To appear in *Journal of Parallel and Distributed Computing*, Special Issue on Distributed Shared Memory Systems, Oct. 1995

Algorithms for dynamic software cache coherence

Harjinder S. Sandhu

Computer Systems Research Institute, University of Toronto

Abstract

In this paper, we investigate a class of cache coherence strategies in which an abstraction for shared data at the program-level, referred to as Shared Regions (SR), is used to manage caches dynamically through software. The practical value of these strategies is measured by their performance relative to existing hardware coherence protocols, and the complexity of the SR programming interface. We present detailed quantitative results highlighting the performance of a wide array of SR coherence algorithms, including some novel algorithms introduced in this paper that use direct cache-to-cache data transfers via software to improve performance. These algorithms are studied using execution-driven simulation and compared to a representative hardware strategy for a suite of parallel applications. We study the issue of programming complexity by analyzing the process of inserting Shared Regions program annotations into these applications. The experimental results show that the best SR coherence strategy for each application is comparable to or significantly better than the representative hardware strategy for all of the applications that we examine. Our study of programming complexity in SR finds that, for the types of applications that we study, inserting SR annotations is a relatively simple and methodical task. Decisions about the insertion of these annotations can be made based solely upon local information without regard to the global behaviour of the program.

1.0 Introduction

Shared-memory multiprocessors offer a simple and convenient abstraction for programming parallel applications, that of a globally addressable shared address space. Dealing effectively with memory access latency is one of the key issues in the design of these architectures. Processor caches offer a way to significantly reduce this latency but also give rise to the problem of cache coherence. To date, a large variety of hardware and software cache coherence strategies have been proposed that make various trade-offs in cost, complexity, efficiency, and a host of other factors. Hardware strategies are often considered complex, expensive, and inflexible, while traditional software schemes relying on static program analysis are usually inefficient because of limited compile-time knowledge of a program's run-time behaviour.

In this paper, we investigate a class of strategies in which caches are managed dynamically through software, using an abstraction for shared data at the program level referred to as Shared Regions (SR). Shared regions, the unit of sharing in the program, are identified by the application through a set of annotations that also mark all series of references to those regions. Similar annotation models have been proposed for managing shared data in distributed shared-memory (DSM) environments [4][10], for exploiting locality through main memory in scalable shared-memory multiprocessors [11][8], and for improving the performance of hardware cache coherence protocols [14]. In this work, we focus exclusively on the use of this model for managing caches entirely through software in shared-memory architectures.

Relative to other forms of cache management, the Shared Regions approach offers the ability to manage data dynamically at the granularity at which it is shared within the application, but at the cost of requiring specific annotations in the program. Then, the practicality of coherence enforcement strategies based upon the Shared Regions approach must be measured both by the performance of these strategies relative to other hardware and software coherence protocols, and the complexity of programming with the SR program annotations. An earlier implementation and measurement study of Shared Regions on the Hector multiprocessor shows that this approach can be implemented entirely in software without any hardware support, and that it offers an efficient alternative to other forms of software cache management for shared-memory architectures without hardware cache coherence [22]. An analytic study of several SR algorithms and a representative hardware strategy also shows that, for coarse-grained applications at least, this approach may offer an efficient alternative for coherence enforcement even on architectures that already provide coherence in hardware [23]. In this paper, we build upon this earlier work in a number of ways. First, we present detailed quantitative results highlighting the performance differences of a wide array of SR coherence algorithms, including some novel algorithms introduced in this paper. This performance evaluation is conducted under various architectural assumptions and relative to hardware coherence protocols for a suite of coarse-grained parallel applications. Second, we study the issue of programming complexity in SR, by analyzing the process of inserting SR annotations into these applications.

One of the primary advantages of software implementations of cache coherence is flexibility. That is, software coherence enforcement strategies can be easily adapted to deal with specific types of sharing patterns for a particular application, or even to deal with different types of sharing patterns within the same application. One of the goals of this paper is to explore the range of alternatives that are available for cache management based upon the Shared Regions abstraction. We begin by presenting a family of algorithms for software cache coherence that rely on selectively flushing data to memory. In these *flush-based* algorithms, the scope of possible algorithms is defined by whether invalidations and writebacks are performed pessimistically, optimistically, or (in the case of invalidations) in a lazy fashion. Next, we introduce a set of algorithms that go beyond the limitations of traditional software coherence strategies by using direct cache-to-cache data transfer (via software) to improve coherence enforcement. These SR *forwarding* algorithms can potentially reduce coherence overhead and increase cache hit rates by forwarding regions directly to the caches of the processors needing them.

A detailed execution-driven simulator and six coarse-grained numerical applications are used to quantify the performance of the dynamic software coherence strategies presented in this paper. With respect to the flush-based coherence alternatives, the results of this simulation study show that pessimistic invalidation strategies perform poorly in general, that the choice of optimistic or lazy invalidation is largely irrelevant to performance, and that the strategies employing pessimistic writeback perform better than those using optimistic writeback in applications where the number of blocks flushed in the pessimistic and optimistic writeback strategies are comparable. An examination of the SR forwarding schemes highlights the impact of cache size versus region size when using forwarding. The use of forwarding in SR shows significant improvements in performance for some applications, but only when the cache sizes are large relative to the size of the regions being forwarded. Given the trends in the design of shared-memory multiprocessors -- caches continue to become larger and memory latency continues to increase -- the use of forwarding through software will likely become more feasible, and more important, in future architectures.

The results of this simulation study also show that the performance of the best SR strategy for each application is comparable to or significantly better than that of a representative hardware coherence strategy for all of the applications that we study. This result is not entirely surprising considering the coarse-grained nature of the applications that we study, and given the results of an earlier analytic study that shows that coherence strategies based upon the Shared Regions approach tend to outperform hardware protocols when managing data with coarse-grained sharing patterns [23]. However, for scalable shared-memory architectures, programs with coarse-grained sharing patterns represent an important class of applications. Our analysis of the process of programming with SR in these applications finds that while considerable effort must often be expended in understanding the intrinsic sharing patterns of an existing application, inserting SR annotations into the program is a comparatively simple and methodical task. Unlike message-passing primitives, the SR annotations can be inserted into the program based only on local information and without regard to the global behaviour of the program. This suggests that, in the future, compilers may be able to insert these SR annotations automatically for programs in which the sets of shared regions are identified by the user.

The organization of this paper is as follows. Section 2 presents the Shared Regions model and also discusses related work based upon similar models. Section 3 presents algorithms for dynamic software cache coherence algorithms based upon the SR annotations. Section 4 describes the applications used in this study and how SR annotations are inserted into these applications. Section 5 discusses the experimental framework used to obtain quantitative results. Section 6 presents experimental results for the SR algorithms presented in this paper. Section 7 presents the conclusions of this work.

2.0 Shared Regions

2.1 Model

The Shared Regions (SR) model introduces an abstraction at the program-level that explicitly relates synchronization tokens with the data they protect. The term ***shared region*** denotes a set of memory locations that are bound to a specific synchronization token. Regions are created through a function called *bindregion* which is used to establish the association between these memory locations and a synchronization token, as in:

Region *R* = **bindregion**(*addr, size*)

In theory, the set of memory locations bound to a particular region can occupy an arbitrary set of addresses throughout memory. In practice, however, specifying and efficiently managing complex regions can be a difficult task and, for most programs, simpler region definitions with contiguous sets of memory locations specified by (*addr,size*), or blocks of data specified by (*addr,size,stepsize,nsteps*), will suffice.

Accesses to memory locations bound to a region are guarded by region primitives that may provide both synchronization and coherence. These primitives are **readaccess(R), readdone(R), writeaccess(R),** and **writedone(R).** In general, any series of accesses to memory locations bound to a particular region must be preceded by a *readaccess* or *writeaccess* operation to that region, and followed by a *readdone* or *writedone* operation. In many cases, these access/done primitives will replace other synchronization operations in the program. In other circumstances they will not, and the synchronization component of the access/done primitives may be disabled from the application.

2.2 Related Work

The notion of binding data to synchronization variables was first introduced as a method for structured concurrent programming by Hoare [15] and Brinch-Hansen [13]. In this approach, called Conditional

Critical Regions (CCR's), the variables used as monitors are explicitly bound to the shared variables that they protect, enhancing program clarity by making the effect of entering the critical section more obvious, but also permitting the compiler to perform automatic program verification by comparing the conditions for entering a monitor with the variables accessed within it. More recently, constructs for linking data and synchronization were also proposed in the Jade programming language as a means for permitting the compiler to detect concurrency in the program [18]. Conceptually, however, the Shared Regions program model can be viewed as a logical extension of existing memory consistency models such as *weak consistency* (WC) [2] and *release consistency* (RC) [12], and supporting a memory consistency framework that has been defined as *entry consistency* [4]. Unlike release consistency, in which coherence enforcement is performed at the invocation of a *release* operation, entry consistency permits coherence enforcement to be delayed until the corresponding *acquire* operation is issued.

