computer Architecture, June 1995.

[50] Y. Zhou, L. Iftode, and K. Li. Performance Evaluation of Two Home-Based Lazy Release Consistency Protocols for Shared Virtual Memory Systems. In Proceedings of the Operating Systems Design and Implementation Symposium, October 1996.

BIOGRAPHY OF THE AUTHOR

Jonathan Thomas was born in Middlebury, Vermont on November 29, 1965. He was raised in Underhill, Vermont and graduated from Mount Mansfield Union High School in 1983. He attended the Unversity of Vermont and graduated in 1995 with a Bachelor's of Science in Biological Science. He enrolled in graduate school at the University of Maine in 1996 and graduated with a Master's of Science in Computer Science in 1998. Jonathan began work toward a Ph.D. in Computer Science at the University of Maine in 1998 and was formally enrolled in 1999.

After receiving his degree, Jonathan will be beginning a career at IBM. Jonathan is a candidate for the Doctor of Philosophy degree in Computer Science from The University of Maine in May, 2001.