# The Shared Regions Approach to Software Cache Coherence on Multiprocessors

**Harjinder S Sandhu** **Benjamin Gamsa** **Songnian Zhou**

Computer Systems Research Institute
University of Toronto

Oct 1992
Technical Report 277

*To Appear in* Fourth ACM SIGPLAN Symposium on
Principles and Practices of Parallel Programming, May 1993

**Abstract**

The effective management of caches is critical to the performance of applications on shared-memory multiprocessors. In this paper, we discuss a technique for software cache coherence that is based upon the integration of a program-level abstraction for shared data with software cache management. The program-level abstraction, called *Shared Regions*, explicitly relates synchronization objects with the data they protect. Cache coherence algorithms are presented which use the information provided by shared region primitives, and ensure that shared regions are always cacheable by the processors accessing them. Measurements and experiments of the Shared Region approach on a shared-memory multiprocessor are shown. Comparisons with other software based coherence strategies, including a user-controlled strategy and an operating system-based strategy, show that this approach is able to deliver better performance, with relatively low corresponding overhead and only a small increase in the programming effort. Compared to a compiler-based coherence strategy, the Shared Regions approach still performs better than a compiler that can achieve 90% accuracy in allowing cacheing, as long as the regions are a few hundred bytes or larger, or they are re-used a few times in the cache.

# 1 Introduction

Shared-memory multiprocessors offer a simple and convenient model for programming parallel applications. For highly parallel architectures, the Non-Uniform Memory Access (NUMA) characteristics of these machines make memory access latency a central issue. Techniques for cache coherence attempt to exploit the advantages of low latency processor caches while minimizing cache management overhead. Weak consistency models attempt to reduce memory access latency by allowing memory access operations to be buffered and overlapped based upon the information provided by synchronization events in the program. In this paper, we present a new approach to shared data management on shared-memory multiprocessors based upon the integration of an extended weak consistency model with software cache coherence.

Hardware cache coherence techniques that have been proposed include snoopy caches and directory schemes [Ste90] [LLG$^+$90][CKA91]. A problem with hardware solutions is that they are either non-scalable or require complex and expensive hardware. Thus, software techniques that can guarantee consistency and achieve reasonable cache hit performance are desirable. On multiprocessors without hardware coherent caches, the task of enforcing consistency may be assumed by the user [PBG$^+$85] [GKM$^+$82] [MOR86], the compiler [CKM88] [CV90], or by the operating system [SUK92]. User-based approaches introduce significant complexity to the programming task, while compiler-based approaches require sophisticated static analysis which is difficult for in a language such as C. Operating system-based approaches often perform poorly because they determine cacheability on a per-page basis.

Weak consistency models rely on the fact that synchronization points and memory accesses are related [AH90]. For instance, by assuming that the value of an element being written to by processor $P_i$ will not be used by another processor until after the next synchronization point has been reached by $P_i$, $P_i$ can avoid waiting for the completion of the write operation. Read operations initiated after a write operation has been started may overtake and complete before the write operation has completed. A weakly consistent system must be aware of synchronization points, and ensure consistency at those points. The Release Consistency

[GLL$^{+}$90] model is a weak consistency model that uses knowledge of two types of synchronization operations, *acquire* and *release*, to further relax the constraints on the ordering of memory operations.

Our approach, which we call the Shared Regions approach (SR), is a variation of the weak consistency concept in which synchronization objects are explicitly associated with the data they protect. The shared data in the application is partitioned into shared regions of memory, and the application explicitly identifies the region it wishes to access. A processor that is accessing a shared region is guaranteed exclusive or read-shared access to that region. Because of this, shared regions are always *cacheable* by a processor accessing that region; the system has only to ensure consistency of a shared region at the shared region access control points in the application.

Whereas traditional weak consistency models relax the constraints on the ordering of memory operations and thereby improve performance, the Shared Regions approach extends this by permitting more efficient management of caches and main memory. In this paper, we show that SR cache coherence can be implemented efficiently in software, with no special hardware support. NUMA memory management can also be enhanced by exploiting knowledge of shared regions, but this is discussed in a separate paper [xxxx].

An overview of the rest of this paper is as follows: Section 2 discusses the program-level abstraction that relates synchronization objects with the data they protect. Section 3 presents algorithms for SR cache coherence. Section 4 presents some experiments from an implementation of SR cache coherence on a shared-memory multiprocessor using a few sample applications, and compares this strategy with some other software-based cache coherence strategies. Section 5 presents a comparative analysis of SR cache coherence and a compiler-based strategy. Section 6 summarizes the ideas presented in this paper.