The use of program annotations for optimizing data management has become common in distributed shared-memory (DSM) environments. The Munin system [5] used the RC conventions along with simple annotations that describe the sharing patterns for a specific data structure to exploit release consistency and permit different coherence enforcement protocols for different types of data. Treadmarks [16] builds upon this work by supporting lazy consistency, similar to entry consistency, for programs that obey the programming conventions of RC. Feeley *et al* [10] describe an approach, called Versioned Objects, in which annotations similar to those in SR are used to eliminate false sharing among pages, while the Midway system used these same annotations to allow the system to exploit multiple consistency models including entry consistency [4]. Other work has focussed on exploiting locality among the distributed memory units of hardware distributed shared-memory multiprocessors. For instance, in Romm [11], replication of shared regions among memory units was used to reduce memory access latency. The Hybrid Hardware/Software protocol presented by Chandra *et al* [8] also uses shared regions annotations and bulk data transfers for placing regions in the memory unit of a processor that is to access it next. In the CheckIn/CheckOut model [14], similar annotations are used as hints to the underlying system for optimizing the performance of a simple hardware coherence protocol.

3.0 SR Algorithms

Software cache coherence strategies typically enforce coherence through memory, using software initiated instructions that write back or invalidate data selectively from the cache. In SR, coherence operations are initiated through software at the access/done points in the program, using three basic instructions as the basic tools: (1) *inval(i,R)* - invalidate data associated with region R in cache *i,* (2) *wback(i,R)* - write back dirty data associated with region R from cache *i* to memory, and (3) *flush(i,R)* - invalidate and write back dirty data associated with region R from cache *i* to memory. When the first parameter is absent in each of these operations, the processor operates on its own cache, and this is referred to as a *local coherence operation*. A *remote coherence operation* is one in which one processor initiates a coherence operation on another processor's cache. The second parameter selects specific data ranges in the cache but is limited to operating at the granularity of a set of cache lines. The implications of this are discussed in the performance evaluation of section 6. We describe next a family of SR coherence algorithms that can be implemented using these operations. Section 3.2 describes another class of algorithms that use direct cache-to-cache data transfers to improve the way in which coherence is enforced.

3.1 Flush-based algorithms

The range of alternatives for cache coherence in SR can be described and enumerated based upon the interaction between two processors, P_i and P_j, with respect to a specific region R. Consider the following sequence:

$$P_i\text{:}done(R) \rightarrow P_j\text{:}\{writeaccess(R) \rightarrow writedone(R)\} \rightarrow P_i\text{:}access(R)$$

Here, the done(R) at P_i implies either writedone(R) or readdone(R), and access(R) implies either readaccess(R) or writeaccess(R). The done(R) operation indicates that P_i has referenced the region, and that some portions of the region therefore currently reside in P_i's cache. The writeaccess/writedone operations at P_j indicate a modification to some or all of the region, and the access(R) at P_i indicates that P_i is about to referenced the region again.

To correctly resolve the coherence conflict in this sequence, two conditions must be satisfied prior to the completion of the *access(R)* at P_i: (1) invalidate R in P_i, and (2) write back R from P_j to memory. Each of these conditions can be performed at several different points. The invalidation of R in P_i can be performed pessimistically when the P_i*:done(R)* is invoked, optimistically at P_j*:{writeaccess(R)* $\rightarrow$ *writedone(R)}*, or in a lazy fashion at P_i*:access(R)*. The writeback of R in P_j can be performed pessimistically at P_j*:writedone(R)* or optimistically at P_i*:access(R)*. The combination of these alternatives for invalidation and writeback gives rise to a range of SR algorithms, labelled *SRp*, *SRlp*, *SRlo*, *SRop*, and *SRoo*. Of the two letters following *SR*, the first indicates the invalidation strategy and the second the writeback strategy. They are chosen from the set {*p*=pessimistic, *o*=optimistic, *l*=lazy, *=any}. The * is useful for describing a group of strategies. For example, *SR*p* denotes all strategies using pessimistic writeback, and *SRl** denotes all strategies using lazy invalidation.

The modelling of these algorithms in the simulation study presented in this paper is an extension of the implementation of two of these algorithms, *SRp* and *SRlp,* on the Hector multiprocessor [22]. *SRp* makes coherence decisions statically and therefore requires no run-time state information. The implementation of *SRl** and *SRo** uses a region status vector **s** associated with each region in the system, having one entry for each cache in the system. Entry $\mathbf{s}_i$ describes the state of cache i with respect to that region and has three possible states: (1) *Not Have* (*NH*) - P_i does not have a copy of this region, (2) *Have Valid* (*HV*) - P_i has a valid copy of this region, and (3) *Have Invalid* (*HI*) - P_i has an invalid copy of this region. For *SR*o*, a variable d of size $\log_2 P$ (for a P processor system) is used to keep track of which cache, if any, has a dirty copy of the region. Each of these data structures is stored in memory and remains uncacheable in order to ensure that it remain consistent. The algorithms are described in Table 1.

Algorithms employing only pessimistic or lazy operations have the advantage that only local coherence operations are required. For the remote coherence operations, a processor requires the ability to invalidate or writeback a region from another processor's cache. To do this, each processor must have an agent that performs local coherence operations (on its own cache) on behalf of another processor. For remote operations, a processor sends a message to another processor and waits for a response. The remote processor is interrupted, carries out the coherence operation, and then responds to the original processor indicating that the coherence operation has completed.

The use of a fullmap region status vector, allocated through software but stored in memory, gives rise to two potential problems. First, the fact that this vector is stored in memory may present a performance bottleneck. However, for coarse-grained applications, the overhead of managing the region status vector usu-

Processor *i*	***SRp***	***SRl****	***SRo****	***SR*p***	***SR*o***
initialization		**for** k=0..P $\mathbf{s}_k = NH$	**for** k=0..P $\mathbf{s}_k = NH$		d = -1 (not dirty)
readaccess(R) writeaccess(R)		if $\mathbf{s}_i$=HI *inval(R)* $\mathbf{s}_i = HV$	$\mathbf{s}_i$ = HV		if d>=0 and d!=i *wback*(P_d,R) d = -1
readdone(R)	*inval(R)*				
writedone(R)	*flush(R)*	forall $\mathbf{s}_j = HV$ $\mathbf{s}_j = HI$	forall $\mathbf{s}_j = HV$ *inval*(P_j,R) $\mathbf{s}_j$ = NH	*wback(R)*	$d = i$

Table 1: Implementation of the flush-based SR algorithms.

ally amortized by the coarseness of the computation, as shown in the experimental results in section 6 and through implementation and measurement on Hector. The second problem has to do with the memory space overhead of the fullmap status vector. Although fullmap vectors are the simplest and possibly the most efficient structures for maintaining state information in coherence protocols, most hardware protocols avoid these because of their space overhead. However, unlike hardware schemes that associate state information with each cache line-sized block in memory, in SR there is a single status vector for each shared region. Given that our primary focus is on applications with coarse-grained sharing, the space overhead of the status vector too will likely be amortized by the coarse-granularity at which regions are defined.

3.2 Region Forwarding

Traditional software coherence strategies invariably enforce coherence through memory, by selectively flushing data from the cache. However, enforcing coherence through memory has an inherent disadvantage due to the latency of fetching data from memory and, in contrast, many hardware coherence strategies exploit direct cache-to-cache communication, using a technique referred to as forwarding, to improve performance. In this section, we consider the use of forwarding in the SR coherence strategies. We begin by describing extensions to the flush-based algorithms that use region forwarding. Then, we describe how this forwarding may implemented in software using a simple extension to the hardware.

In hardware coherence protocols, blocks may be forwarded directly from one cache to another whenever a processor performs a read or write operation to a block that currently resides in a dirty state in another cache. In SR, this same principle can be extended to forward regions between caches. We define a shared-dirty region as a region that resides in a dirty state in a remote cache. Forwarding in SR can be used on a readaccess or writeaccess operation to a shared-dirty region, by transferring the region from the remote cache (also referred to as the *source*) to the cache of the processor performing the access operation (also referred to as the *target* cache). However, the behaviour of forwarding depends on whether a readaccess or writeaccess operation is issued, as follows:

(a) On a writeaccess to a shared-dirty region, the region is forwarded directly from the source cache to the target cache. The forwarded blocks enter the target cache in a dirty state, and are set to an invalid state in the source. In the region status vector, ownership of the region passes from the source to the target.

(b) On a readaccess to a shared-dirty region, the region is forwarded from the source to the target but also written back to memory from the source, since subsequent readaccess operations by other processors will expect to find a copy of that region in memory. The forwarded blocks enter the target in a clean state, and are left in the source in a clean state.

Extensions to this basic forwarding algorithm can be made to optimize it for specific types of sharing patterns. For example, for regions in which there is one reader and one writer, the algorithm can be adapted so that the source does not write the region back to memory on a readaccess to a shared-dirty region. Instead, the source forwards the region but retains a dirty copy in its own cache. This reduces unnecessary copying of the region back to memory. The region now also enters the target in a dirty state, so that an ejection and subsequent access of one of these forwarded blocks at the target finds valid data in memory. For single-writer-multiple-reader regions, we can extend this algorithm still further so that the source retains ownership of the region after forwarding on a readaccess operation, so that subsequent readaccess operations by other processors will also send forwarding requests to this source. This extension, however, gives rise to a trade-off between burdening the source with extra work in serving multiple forwarding requests and the potential benefits of forwarding to the multiple targets.

In the SR forwarding algorithms, both the source and target processors will be stalled while the forwarding takes place, just as both are assumed to stall while a remote writeback takes place in *SR*o*. When regions are only forwarded to another cache (as on writeaccess operations to dirty regions), forwarded blocks can be pipelined through the interconnection and there is no need for a response from the target on a per-block basis. Instead, as soon as all of the blocks have been sent from the source, the source sends a response back to the target indicating that the region has been forwarded and also indicating the number of blocks that have been forwarded. When regions are both forwarded and written back to memory, the source must send a copy of each block to memory, but it can simultaneously forward the same block to the target processor. The source cannot send a message indicating the completion of the forwarding request until all flushes are completed at memory.

The use of forwarding in SR is conceptually similar to prefetching, with blocks arriving in the cache prior to being needed. However, unlike prefetching, the request for those blocks is transparent to the hardware. In our simulation of the forwarding algorithms, forwarded blocks are sent and received by a software protocol handler. At the source, the protocol handler steps through each line of the region and, if that line is in the cache, forwards it to the target processor. At the target, the blocks are received by a software handler and placed into the local cache one block at a time. Since the target is blocked waiting for lines to arrive, no context switching is required to receive and handle forwarded blocks in software. However, placing blocks into the cache via software may present a problem. Most contemporary architectures support the ability to test for specific lines in the cache from software, and to selectively read, write back, or invalidate those lines from the cache through software. To place forwarded lines into the local cache from software, we assume additional cache support via a version of the store operation, a *store-in-cache* operation, that allows a store to complete in cache even if the line to be stored is not currently in the cache. Such a store-in-cache operation is useful for efficiently initializing data that does not currently reside in the cache, but it can also be used to permit regions to be forwarded through software.

Forwarding in the SR strategies has an obvious potential for performance improvement, by allowing caches to transfer regions directly and avoid the latency of accessing memory. However, forwarding large regions from one cache to another also has the potential to degrade performance by introducing conflicts between forwarded regions and data that currently resides in the target cache. This potential for performance improvement or degradation using forwarding is studied in the experiments of section 6.4.

4.0 Programming with Shared Regions

Six parallel applications are used in the experimental evaluation presented in this paper. Of critical importance, in addition to the performance of these applications using the SR coherence strategies, is the complexity of programming with SR in these applications. In this section, we discuss how SR annotations are used in these applications. We begin by describing the six applications.

4.1 Application characteristics

For each of the applications used in this study, both an SR version and a non-SR version is written. In the SR versions, access/done operations replace other synchronization primitives whenever possible. However, in cases where synchronization through the SR access/done operations is redundant, the synchronization component of these operations is turned off through user control in the application. A summary of the applications is given in Table 2.

	Problem size	Run-time (cycles)	Memory refs (loads/stores)	Region ops (access/done)
FFT	16,000 element array	6983829	1431876	41
LU	200x200 matrix	19007933	8079408	1760
MVA	184x84 grid of vectors	18214766	10511280	15976
SPX	427x1113 tableau	23728387	9651960	344
NRL	351 units/layer, 4 layers	12977060	1876410	216
CG	900x900 sparse matrix	140216866	16043028	160

Table 2: Application run-time characteristics on an 8 processor system.

Fast Fourier Transform (FFT) performs both the forward and reverse segments of a 2-dimensional fast-fourier transform. It's primary data structure is an array of complex numbers partitioned into sections, with each section assigned to a separate processor initially. **LU Decomposition (LU)** performs a row-oriented decomposition of a matrix *A* into an upper and a lower triangular component. Each processor is statically assigned a set of rows to decompose. **Mean Value Analysis (MVA)** calculates average customer response times and queue lengths for a 2-class closed queueing network. It's primary data structure is a two-dimensional grid of queue-length vectors, with a set of vertical rows of vectors assigned to each processor. **Simplex (SPX)** implements the simplex optimization algorithm for solving linear programming problems. Each processor operates on a separate area of a tableau, and sharing occurs about the pivotal column of the tableau in each phase of the computation. **Neural Networks (NRL)** implements the back-propagation learning algorithm. There are six major shared data structures, matrices of one, two, and three dimensions, and sharing occurs in a different fashion for each data structure. **Conjugate Gradient (CG)** uses the conjugate gradient algorithm on a sparse matrix containing a set of partial differential equations. The sparse matrix is first re-organized as a set of submatrices, some of which are dense and some of which are sparse. The matrix is partitioned in such a way that each processor will operate only on a specific portion. Output vectors containing partial results are actively shared for read and write.

4.2 Programming with SR

In programming with SR, there are two separate tasks that are referred to as *region partitioning* and *region annotation*. Region partitioning has to do with identifying and defining the sets of shared regions in the program. Region annotation is the process inserting the SR access/done operations into the program.

In order to perform region partitioning, one must understand the way in which shared data is inherently *partitioned* in the program; that is, how it is grouped together and accessed by different processors. The region partitioning should match the intrinsic data partitioning of the program. In LU, for instance, sharing occurs along row boundaries, so regions are defined as rows of the matrix. Similarly, in MVA, sharing occurs along vector boundaries in a two dimensional grid of vectors, so each vector is defined as a region. In FFT, sharing occurs along subsections of a large vector, so each subsection is defined as a region. In NRL, and CG, each processor operates on portions of more complex data structures, and each subsection is defined as a region. In most cases, understanding the intrinsic data partitioning of an existing program requires a significant amount of work. Once this is done, performing region partitioning is typically easy. Usually, explicitly identifying the regions in the program also adds to its clarity, since the intrinsic partitioning is otherwise only implicitly defined in the program.

In the region annotation phase, the first step is to identify all of the intrinsic synchronization in the program, as well as where each region is accessed. If there is a one-to-one correspondence between the regions accessed and the synchronization variables used to guard them, then the existing synchronization operations can be replaced by SR access/done operations. This approach works in LU, and partially in FFT, but for the other four applications, it is not sufficient and a different approach must be taken. This by itself is interesting in that, while approaches based upon concepts such as Shared Regions usually imply a *link* between synchronization constructs and data, our study of these six applications finds that, more often than not, the SR access/done primitives do not serve as an adequate substitute for a program's inherent synchronization. In our study, the intrinsic synchronization was left intact for the other four programs. Here, SR access/done operations are inserted but with the synchronizing component of these operations disabled so that, technically, synchronization and data are not linked in these applications.

In inserting region annotations in the other four applications (MVA, SPX, NRL, and CG), we experimented with different approaches that seemed most natural to the application. These were: (1) barrier-based, (2) procedure-based, and (3) method-based. In the barrier-based approach, barriers are used for synchronization, and all of the regions accessed within two barriers points are enclosed in access/done operations that are placed next to the barriers. This approach is used in MVA and SPX, and works well enough if the progression of the program from one synchronization point to the next is readily apparent. Often however, the behaviour of the program is difficult to determine statically, as was the case in NRL. Here, a procedure-based approach is used. At the beginning of each procedure accessing shared regions, access operations are inserted for all regions accessed within that procedure, and corresponding done operations are placed at the end of the procedure. In CG, all of the shared data structures are encapsulated within C++ classes, so access/done operations are inserted within each method used to access that data. For both NRL and CG, the overall sharing patterns of the applications are rather complex, but region annotation is relatively simple because the insertion of these operations is highly localized. That is, one considers only the regions accessed in a particular procedure or method without regard for what is happening in the rest of the program. On the surface, it may appear that this would generate significant runtime overhead in constantly checking the validity of these regions, especially since the region status vector is always in memory. However, the coarse granularity of the applications, apparent in the number of region operations in Table 2, limits this overhead. This is also shown in the results presented in section 6.

For data structures in which the partitioning of regions changes dynamically (linked lists for example), the region definitions too must change dynamically, and this can be a complicated process. We have experimented with two such applications, a merge sort program and a program using stacks implemented as a linked list. In mergesort, an *unbindregion* operation is used that first flushes a region from all caches with copies of that region, and then permits the region to be redefined. In the stack program, each node of the linked list is defined as a region, so that regions are created and deleted on demand just like the nodes of the linked list. Neither of these programs obtain much speedup from multiple processors (less than 2x on 8 processors even with no coherence overhead) so results for them are not presented. In general, however, programs with such dynamic or complex structures may fare better with an approach which permits multiple consistency models. In our implementation of Shared Regions on the Hector multiprocessor, data that is too fine-grained or too difficult to program using SR annotations may be allocated on separate pages for which coherence is enforced by the operating system. Similarly, the Hybrid-Hardware/Software approach [8] uses the Shared Regions annotations only for managing coarse-grained data, leaving fine-grained data to be managed by the hardware coherence protocol, and Midway [4] permits the programmer to mix various consistency models, including release consistency and entry consistency, in the same application.