# 2 Shared Regions

A Shared Region is a high-level programming abstraction for shared data. In much the same way as a user may specify the threads of execution in the application but leave the scheduling and management of those threads to the system, a user specifies the shared regions in the application and leaves the management of that data to the system. In this section, we discuss the specification and integration of shared regions into a high-level programming language such as $C++$.

A *shared region* is defined loosely as a set of memory locations that are accessed together (i.e., within the same task) and in the same mode (read or write) in a parallel application. A region may encompass an entire user-defined data object, it may encompass only some subsection of a data object, or it may comprise subsections of multiple data objects. For example, in a parallel matrix multiplication ($C = AB$) where each task computes a row of $C$ using a row of $A$ and the entire matrix $B$, each row of $A$ and $C$ may be defined as a separate shared region, while the entire matrix $B$ may be defined as a single shared region.

Using the Shared Region abstraction, performance critical shared data is partitioned into shared regions, and accesses to those regions must be requested explicitly using shared region access primitives (called SR primitives). A *token* representing a shared region is explicitly bound to some data using a *BindRegion* call. The SR primitives *ReadAccess, ReadDone, WriteAccess*, and *WriteDone* are operations on shared regions. *ReadAccess* is a request for read-only access to the shared region, and *WriteAccess* is a request for exclusive write access to the shared region. A series of references to a region are preceded by a ReadAccess or WriteAccess call, and terminated by a ReadDone or WriteDone call.

The use of SR primitives for synchronization obviates the need for other synchronization mechanisms (such as semaphores or monitors) in many cases. However, this introduces a subtle shift in paradigm; whereas traditional synchronization constructs typically protect a critical section of code, the SR primitives protect a section of shared memory. Figure 1 shows an example of region specification and contrasts this with the use of semaphores. In this example, an array of regions $r$ is defined, and each region is bound to a separate row of $A$ much as a separate lock is associated implicitly with a row of $A$ in the semaphore version.[1] A process requesting WriteAccess to a region wait until all previous ReadAccess or WriteAccess requests to that region have completed. A process requesting WriteAccess to a region must wait until all previous ReadAccess requests to that region have completed.

[1] In this example, the *BindRegion* function specifies a contiguous region comprising one row of the matrix. Non-contiguous regions may be specified using a function which allows multiple contiguous regions to be referenced as a single region.

| Semaphore approach | Shared Regions approach |
|---|---|
| double A[N][N]; // A is shared | double A[N][N]; // A is shared |
| lock k[N]; | Region r[N]; // each row of A is a region |
| | |
| for (a=0; a<N; a++) | for (a=0; a<N; a++) |
| ... initialize lock k[a] | r[a].BindRegion( &A[a], rowsize ) |
| | . . . |
| lock(k[i]) | r[i].WriteAccess() |
| for (j=0; j<N; j++) | for (j=0; j<N; j++) |
| A[i][j] = ... | A[i][j] = ... |
| unlock(k[i]) | r[i].WriteDone() |

Figure 1: C++ region specification example compared with semaphores.

In a case where Shared Region primitives replace other synchronization constructs, the programmer's task is not any more difficult than programming a non-Shared Region version of the same application, so the programming complexity of Shared Regions is minimal. Additionally, the cost of acquiring the shared region lock should not be considered an overhead of this technique. In other cases, however, the locking aspect of the region access primitives may be redundant, for example, when using ReadAccess where the region is already guaranteed not to be write-shared at that point in the application. In such cases, the ReadAccess call is still required for coherence purposes, but the locking overhead is avoided by using non-locking versions of the SR primitives.

The advantage of the Shared Region approach is that the sharing characteristics of the application are exposed to the underlying system, thus permitting more efficient shared data management. The next section presents some algorithms for software cache coherence based upon the information provided by SR primitives. The utility of the Shared Region approach to software cache management will depend on: 1) the effectiveness of SR cache coherence, in terms of cache hit rates and application performance, 2) the overhead of managing the cache in software, and 3) the complexity of writing applications with Shared Region specifications. These issues are addressed using measurements and experiments in section 4.

# 3 Algorithms for SR cache coherence

In this section we discuss two algorithms for SR cache coherence. For an implementation of these algorithms, the following software cache control operations are required:

1. *cache-invalidate* (addr, size): invalidate *size* bytes starting at virtual address *addr*.
2. *cache-copyback* (addr, size): copy dirty bytes to memory for the specified address range.
3. *cache-flush* (addr, size): copyback any dirty bytes and also invalidate the address range.