In general, for the applications used in our experimental study, programming with SR requires only a small effort beyond that of programming on a plain shared-memory interface. Placing SR annotations into existing programs may require a significant amount of work in understanding the intrinsic sharing patterns of the program, but the effort in adding the SR annotations once these sharing patterns are understood is considerably less. Unlike message passing, which requires understanding the *global* behaviour of the program, SR annotations can be placed into a program based entirely on local information, for instance by isolating one procedure and considering only the set of memory locations referenced in that procedure. Because regions annotations can be inserted based solely on local information, it is likely that compilers may, in the future, be able to insert these annotations into the program automatically once the set of shared regions have been identified by the user. In [24], we also consider another alternative for avoiding the programming overhead of SR, using architecture-level mechanisms that automatically detecting references to shared regions and thereby avoid the need for access/done annotations altogether.

5.0 Simulation model

A detailed execution-driven multiprocessor simulator is used in the experimental analysis presented in this paper. Applications execute on a base machine, the machine on which the simulator is running, and interact with the simulator only when memory operations (including synchronization and coherence operations) are encountered. For these, the type of operation, the address, and the approximate number of cycles elapsed since the last memory operation are provided as parameters to the simulator. The simulator in turn models memory operations in detail, including the effects of contention, and also correctly sequences the memory operations from each process in the application.

5.1 Simulated Architecture

The modelled architecture is patterned after the Hector multiprocessor [22], but cost/service ratios reflect those expected for Hector's successor, the NUMAchine multiprocessor [26]. Processors have a single-level direct-mapped cache, the size of which is varied between 64 Kbytes and 1 Mbyte in the experiments. The cache block size, also variable, is 16 bytes (as in Hector) by default. The simulated architecture is composed of memory/processor pairs, with each memory unit being globally addressable and containing a large priorityless buffer for incoming requests. Using expected latencies for NUMAchine, we model an interconnection and memory clock speed that is one-fourth that of the processor clock speed, and base service times at memory that are 7 memory cycles (28 processor cycles) for normal requests, and 2 memory cycles for hardware directory related operations.

The interconnection is hierarchically arranged, with buses at the lowest level and a hierarchy of rings at the next two levels. Each of the resources in the interconnection has a large priorityless buffer, so that transactions are never dropped in transit, and the interconnection bandwidth is four words, equivalent to the default cache block size. Each ring or bus component adds just one interconnection cycle to the time required for a request to traverse the interconnection. Both the interconnection and memory are modelled on a cycle-by-cycle basis.

The architecture is modelled in two ways, with sequential consistency (SC) and no store buffers, and with weak consistency (WC) and store buffers. In the SC version, a processor issues a memory operation and then stalls until its completion. In the WC version, store operations are placed in store buffers and the processor continues executing until either the store buffer is filled or a load or synchronization operation is issued. Load operations may proceed concurrently with outstanding stores, but synchronization operations stall until all outstanding memory operations are completed.

Synchronization operations are modelled in detail. The simulator implements low-level synchronization mechanisms such as *testandset* and uses these to build higher level synchronization mechanisms such as semaphores, barriers, and the SR access/done primitives. When blocking is necessary, all of the applications use queueing rather than spin-waiting. A number of other instructions provide some basic memory management tools, allowing the applications to lay out data structures precisely but statically in the simulated system's memory. For all of the applications, data is allocated so as to be localized and uniformly distributed across all memory units whenever possible.

5.2 Simulation of Coherence Strategies

For the SR strategies, the simulator models the basic software coherence instructions discussed in section 3 in detail. Their behaviour and cost is patterned after an implementation of Shared Regions on the Hector multiprocessor. Each coherence instruction incurs a fixed system call overhead (40 cycles), additional cost for checking/updating the region status vector in memory, and additional cycles for each block that is to be operated on. For flushing regions from the cache, each flushed block requires an acknowledgment from memory in order to ensure that a *done* operation on the corresponding region is not issued until all blocks have been successfully flushed. When store buffers are used in the architecture, flushes to multiple blocks are place in the store buffers and proceed concurrently. In the absence of store buffers, blocks are flushed serially.

For remote coherence operations, a message is sent to a remote processor, and there is a fixed cost for interrupting and context switching at the remote processor. Experiments with variations in the cost of context switching, from 4 cycles to 50 cycles, show little difference in performance due to the coarse-granularity of the applications. However, the behaviour of the two processors in coordinating the coherence operation represents an important consideration. On a remote coherence operation, the processor that initiated the operation stalls until it receives a reply from the remote processor. The remote processor performs the coherence operation as if it were local, and sends a reply back to the original processor when it is complete. For region forwarding, there is a four cycle cost at the target for placing a received block into the cache, but no context switching since the target is stalled waiting for the blocks to arrive.

The fullmap directory scheme is implemented and used for comparison. Although there are a large number of variants of this basic directory strategy, most of these are concerned with reducing its memory overhead (for example, the LimitLESS directory strategy [7]). In terms of performance, the fullmap scheme remains characteristic of the general class of directory strategies. In the fullmap scheme, each block in memory has a corresponding bit vector and a dirty bit. The bit vector contains a one bit entry for every cache in the system. The directory scheme is modelled in detail, both with and without forwarding. On a cache miss, the directory in memory is consulted (at no cost) and, if other processors have clean copies of the block, they are sent invalidation messages from the memory controller. If another processor has a dirty copy of the block, a writeback or forward request is sent to that processor. In the forwarding version, invalidation acknowledgments are also forwarded directly to the processor performing the write. In the non-forwarding version, invalidations are processed by the memory controller at a cost of 2 memory cycles. In the version of the directory scheme modelled in the simulator, ejections of clean blocks do not notify memory, and store operations to clean blocks are treated as cache misses.

6.0 Performance evaluation

Experimental results for the SR coherence strategies are presented in this section. We first compare the SR flush-based algorithms to each other, and then compare the hardware strategy to these flush-based algorithms. Then we consider the impact of some key architectural variations on the performance of these strategies. The last part of this section studies the performance of forwarding in SR.

Normalized execution time is used as the primary performance metric. Execution times are broken up into five components that are averaged over the set of processors. These components are: (1) *computation time* - the amount of time a processor spends performing computation, excluding computation within the SR access/done operations, (2) *coherence overhead* - the amount of time spent performing coherence related operations such as *writeback* or *invalidate*, or stalled waiting for remote coherence operations to complete, (3) *region overhead* - includes the basic system call overhead of SR, and the amount of time spent checking or updating the region status vector for the SR strategies, (4) *synchronization time* - the time spent performing synchronization operations, including instances when that synchronization is part of the SR access/done operations, and (5) *memory time* - the time spent stalled for normal memory operations.

The simulation experiments were conducted under a wide range of variations in the simulated architecture and the simulator parameters. Among these variations, we have found that the two most important considerations that impact the results are: (a) the cache size relative to the problem size, and (b) whether or not the system uses store buffers and weak consistency. Results are presented first for an eight processor system with weak consistency, store buffers, and 1 Mbyte caches. The large 1 Mbyte caches (large relative to the problem sizes) allow us to initially isolate the impact of coherence related misses and hence the impact of the coherence enforcement strategy. Later results for 64 Kbyte caches show the impact of capacity misses on the performance of these strategies.

6.1 SR flush-based algorithms

Figure 1 shows the first set of results for each of the five SR flush-based algorithms described in section 3.1. These results show that, in general, the pessimistic invalidation strategy (*SRp*) performs poorly, that the choice between optimistic and lazy invalidation (*SRo** versus *SRl**) is largely irrelevant, and that the choice between pessimistic and optimistic writeback (*SR*p* versus *SR*o*) depends on the data sharing patterns. These results are discussed in more detail next.