For *write-through* caches, both the *cache-copyback* and the *cache-flush* operations become unnecessary, although for generality we assume that the cache is not write-through. We also assume that shared regions are aligned to cache line boundaries, since software cache control operations usually operate at cache line granularity.

Two algorithms for SR cache coherence are shown in Table 1. In the Cache-Flush algorithm, regions are invalidated from the cache at ReadDone time, and flushed at WriteDone time. Using the Cache-Validate algorithm, the region remains in the cache at ReadDone/WriteDone time, but it must be checked for validity at Access time. If the region has been modified since the last access by this processor, the region must be invalidated.

The Cache-Flush algorithm is the simplest SR cache management algorithm to implement, and behaves similarly to the user-based coherence approach described in [OA89, MOR86] and the compiler-based approach described in [CKM88]. However, the Cache-Flush algorithm may perform poorly for many applications

| Access Control Primitive | SR Cache Management Algorithm | |
|---|---|---|
| | Cache-Flush | Cache-Validate |
| ReadAccess and WriteAccess | - | if status[P] == HI<br>cache-inval(R)<br>status[P] = HV |
| ReadDone | cache-inval(R) | - |
| WriteDone | cache-flush(R) | cache-copyback(R)<br>foreach Cache $\neq$ P and<br>status[Cache] == HV<br>status[Cache] = HI |

Table 1: SR cache coherence algorithms: region R, processor P

because a region may be unnecessarily invalidated from the cache after every ReadDone or WriteDone operation. Invalidations from processor $P_i$ are necessary only in those cases where $P_j$ $(j \neq i)$ modifies region $R$ before $P_i$ accesses it again. In all other cases, this algorithm incurs unnecessarily the overhead both of invalidating the region from the cache, as well as that of reloading the region on a subsequent access.

The Cache-Validate algorithm performs only necessary cache invalidations and is thus preferable. To implement this algorithm, we associate with each region a state vector which contains information about the current state of all caches in the system with respect to that region. The cache states are: a) this cache does *Not Have* a copy of the region (state NH), b) this cache *Has* a *Valid* copy (state HV), and c) this cache *Has* an *Invalid* copy (state HI). For either ReadAccess or WriteAccess calls, the region's cache state vector is checked. If the current state of $R$ for cache $C_i$ is HI, then $R$ has been modified by another processor since the last Done operation by processor $P_i$, and it must be invalidated from $C_i$ before $P_i$ may continue. At the time of a WriteDone operation on $R$ by $P_i$, the state of those caches (except $C_i$) for which the current state is HV is set to HI to indicate that the region has been modified.

The state vector is left uncached in order to ensure that it remains consistent. A separate lock for the state vector should not be necessary as long operations on individual state vector entries can be performed atomically, since, in the Cache-Validate algorithm, the state vector is accessed by a processor only after it has first acquired access to the region. A processor with WriteAccess to a region is guaranteed that no other processor is currently accessing the state vector, and it is free to modify it. A processor performing a ReadAccess will only access and potentially modify its own entry in the vector, so it will not interfere with other processors performing a ReadAccess, and it is ensured that another processor cannot be performing a WriteAccess and potentially modifying its entry. The performance issues related to managing the state vector are addressed in the next section.

# 4 Experiments

On a multiprocessor without hardware support for cache coherence, there are a few alternatives for cache management: 1) leave shared data uncached, 2) manage shared data within the operating system, as in Hurricane [SUK92], 3) allow the user to explicitly manage the cache from the application, as in the RP3 [PBG+85], the NYU Ultracomputer [GKM+82], and the ELXSI [MOR86], 4) use compiler-based techniques, as suggested in [CKM88] [CV90], and 5) use Shared Region-based techniques, as discussed in this paper. In this section, we examine the costs involved in using region-based techniques, and compare the performance of SR cache coherence with the first three alternatives. Section 5 presents an analysis comparing SR coherence with a compiler-based technique for cache management.

Experiments with SR coherence were done on a hierarchical shared-memory multiprocessor called Hector [VSLW91] using some sample applications. Hector consists of up to 256 16MHz MC88100 CPU's with 16KB data and 16KB instruction caches, and 4MB local memory per processor. The interconnection consists of buses at the lowest level connected by a hierarchy of bit-parallel rings. Our experiments were done on a 16-processor configuration of 4 processors per bus and 4 buses on one ring.