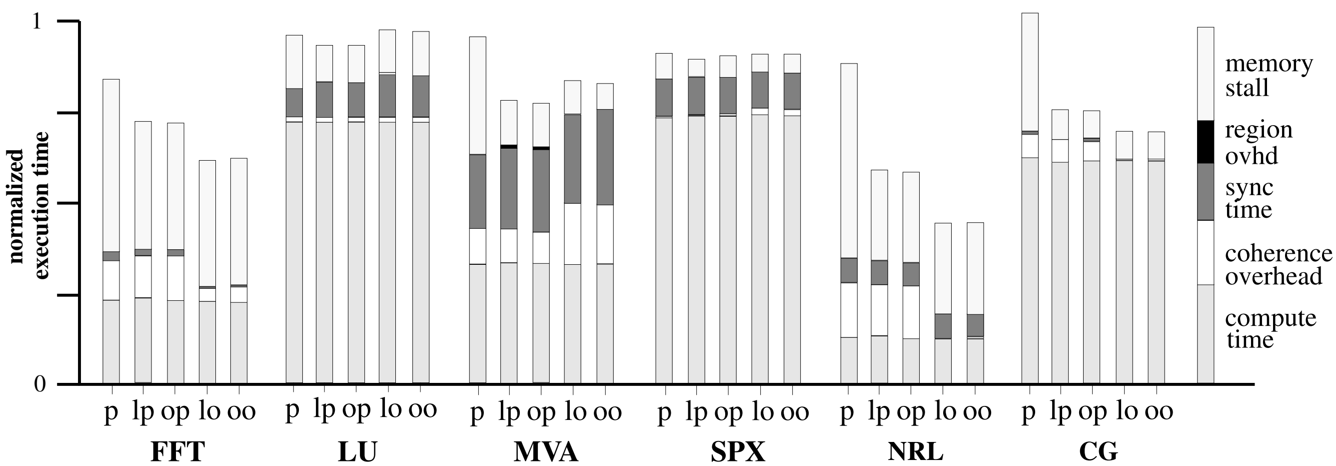

Figure 1. Results for the SR flush-based algorithms, *SRp, SRlp, SRop, SRlo*, and *SRoo*, using 1 Mbyte caches.

6.1.1 Pessimistic versus optimistic invalidation

The impact of pessimistic invalidation is apparent in the performance of *SRp* relative to *SRlp* in Figure 1. In *SRp*, regions are ejected pessimistically from the cache on each readdone or writedone operation. Because of this pessimistic behaviour, the performance of *SRp* is generally worse than *SRlp*, as would be expected. This performance difference may be a result of two factors: (a) additional coherence overhead due to unnecessarily ejecting data from the cache, and (b) lower cache hit rates. Interestingly, the first of these factors, the additional coherence overhead, has almost no effect on performance relative to *SRlp*. At writedone time, *SRp* and *SRlp* incur the same coherence overhead since the cost of writing a region back to memory is the same as the cost of writeback and invalidation together. The difference lies in the behaviour of the algorithms at readdone time, where *SRp* ejects the region from the cache while *SRlp* performs no coherence operations. However, since ejections of clean lines complete within the cache, these are relatively inexpensive operations, and make little difference in terms of overhead.

6.1.2 Optimistic versus lazy invalidation

In the lazy invalidation strategies, *SRlp* and *SRlo*, a processor checks the region status vector and invalidates its own copy of the region if its entry in the vector indicates stale data. The optimistic invalidation strategies, *SRop* and *SRoo*, eliminate the need to check the status vector at access time, and instead perform remote invalidations at writeaccess/writedone time. However, the results show little difference in the performance of optimistic and lazy invalidation. This is because the cost of checking the region status vector is negligible for these applications, and because the cost of invalidations is relatively low when clean data is being ejected from the cache.

6.1.3 Pessimistic writeback versus optimistic writeback

The choice of writeback strategy is more critical. In two of the applications, LU and MVA, the pessimistic writeback strategies perform better, by about 4% in LU and about 12% in MVA. In three of the other applications (FFT, NRL, and CG), optimistic writebacks perform better, by as much as 28% in FFT and 34% in NRL. Intuitively, pessimistic writeback strategies will perform better when the writeback is usually required. In these circumstances, optimistic writeback strategies incur the cost of sending a remote writeback request and stalling both the local and remote processors until the writeback completes. On the other hand, when writebacks are often not necessary, pessimistic writeback is likely to incur unnecessary coherence overhead. In SPX, a small number of blocks are flushed, thus making little impact on the runtime of the program. In the three applications for which the pessimistic writeback algorithms perform worse, there is a large disparity in the number of blocks flushed: 63% more blocks are flushed in FFT using pessimistic writeback, 99% in NRL, and 98% in CG. On the other hand, for those applications in which the pessimistic writeback strategies perform better, there is only a small difference in the number of blocks flushed pessimistically and optimistically.

6.2 Hardware versus software

Figure 2 shows the performance of the basic hardware directory-based strategy, referred to as *Dir1*, relative to two of the flush-based algorithms, *SRlp* and *SRoo*. *Dir1* performs worse than the best SR algorithm in four of the six applications (FFT, LU, MVA, and SPX) for these 1 Mbyte caches (the *Dir2* variation of this directory scheme is described later). In the other two applications (NRL and CG), its' performance is comparable to the best SR algorithm. Differences in the performance of the software and hardware strategies may include any of several factors: (1) the region overhead for the SR strategies, (2) differences in cache/memory interaction, (3) the granularity of cache management, and (4) the coherence enforcement strategy. The impact of each of these factors is discussed in next.

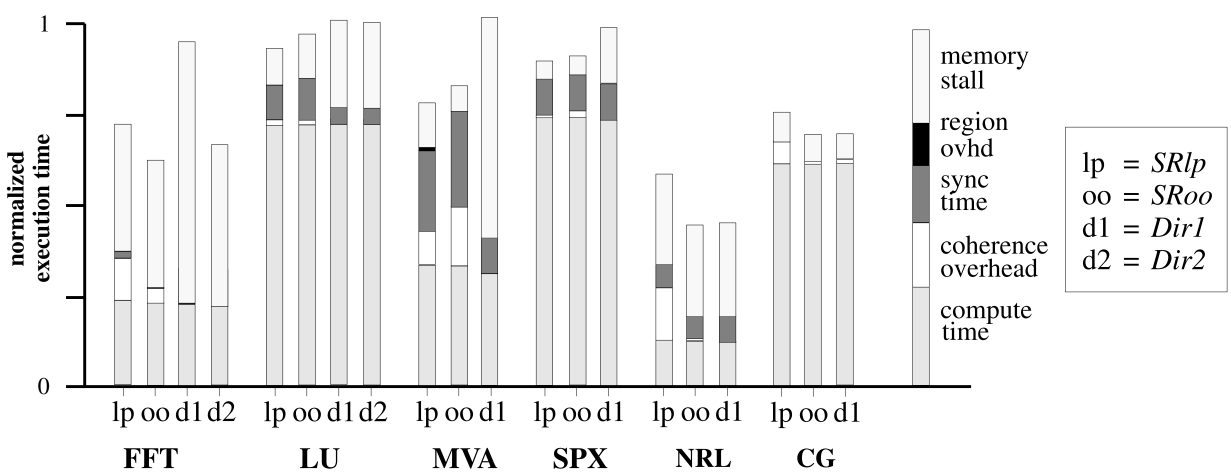

Figure 2. Flush-based algorithms versus the hardware directory scheme.

6.2.1 Region overhead

Each of the dynamic SR coherence strategies except *SRp* introduce additional overhead in checking and updating the region status vector (stored in memory) for each set of access/done operations. However, due to the coarse grained nature of these applications, the impact of this overhead is negligible in all applications. This is consistent with the results of the implementation and measurement study of *SRlp* using two of these applications on the Hector multiprocessor [22].

6.2.2 Cache/memory interaction

The cache/memory interaction behaviour of the software and hardware strategies differ in the way that store operations to clean lines are handled. In software, stores to clean lines always complete within the cache. In hardware schemes, stores to clean lines incur a cache miss, as the cache must first acquire an exclusive copy of the line before modifying it. Often, however, compilers may detect instances of loads followed by stores to the same address and replace the load operation with a read-exclusive operation that will bring the line into cache in a dirty state, thereby avoiding the subsequent cache miss. To take this optimization factor into account, we manually search through each of the six applications and optimize instances of loads followed stores to the same address. To limit the scope of this search to what the compiler would be able to do, only load-store sequences not crossing any synchronization boundaries are considered. The optimized version is shown in Figure 2 as *Dir2* for the two applications in which these load-store sequences are found (FFT and LU).

In both FFT and LU, using this optimization achieves performance for *Dir2* closer to those of the SR strategies. In FFT, for instance, this eliminates much of the 24% difference in execution time between the hardware strategies relative to the best software strategy (*SRlo*). However, *SRlo* still performs about 7% better, while in LU, the optimization has little impact. The remaining difference in performance can be attributed to other factors described later.

In MVA, the *Dir1* strategy has poorer performance than the SR strategies due to the store-to-clean-line problem, but unlike FFT and LU, this difference cannot be optimized away. In MVA, there is pair-wise sharing of the queue-length vectors computed in a two-dimensional grid. The sharing pattern of any particular set of memory addresses in these vectors is $P_i{:}write \rightarrow \{P_j{:}read, P_i{:}read\} \rightarrow P_i{:}write$. In the SR strategies, after a particular vector is brought into P_i's cache, all subsequent accesses to that vector from P_i complete within the cache. For the hardware scheme, on the other hand, the read operation by P_j forces the data in P_i to assume a clean state, and a subsequent write operation by P_i to that same data must first invalidate the copy in P_j, one line at a time, and acquire an exclusive copy for itself. This factor results in lower cache hit rates for *Dir1* and hence higher average memory access latency.