Hector has no hardware cache coherence. By default, Hector's operating system, Hurricane, maintains cache coherence on a per-page basis (where each page is 4KB). According to the coherence protocol, cacheing of a page is disabled whenever writes to the page are detected by the hardware and the page is shared by multiple processors. In the SR implementation, the standard HURRICANE cache coherence mechanisms are disabled for a specific memory pool. Shared data that is defined as Shared Regions is allocated in a this memory pool, and coherence of this pool is controlled through region access control operations, using the Cache-Validate cache coherence algorithm by default. Typically, only shared data that is critical to performance is allocated in the region memory pool. Shared data that is not performance critical is managed using the existing HURRICANE cache management protocol.

## 4.1 Application description

Three applications were used in these experiments. We first describe each application, and then characterize the use and overhead of shared regions in each application. The applications are:

- LU-Decomposition (LU): Row-oriented LU decomposes a dense $N \times N$ matrix $A$ into a lower triangular matrix $L$ and an upper triangular matrix $U$. The sequential algorithm goes through $N-1$ phases; at phase $i$: 1) row $i$ becomes the pivot row and is reduced, and 2) all rows $j$ ($j > i$) are reduced using the pivot row. This continues until all rows have become the pivot once. LU-decomposition is performed in parallel by assigning the rows of $A$ to the $P$ processors in a round-robin fashion. Each row is defined as a separate shared region. In our experiments, the matrix size for LU is 512×512.

- Mean Value Analysis (MVA): MVA calculates average customer response times and queue lengths for a 2-class closed queueing network [RL80] by successively calculates the values for a system with $n_1$ and $n_2$ customers in each of the two classes, using the values for systems of populations $(n_1-1, n_2)$ and $(n_1, n_2-1)$. A task solving a system of a particular size requires the queue length vectors from its two parent tasks in order to proceed, and each queue length vector is defined as a shared region. Figure 2 shows a skeleton of the code for the task $(n_1, n_2)$, together with the six lines of region access control code. In the experiments, $C = 200, K = 400$.

```
mva_task(n1, n2)
{
    ... allocate my queue length vector
    regions[n1, n2].WriteAccess();

    regions[n1, n2-1].ReadAccess();
    ... use left parent's vector to update mine
    regions[n1, n2-1].ReadDone();

    regions[n1-1, n2].ReadAccess();
    ... use right parent's vector to update mine
    regions[n1-1, n2].ReadDone();

    regions[n1, n2].WriteDone();
}
```

Figure 2: A single task in parallel MVA.

- Matrix Multiplication (MM): MM calculates the product of two $N \times N$ matrices $A$ and $B$ and places the result in matrix $C$. This is done in parallel by creating $N$ threads and assigning to each thread one row of $A$ and $C$ each, and dividing the threads evenly among processors. The entire matrix $B$ is read-shared by all threads. In the Shared Regions version, each row of $A$ and $C$ is defined as a separate region, while all of $B$ is defined as a single shared region. Thus, each thread $T_i$, upon beginning execution, does a ReadAccess on row $A_i$, a ReadAccess for $B$, and a WriteAccess on $C_i$. A matrix size of 460x460 is used in the experiments.

## 4.2 Regions overhead in applications

Now we examine the use of shared regions in each application and their overheads. Figure 3 shows the number of region constructs (Access and Done pairs) in each application code and the number of calls to region primitives during program execution. The number of region constructs in the application code is one measure of the complexity of using the region paradigm in each application. In LU, region constructs replace synchronization constructs that are required in a non-SR version of this program, so the only additional effort on the part of the programmer is that of initially binding the regions to the data they protect. For each of these applications, there is a natural decomposition of data into shared regions, so the task of initially defining the regions is trivial.

| | LU | MVA | MM |
|---|---|---|---|
| Synch Constructs Replaced with Regions | 2 | 0 | 0 |
| New Region Constructs Added | 0 | 3 | 3 |
| WriteAccess/WriteDone Calls | 512 | 80,000 | 920 |
| ReadAccess/ReadDone Calls | 8,190 | 160,000 | 1840 |

Figure 3: Application characteristics, independent of the number of processors

For the MVA and MM applications, no existing synchronization constructs are replaced, and three new region Access/Done pairs are added. From a synchronization point of view, the locking of the region is unnecessary, so the non-locking versions of the primitives are used in the MVA version to avoid the locking overhead. In the MM program, the region overhead was low enough (as shown later) that this overhead did not matter.

Even though both MVA and MM required new constructs in the region version, the programming effort required is insignificant, since there are only six lines of new code for shared region access primitives, and their use is intuitive in these applications (as an example, see Figure 2). However, the number of calls made to region primitives during execution is substantial. This is especially true in the case of MVA, which heavily taxes the region primitives.

Figure 4 shows breakdowns of the program execution times on 16 processors. For LU, only 2.9% of the time is spent executing region code. In MM, the region code execution time, at 0.2%, is completely insignificant. In the case of MVA, region code occupies a significant 11% of the execution time. This is to be expected considering the very large number of times region access primitives are called during execution for MVA. Because of the pair-wise sharing in MVA, however, this overhead should not increase significantly with system size.

| | LU | | MVA | | MM | |
|---|---|---|---|---|---|---|
| | time | % of total runtime | time | % of total runtime | time | % of total runtime |
| User code | 8.94 sec | 97.1% | 7.65 sec | 88.9% | 18.38 sec | 99.8% |
| Region code | 0.27 sec | 2.9% | 0.96 sec | 11.1% | 0.042 sec | 0.2% |
| Program runtime | 9.21 sec | - | 8.61 sec | - | 18.42 sec | - |

Figure 4: Application runtime measurements, 16 processors

Next, we examine the region access calls in more detail. A microsecond timer was used to measure separate time components of each call (Figure 5). For both the ReadAccess and WriteAccess functions, the time components are: 1) the time spent spinning to acquire the region, 2) the time to actually acquire the lock after spinning [2], 3) for the Cache-Validate algorithm, the time required to invalidate the region if it is dirty, and 4) the region management overhead, which includes the time spent managing the region status vector. For the ReadDone and WriteDone operations, the time components are: 1) the time required to

[2] In the non-locking versions of the SR primitives, lock spinning and acquisition time is zero.

invalidate or copyback the region from the cache, and 2) the overhead, which includes the time required to update the region vector and the time to release the lock in the locking version of the SR primitives. Figure 5 shows the average time for each component for all calls to the region primitives, and the time spent as a percentage of the total program runtime.

| | | LU Cache-Validate | | MVA Cache-Validate No Locks | | MM Cache-Flush | |
|---|---|---|---|---|---|---|---|
| Region Primitive | Cost Factor | avg time | % of total runtime | avg time | % of total runtime | avg time | % of total runtime |
| ReadAccess and WriteAccess | 1. spin-wait | 413 | 2.6% | - | - | 0 | 0 |
| | 2. lock acquire | 7 | 0.05% | - | - | 20 | 0.01% |
| | 3. cache-inval | 0 | 0 | 8 | 1.5% | - | - |
| | 4. overhead | 9 | 0.05% | 9 | 1.5% | 3 | 0.001% |
| ReadDone | 1. cache-inval | - | - | - | - | 614 | 0.17% |
| | 2. overhead | 5 | 0.05% | 3 | 0.08% | 15 | 0.004% |
| WriteDone | 1. cache-copyback | 288 | 0.07% | 202 | 7.0% | 101 | 0.01% |
| | 2. overhead | 9 | 0.002% | 2 | 0.07% | 5 | 0.007% |

Figure 5: Application region primitives overhead measurements (in microseconds)

In the case of LU, most of the time is spent either spin-waiting to acquire the lock (2.6% of program execution time, or 90% of the time spent executing region code). This cannot be considered SR overhead since these locks replaced synchronization constructs that are necessary in LU. Copying back the region from the cache also occupies a significant amount of time for each call (288 $\mu s$). The actual region overhead is insignificant.

In MVA, the non-locking versions of the SR primitives are used so no time is required waiting or acquiring the region lock. As in LU, the average time spent flushing the cache is high compared to the other factors. The region overhead itself is relatively low.

In MM, most of the time in region code is spent invalidating $B$ after every ReadDone. This average time is quite large since $B$ itself is very large (1692800 bytes), and even though any particular element of $B$ is unlikely to remain in the cache, simply executing a null invalidate (that is, an invalidate of an element that is not in the cache) requires a significant amount of time.

In summary, several observations can be made on the measurements in this section. First, the complexity of using Shared Regions is low for these applications. Second, the overhead of the SR primitives is generally low. Most of the time spent executing region code is spent either waiting to acquire the region, or invalidating or copying back the region to memory. It can be argued that neither of these factors can be considered region overhead, because lock acquisition can be turned off if it is not necessary to the algorithm, and the copyback and invalidation will likely be necessary regardless of which cache management strategy is used.