6.2.3 Granularity and Coherence Enforcement

The management of data at region granularity versus cache block granularity presents a number of trade-offs. In hardware strategies, there is a mismatch between the granularity at which hardware manages shared data (cache blocks) and the granularity at which the application shares data. In the SR schemes, on the other hand, there is a mismatch between the granularity at which data is managed (shared regions), and the basic units of coherence, which are still restricted to cache blocks. In both cases, granularity mismatches affect the potential for false sharing within cache blocks, and the efficiency of data management. We consider the problem of false sharing first.

For hardware strategies, false sharing may lead to the thrashing of a block between caches, but it does not affect consistency. For software strategies, false sharing may lead to inconsistent data and thus presents a more serious problem. A number of strategies are commonly used to deal with it, including padding shared data structures (or regions) or allocating data in an aligned fashion. Unfortunately, such techniques are not always feasible, particularly when data allocation is performed independently and prior to the partitioning of that data into shared regions. In the SR approach, false sharing can only occur when region boundaries do not match cache block boundaries. A simple solution then is to check the alignment of regions when they are defined and to disable caching for those blocks on which multiple regions overlap. In the experiments presented here, a combination of aligned data allocation and disabled caching for overlapping blocks is used. Simple data alignment strategies eliminate false sharing in four of the applications. For the other two, accesses to the uncacheable blocks account for a negligible fraction of the total and have no impact on performance.

To understand the efficiency of managing data at block granularity versus region granularity, we focus on the difference in coherence overhead in *SRoo* and *Dir*, both of which perform invalidations and writebacks in an optimistic manner. Coherence overhead for the hardware strategy is measured only when the system is sequentially consistent, and results for this are shown for two of the applications in Figure 3. For *SRoo*, there are a number of factors present in the coherence overhead, including (a) the software component - initiating per block coherence operations through software, and (b) remote interruption - the interruption of a remote processor and stalling both processors for remote writeback operations. The software component of *SRoo*'s coherence overhead is generally negligible, but the remote interruption cost is not for remote writebacks. In MVA, for instance, the coherence overhead for *SRoo* is double that of *SRlp*, and the difference is due almost entirely to the fact that both processors are stalled for remote coherence operations in *SRoo*. In the other applications, this factor is less apparent but still present.

The coherence enforcement overhead for *Dir* under SC is higher than *SRoo* in all applications, ranging from about 20% higher in FFT to several times higher in MVA. This difference can be attributed to the fact that coherence operations are initiated on a per-block basis in *Dir*. Under WC, the coherence enforcement overhead for *Dir* is not measured, but for the large 1 Mbyte caches, the overall run-time of *Dir* is not affected in any significant way. The reasons for this are discussed in section 6.3. The potential to reduce this coherence overhead through forwarding in hardware is examined in section 6.4.

6.3 Architectural effects

In our experimental study, a wide range of architectural parameters are varied and their impact on the results studied. This includes variations in interconnection and memory latency, interconnection bandwidth, cache size, cache block size, and system size. Although variations in each of these effect the runtime of the applications, they do not in general change the nature of the results. The two most important exceptions to this are variations in the cache size, and the use of store buffers and sequential versus weak consistency. We examine these factors next.

The results presented so far assumed large cache sizes in which capacity misses are for the most part absent. In practice, however, capacity conflicts are likely to be an important factor, either due to larger problem sizes or smaller cache sizes. In Figure 3, results are shown for cache sizes of both 64 Kbytes and 1 Mbyte, for a sequentially consistent system with no store buffers, and for a weakly consistent system with store buffers. For the sake of brevity, results are shown for just two representative applications, FFT and MVA.

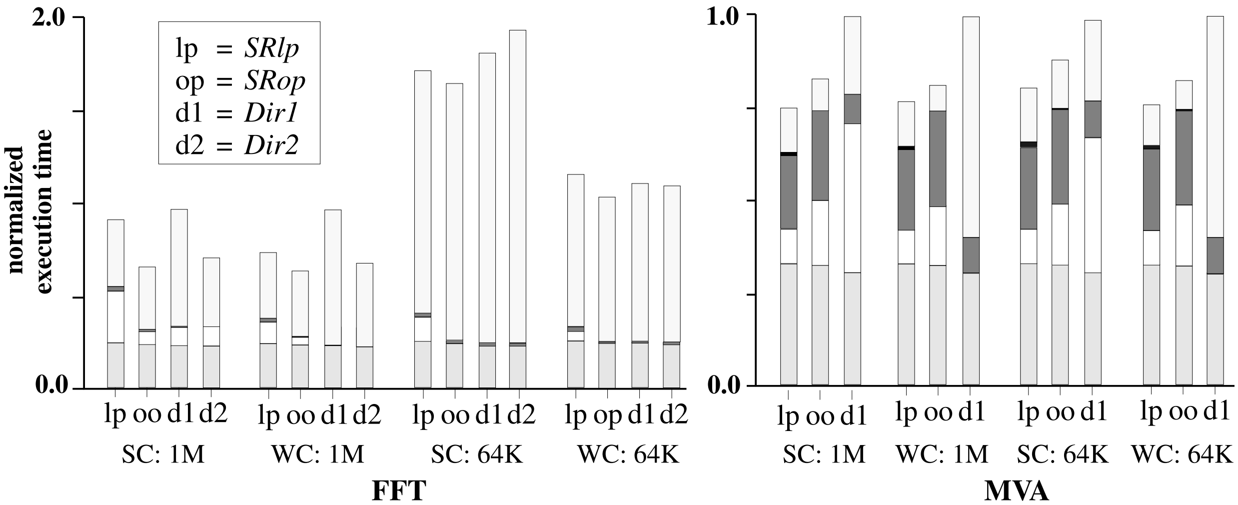

Figure 3. Results for the SC and WC versions, for 64 Kbyte and 1 Mbyte caches.

Using SC with 64 Kbyte caches, the run-time of FFT is almost doubled for all coherence strategies from the system with 1 Mbyte caches. Further, the gap between the pessimistic and optimistic SR algorithms, significant under 1 Mbyte caches, is diminished considerably under 64 Kbyte caches. The dominance of capacity misses on the run-time of the application causes the coherence enforcement algorithm to be less important, and pessimistically ejecting the region from the cache is not as bad when a significant proportion of that region will be ejected due to capacity conflicts anyways. In MVA, a much smaller per-processor working set leads the execution time to remain largely unchanged for cache size variations in the range of 64 Kbytes to 1 Mbyte.

The use of store buffers and weak consistency has two potential effects. First, it may reduce the latency of normal load and store operations by allowing these to overlap with computation and other memory operations. For the SR strategies, it may also reduce the latency of coherence operations by permitting the write back of multiple blocks to proceed in a pipelined fashion. Note that although a system with sequential consistency may also use store buffers and obtain some or most of these benefits, we have chosen for simplicity to model just the two extreme cases: SC with no store buffers and WC with store buffers.

When caches are 1 Mbyte in size, FFT and MVA show a slight decrease in the latency of loads and stores from to the SC to the WC version, but the decrease is no more than 3-5%. This is true for all six applications. A detailed examination shows that three different factors contribute to this relatively poor improvement: (1) high cache hit rates in which most misses are coherence related misses, (2) a high frequency of load operations relative to stores, and (3) memory contention. The decrease in coherence overhead with WC, on the other hand, is apparent for all of the SR algorithms in FFT. For SPX, LU, and CG, coherence overhead is already low enough that there is no change when using WC. Only MVA shows a significant coherence overhead that is not decreased with WC. Here again, the increase in contention under WC is responsible.

With WC, the results for the 64 Kbyte caches look closer to that of the 1 Mbyte caches, with the application run-time improving by about 40% in all of the coherence strategies in FFT. Clearly, the use of store buffers and WC is significant in tolerating the latency of capacity misses in the smaller caches for FFT. This is true to some degree in all of the applications except MVA.

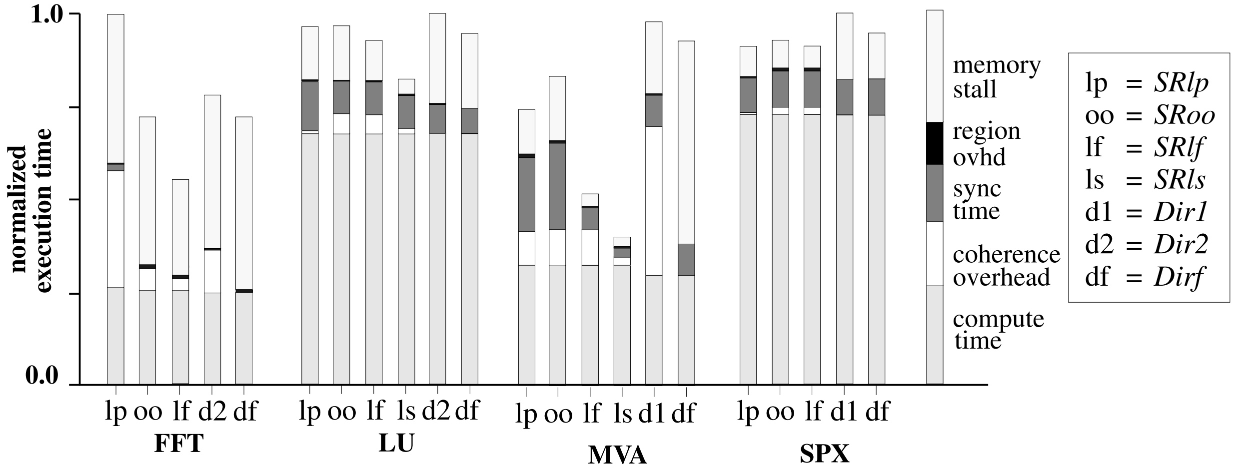

Figure 4. Results for SR forwarding, using 1 Mbyte caches.