## 4.3 Application performance

In this subsection, we present speedups for each of the applications, and compare the SR versions with 1) a version where Hurricane page-based cache management is used on an implementation of the application with little architecture specific code (the OSpage version), and 2) a version that was hand-tuned to take into account the specific architectural characteristics of the experimental platform with Hurricane's page-based cache management running underneath (the Handtuned version). The alternative of leaving shared data uncached was also initially considered, but for each application, the performance on 16-processors is worse than the 1-processor case, so this alternative will not be discussed further. The algorithm for user-controlled caches described in [OA89, MOR86], in which the user explicitly flushes data before leaving a critical section, is similar to the SR Cache-Flush algorithm, so we consider the performance of the Cache-Flush algorithm to be representative of this class of techniques.[3]

[3] Unlike user-controlled caches, however, the task of coherence using the SR Cache-Flush algorithm is automated and the details hidden from the user, leading to a simpler programming interface.

In the OSpage and Handtuned versions, the central concerns with respect to cacheing are the false sharing of data on fixed-sized pages and reenabling cacheing for a page after a write to the page has caused cacheing to be disabled. Thus, much of the hand-tuning for the Handtuned version focuses on addressing these concerns. The programming effort involved is considerable, requiring days of performance debugging in an application such as LU. In the SR version of the algorithm, we take a version that has no architecture specific code and insert SR primitives. For both LU and MM, there is no difference in the performance of the Cache-Flush and Cache-Validate algorithms, because of the cache reuse patterns and the small fraction of time spent executing region code. The difference in these two algorithms is shown only in the case of MVA.

Figure 6 shows the results for LU-decomposition using matrix size 512×512. Speedups are calculated relative to the best sequential version of the algorithm not using Shared Regions, and speedup curves are annotated with response times for the 16-processor cases. In the SR version, the rows are always cacheable by the accessing processor, and the corresponding region management overhead is low (as shown in the previous subsection). The Handtuned version is limited by the fact that data still must be managed at page granularity, and thus, it still performs slightly worse than the SR version which has close to linear speedup.

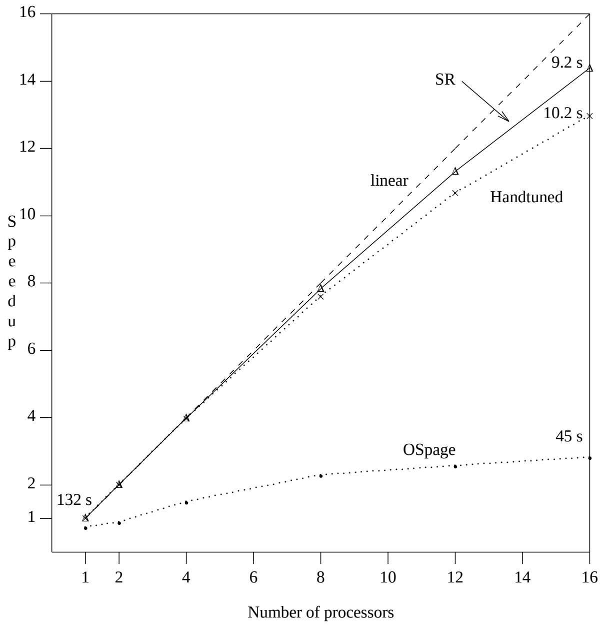



Figure 6: LU-decomposition on 512×512 matrix.

Results for MVA are shown in Figure 7. Here, a queue length vector is 1600 bytes long. Since each occupies only about 1/3 of a page, multiple vectors are dynamically allocated onto the same page. This makes the vectors uncacheable except in the SR version. Each task reads the queue length vector of each of its parents only once, and has only a 50% chance of being able to reuse its parents' cached data. Even with this low amount of cache reuse, and the significant fraction of time spent executing region code (11% from Figure 4), the speedup curve for the SR Cache-Validate version is straight and close to linear for up to 16 processors. The SR Cache-Flush algorithm does not perform as well, because it unnecessarily invalidates the region at every Done operation.