6.4 SR forwarding

In this section, we examine the performance of the SR forwarding strategies described in section 3.2. Since the choice of invalidation strategy was found to be largely inconsequential for the flush-based algorithms, we consider only the forwarding algorithm with lazy invalidations, labelled *SRlf*, as well as a forwarding algorithm optimized for single-writer-multiple-reader sharing patterns, *SRls,* for those applications which exhibit these patterns. Two flush-based SR algorithms, *SRlp* and *SRoo*, are used for comparison. A version of the hardware strategy using forwarding, *Dirf*, is also shown.

Region forwarding has a number of potential advantages. Among these is the potential for reducing coherence enforcement overhead, increasing cache hit rates, and secondary effects such as reducing memory contention. However, other negative effects must be taken into consideration, including the potential for generating conflicts in the target caches (the cache receiving the forwarded blocks), and for *SRls* the potential for burdening the source with extra work in forwarding the region to other caches.

Figure 4 shows the results with forwarding for four of the applications (the other two are discussed but not shown for the sake of brevity), using 1 Mbyte caches. Results with smaller caches are shown later. *SRlf* shows an improvement over *SRoo* for three of the applications -- 25% in FFT, 4% in LU, and 39% in MVA. For the two applications with single-writer sharing patterns (LU and MVA), *SRls* shows still greater improvements, 16% over *SRoo* in LU and 52% in MVA. SPX shows little improvement, as do NRL and CG (not shown). With forwarding in *Dirf*, all four of the applications show fairly consistent improvements up to 8%. These results are discussed in some more detail next.

6.4.1 SR coherence enforcement overhead

The potential for reducing the coherence enforcement overhead using forwarding arises from the fact that, in the modelled architecture, transfers from cache to cache may be accomplished faster than writing data back to memory. This is true for two reasons. First, since the processor clock rate is four times higher than the memory clock rate and the caches are implemented in fast SRAMs, blocks can be processed more rapidly at the target than at memory. Second, since no per-block acknowledgment is required when regions are forwarded from cache to cache, the region can be pipelined more effectively through the interconnection network, without placing forwarded blocks into store buffers that tend to fill up rapidly.

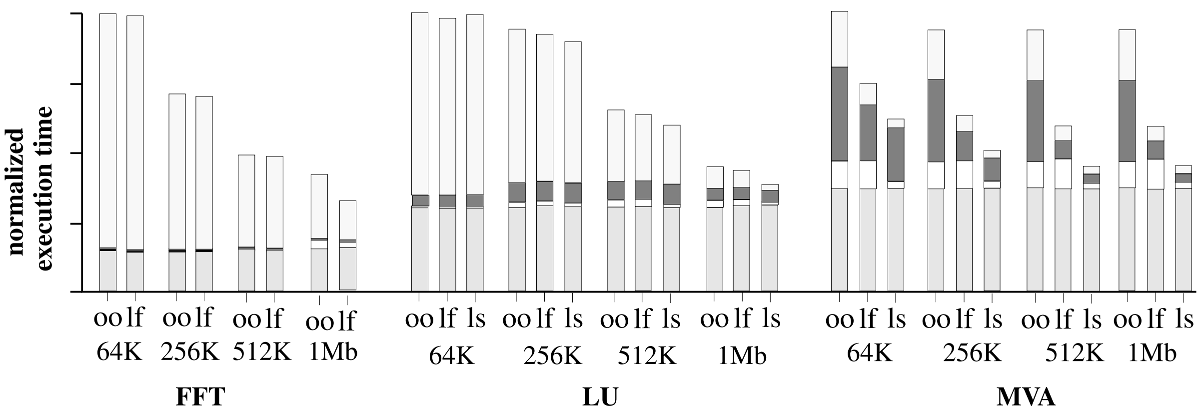

Figure 5. The effect of cache size variations on the SR forwarding algorithms.

The basic forwarding algorithm *SRlf* shows a reduction in coherence enforcement overhead for FFT but no change in LU, MVA, and SPX. Given the sharing patterns of these applications, this is to be expected. FFT uses forwarding on writeaccess operations to dirty regions, while the other three applications use forwarding on readaccess operations to dirty regions. With writeaccesses to dirty regions, the region is forwarded to the target processor instead of being written back to memory, resulting in close to a 50% reduction in the coherence enforcement overhead of FFT. For readaccesses to dirty regions, the region is written back to memory in addition to being forwarded to the target processor. Because of this, MVA, LU, and SPX show no decrease in the coherence enforcement overhead with *SRlf*. For regions that exhibit single-writer sharing patterns, readaccesses to dirty regions are optimized in *SRls* by allowing the source to forward the region to another processor without also writing it back to memory. Applied to MVA and LU, there is a reduction in coherence enforcement overhead similar to that witnessed in FFT using *SRlf*.

The large improvement in performance for LU using *SRls* (at least for the large caches) is especially interesting considering the possible negative impact of interrupting and burdening the source with extra work in serving multiple forwarding requests, and requiring the first readaccess from each of the other processors to block while they receive forwarded data rather than access the region from memory. Here, the extra work performed by the source slows that processor down, but the behaviour of LU is such that without this slow down, the source processor reaches the next synchronization point faster than any of the other processors and blocks there in either case. The source slow down is therefore not detrimental to the overall performance of the application. At the other end, the additional blocking by the other processors is also justified. With *SRlf*, the first processor receives a forwarded copy of the region and the other processors access it from memory, creating a hotspot at that memory unit. In *SRls*, forwarding eliminates the contention at memory, and each processor receives the data it needs faster than it would have one block at a time through memory.

6.4.2 Cache conflicts

For the SR forwarding strategies, cache size plays a crucial role. Conflicts in the source or target cache limit the usefulness of forwarding. Figure 5 shows the impact of cache size on performance for *SRoo*, *SRlf*, and *SRls* for the three applications which show improvements with the larger cache sizes. Table 3 shows the frequency of conflict misses in the source and target caches as the cache size is varied.

Conflicts in the source cache cause some of the blocks comprising the region to be absent in the cache at the time of forwarding. In terms of performance, a high degree of conflicts in the source will likely limit

cache size	fraction region in source			forwarding hit rate			useful data eject rate		
	FFT	LU	MVA	FFT	LU	MVA	FFT	LU	MVA
64 Kbyte	0.14	0.49	0.99	0.89	0.86	0.97	0.0001	0	0.01
256 Kbyte	0.26	0.76	1.0	1	0.65	0.99	0	0	0
512 Kbyte	0.26	0.92	1.0	1	0.54	0.99	0	0	0
1 Mbyte	0.63	1.00	1.0	1	0.51	0.99	0	0	0

Table 3: Cache conflicts in the source and target caches, using region forwarding.

the usefulness of forwarding if few blocks remain to be forwarded. In Figure 5, the average fraction of the region in the source at the time of forwarding is shown for cache sizes ranging from 64 Kbytes to 1 Mbyte. MVA is not affected significantly by these cache size variations due to its small working set. For the other two applications, there is a significant decrease in the fraction of the region in the source cache, from 0.63 to 0.14 in FFT, and from 1.0 to 0.49 in LU using *SRlf*.

Interference in the target cache can pose a significant problem for region forwarding. This interference can be of two types: (1) cross-interference - interference between the region being forwarded and other data residing in the target cache, and (2) self-interference - interference among blocks of the forwarded region. Self-interference is not a problem at the time of forwarding, since only those portions of the region that reside in the source are forwarded, and these can not interfere with each other in the target cache. However, forwarded data may be ejected in the target prior to being used, due to either cross-interference or self-interference.

Table 3 shows two measures of the effect of cache conflicts in the target as the cache size is varied: (1) the forwarding hit rate - the fraction of the region used in the target prior to being ejected due to capacity conflicts or eventual coherence ejections, and (2) useful data ejection rate - the fraction of forwarded blocks that eject other useful data from the target cache. A forwarded block is said to replace useful data in the target cache if the replaced block is referenced in the target prior to a reference to the forwarded block that replaced it.