Figure 8 shows the results of matrix multiply on 460×460 matrices. In the Handtuned version, each processor is assigned a group of contiguous rows, but its perimeter rows of $C$ fall on pages shared with other processors, causing these to become uncacheable. However, these perimeter rows account for only a small fraction of the total number of accesses, so, although the SR version allows full-cacheing of all rows and thus performance better than the Handtuned version, the difference is small. The substantial difference between the SR speedup and the linear curve is due to the lack of memory locality; a version of MM with $B$ pre-replicated at each memory unit achieves linear speedup.

A number of other applications that are not discussed here were written using Shared Regions. These

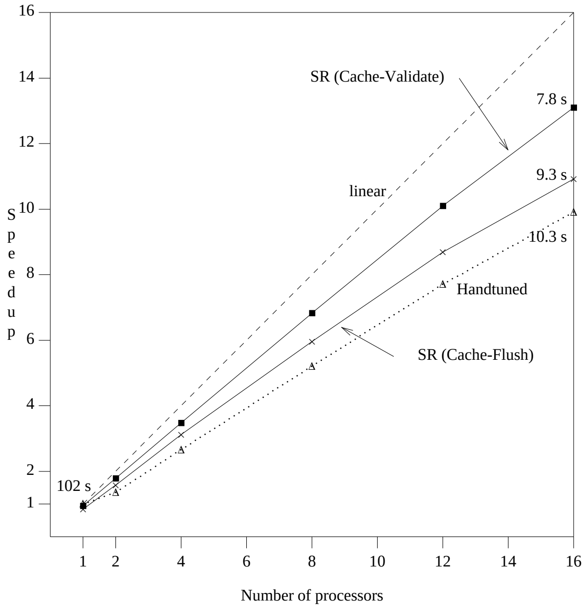



Figure 7: MVA for $C = 200, K = 400$.

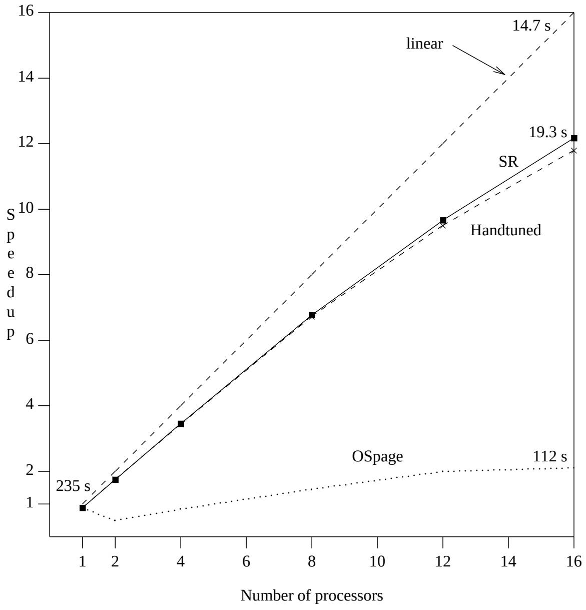



Figure 8: Matrix multiplication: 460x460.

include Fast Fourier Transform, Mergesort, Hough Transforms, and Traveling Salesman. For the applications with regular data structures, using the Shared Regions approach is simple. For problems without regular data structures, such as the Traveling Salesman and Mergesort, using Shared Regions is not as straightforward. However, all of the applications perform well with Shared Regions compared with some of the other alternatives for cache coherence discussed here.

A number of conclusions can be drawn from the performance results in this section. Leaving shared data uncached is not a viable alternative to cache coherence. Having the operating system enforce cache coherence at fixed page-granularity requires the user to expend considerable effort tuning the application to deal with architecture specific problems. Even a hand-tuned version may not perform as well as a SR version if data is still managed at fixed page granularity. In comparison to the Cache-Flush algorithm (and hence the user-controlled cache management techniques that behave similarly), the Cache-Validate algorithm performs significantly better in the case of MVA, and approximately the same for LU and MM. In all cases, because data is always cacheable in SR, and the corresponding region overhead is relatively low, the applications are able to achieve good performance using the Cache-Validate algorithm.

# 5 Comparison to compiler-based technique

In this section we compare the performance of the SR Cache-Validate strategy with the expected behavior of a compiler-based strategy. Although several algorithms for compiler-based cache management have been suggested, in [CKM88] and in the Cedar work [CV90], compiler techniques are limited by the fact that static analysis techniques are unable to accurately predict the runtime behavior of a program. In our comparison, the behavior of the compiler scheme is based upon the expectation that future advances in compiler technology will make these more feasible, using compiler error rate to parameterize the accuracy of the compiler. Adve *et al* [AAHV91] compare compiler-based techniques with hardware coherence techniques using a similar notion of compiler accuracy, but they are more comprehensive in classifying the types of errors the compiler may make. In our model, we will consider, for simplicity, only those errors in which data that could have been cached is accessed as uncacheable.