For the 1 Mbyte caches, there are no conflicts in the target caches. However, in LU, even in the absence of conflicts, the forwarding hit rate is only 0.51. This has to do with the access patterns in LU. At phase k of the N phase computation, only the last N-k elements of each row are used, but the region definition remains fixed at N elements. For this reason, on average, only 50% of the forwarded region is actually used in the absence of other conflicts. MVA remains largely unaffected by cache size variations as before. FFT experiences high forwarding hit rates for smaller cache sizes even as the source conflicts increase. For instance, with 256 Kbyte caches, only 26 percent of the region is in the source at the time of forwarding, but 100% of the forwarded blocks are used in the target prior to getting ejected. For 64 Kbyte caches, 14 percent of the region is in the source and still almost 90% is used in the target. For LU and the smaller cache sizes, less of the region is in the source, but a higher fraction of that region is used in the target. Interestingly, the ejection of useful data using forwarding does not have any impact for any of the applications, even for the smaller cache sizes, as very little useful data is ejected.

6.5 Summary

The experimental results presented in this section highlight the performance of the SR coherence alternatives discussed in this paper for a suite of parallel applications. Among the flush-based coherence alternatives, the key criterion in choosing an algorithm for a particular data structure is in deciding whether to use pessimistic or optimistic writeback. Further, this decision can be made based upon the expected or measured frequency with which pessimistic writeback results in unnecessary flushes compared to the optimistic writeback alternative. Pessimistic invalidation strategies perform poorly in general, and the choice between optimistic and lazy invalidation is largely irrelevant. Relative to a representative hardware coherence strategies, the Shared Regions approach offers performance that is comparable to or significantly better for the coarse-grained applications studied here. The use of forwarding in the SR strategies shows considerable improvements in performance for some applications, but only when the cache size is large relative to the data that is being forwarded. Further investigation is necessary to determine the precise conditions under which forwarding is useful, but in general, it appears that forwarding through software in the SR approach can contribute significantly to improve performance when used selectively for data that may benefit from it.

7.0 Summary

In this paper, we have investigated a class of strategies for cache coherence based upon the integration of a program-level abstraction for shared data, referred to as Shared Regions (SR), with run-time cache management. We propose a wide range of algorithms for cache coherence that are based upon the Shared Regions program annotations, including some novel algorithms that communicate regions directly between caches to improve performance. The practical value of the Shared Regions approach is measured based both on the performance of these coherence alternatives relative to a representative hardware strategy, and the complexity of programming with this model for a suite of parallel applications.

The results of this work show that Shared Regions offer a flexible and efficient alternative for cache coherence on shared-memory multiprocessors. The flexibility is achieved by a software implementation in which the application may chose a coherence strategy that is best suited for a particular task. For the basic flush-based coherence algorithm, our experiments show that the key decision that must be made is whether to perform writebacks in an optimistic or pessimistic fashion. Forwarding regions directly between caches results in significant improvements in performance over the flush-based strategies for three of the six applications used in the study, but only when cache sizes are large relative to the regions being forwarded. For smaller cache sizes, conflicts in the cache limit the impact of forwarding. However, as architectural trends towards larger cache sizes and longer memory latencies continue, forwarding is likely to become more useful. An implementation of region forwarding via software should be possible on any architecture that has the ability to send and receive data directly between processors (in software) in an efficient manner, and supports the ability to place data into a local cache through software without generating a cache miss.

The efficiency of the SR approach is measured relative to hardware coherence strategies. For the coarse-grained applications used in this study, the Shared Regions approach offers performance that is comparable to or significantly better than the representative hardware approach. Further, for these applications, programming with SR represents only a small overhead beyond that of programming with a plain shared memory interface. This suggests that even on architectures with hardware cache coherence, the best performance may be achieved through the selective use of the SR coherence strategies, when it is feasible to do so; when the overhead of inserting SR annotations is tolerable and the sharing patterns are coarse-grained. In our study, we find that decisions about the placement of the SR annotations can be made based solely on local information. This suggests that, in the future, some or most of this programming overhead may be offset by compilers that automatically insert these annotations once the set of shared regions have been identified by the user.

References

[1] S. Adve, V. Adve, M. Hill, and M. Vernon. Comparison of hardware and software cache coherence strategies. In *18th Int'l. Symp. on Computer Architecture*, pages 298-307, May 1991.

[2] S. Adve and M. Hill. Weak Ordering -- A new definition. In *16th International Symposium on Computer Architecture*, May 1989.

[3] A. Agarwal, R. Simoni, M. Horowitz, and J. Hennessy. An evaluation of directory schemes for cache coherence. In *15th Int'l. Symp. on Computer Architecture*, pages 280-289, June 1988.

[4] B. Bershad, M. Zekauskas and W. Sawdon. The Midway distributed shared memory system, In *Proc. of COMPCOM'93*, pages 528-537, Feb. 1993.

[5] J. Carter, J. Bennett, and W. Zwaenepoel. Implementation and performance of Munin. In *13th Symp. on Operating Systems Principles*, pages 152-164, Oct. 1991.

[6] L. Censier and P. Feautrier, A new solution to coherence problems in multicache systems. In *IEEE Transactions on Computers*, c27(12), pages 1112-1118, Dec. 1978.

[7] D. Chaiken, J. Kubiatowicz and A. Agarwal. LimitLESS directories: A scalable cache coherence scheme, *Fourth Int'l. Symp. on Architectural Support for Programming Languages and Operating Systems*, Apr. 1991.

[8] R. Chandra, K. Gharachorloo, V. Soundararajan, and A. Gupta. Performance evaluation of hybrid hardware and software distributed shared memory protocols. In *Eighth Int'l. Conference on Supercomputing*, Jul. 1994.

[9] A. Cox, S. Dwarkadas, P. Keleher, H. Lu, R. Rajamony, and W. Zwaenepoel. Software versus hardware shared-memory implementation: A case study. In *21st Int'l. Symp. on Computer Architecture*, pages 106-117, Apr. 1994.

[10] M. Feeley and H. Levy. Distributed shared memory with Versioned Objects. In *Conf. on Object-Oriented Programming Systems Languages, and Applications*, Oct. 1992.

[11] B. Gamsa. Region-oriented main memory management in shared-memory NUMA multiprocessors. Master's Thesis, University of Toronto, 1992.

[12] K. Gharachorloo, A. Gupta, and J. Hennessy. Performance evaluation of memory consistency models for shared memory multiprocessors. In *Fourth International Symposium on Architectural Support for Programming Languages and Operating Systems*, 1991.

[13] P.B. Hansen. Concurrent programming concepts. In Computing Surveys, Vol. 5, No. 4, pages 223-245, December 1973.

[14] M. Hill, J. Larus, S. Reinhardt and D. Wood. Cooperative Shared-Memory: Software and hardware support for scalable multiprocessors. In *Fifth Int'l. Symp. on Architectural Support for Programming Languages and Operating Systems*, Oct. 1992.

[15] C. Hoare. Toward a theory of parallel programming. In Operating Systems Techniques, Academic Press, New York, pages 61-71, 1972.

[16] P. Keleher, A. Cox and W. Zwaenepoel. Lazy consistency for software distributed shared memory. In *18th Int'l. Symp. on Computer Architecture*, May 1992.

[17] P. Keleher, S. Dwarkadas, A. Cox, and W. Zwaenepoel. TreadMarks: Distributed shared memory on standard workstations and operating systems. Technical Report 93-214. Rice University, Nov. 1993.

[18] M. Lam and M. Rinard. Coarse-grain parallel programming in Jade. In *3rd Symp. on Principles and Practices of Parallel Programming*, April 1991.

[19] L. Lamport. How to make a multiprocessor computer that correctly executes multiprocess programs. *IEEE Transactions on Computers*. 28(9), pages 690-691, Sept. 1979.

[20] D. Lenoski, J. Laudon, K. Gharachorloo, A. Gupta, and J. Hennessy. The directory-based cache coherence protocol for the DASH multiprocessor. In *17th Int'l. Symp. on Computer Architecture*, pages 148-159, May 1990.

[21] S. Owicki and A. Agarwal. Evaluating the performance of software cache coherency. In *Proc. 3rd Int'l. Conf. on Architectural Support for Programming Languages and Operating Systems*, Apr. 1989.

[22] H. Sandhu, B. Gamsa, and S. Zhou. The shared regions approach to software cache coherence. In *Proc. Symp. on Principles and Practices of Parallel Programming*, May 1993.

[23] H. Sandhu and K. Sevcik. An analytic study of dynamic hardware and software cache coherence strategies. In *Proc. of ACM Sigmetrics and Performance '95*, May 1995.

[24] H. Sandhu. Shared Regions - A strategy for efficient cache management in shared-memory multiprocessors. Ph.D. Thesis, University of Toronto, 1995.

[25] Z.G. Vranesic, M. Stumm, D.M. Lewis and R. White. Hector - a hierarchically structured shared-memory multiprocessor, In *IEEE Computer*, Jan. 1991.

[26] Z.G. Vranesic *et al*. The NUMAchine multiprocessor. Technical report, University of Toronto, 1994.