Contrasted with compiler techniques, SR manages the cache entirely at runtime. The advantage of this is that while a compiler technique will make conservative errors, in SR shared regions are always cacheable. The disadvantage is that there is some overhead for managing shared regions at runtime, in particular, that of managing the region state vector. We use a simple model to highlight the tradeoffs between static and runtime management of shared data.

The model has four high-level parameters: 1) region size $S$, in words, 2) region reuse $R$ – the number of times a region is accessed for every Access/Done operation, 3) write probability $W$ – the fraction of Access operations that are WriteAccess operations, and 4) compiler error rate $E$ – the percentage of conservative errors made by the compiler. Default values for these parameters are also shown in Figure 9. The write probability (0.5) is intentionally pessimistic in order to strain both software coherence strategies.

Low level parameters used in this model, also shown in Figure 9, are based upon measurements in HECTOR and assume that all data is on non-local memory being accessed over the bus. For simplicity, we ignore both NUMAness and contention, and assume infinite size caches. Additionally, we assume that for a particular Access/Done session (referred to as a *region access*), the entire region is referenced or modified.

| High Level Parameters | | | Low Level Parameters | | |
|---|---|---|---|---|---|
| parameter | | default | cost factor | | Cost $C$ |
| region size | $S$ | 256 bytes | region overhead | $C_O$ | 200 cycles per Access/Done |
| region reuse | $R$ | 4 | cache-inval | $C_I$ | 1 cycle per cache-line |
| write prob | $W$ | 0.5 | cache-copyback | $C_B$ | 70 cycles per line |
| compiler error | $E$ | 0.1 | cache access | $C_A$ | 1 cycle per word |
| | | | memory word access | $C_M$ | 20 cycles |
| | | | memory line access | $C_L$ | 30 cycles per line |

Figure 9: Model parameters. A line is 4 words

Within the model, processors randomly access a region, choosing to do a WriteAccess with probability $W$. For the base case, in which the region is uncacheable, the region access time is $SRC_M$. In the SR case, each region access costs $C_O$, plus $\frac{S}{4}C_L$ if the region is in the NH state at this processor, $SC_I + \frac{S}{4}C_L$ if the region is in the HI state, or $SC_A$ if it is in the HV state. Since subsequent reuse of the region takes place in the cache, $SC_I(R-1)$ is also added. If the operation is a WriteAccess, cost $\frac{S}{4}C_B$ is also added to account for copying the region back to memory.

In the compiler case, the compiler makes a conservative error with probability $E$ and accesses the region as uncached, inducing cost $SRC_M$ as in the uncached case, plus $\frac{S}{4}C_I$ if the region is already in the cache. Otherwise, the region is accessed as cacheable, with cost $\frac{S}{4}C_L + SC_A(R-1)$ if the region is not currently in the cache, or $SC_A R$ if the region is currently in the cache. If the region is written to (probability $W$), cost $\frac{S}{4}C_B$ is also added to account for copying the region back to memory.

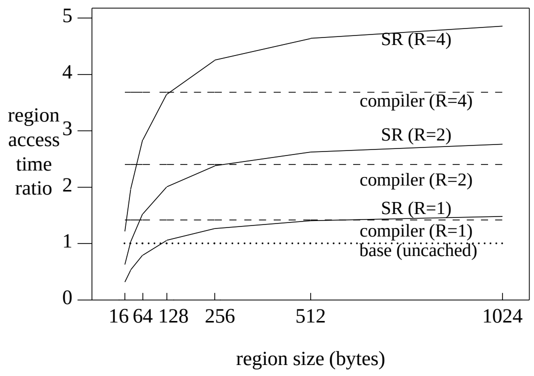
The *region access time ratio* is the average region access time of the base case divided by the average region access time of the other two techniques. A higher region access time ratio indicates an improvement in performance. Figure 10 shows a number of experiments using the region access time ratio as a performance metric.

Figure 10(a) shows a comparison of SR coherence with the compiler strategy for various region sizes and reuse factors. For a reuse factor of 1, SR coherence performs better than the compiler with 90% efficiency only when the region is larger than about 512 bytes. However, as the reuse factor increases, SR coherence performs comparatively better. For a reuse factor of 4, SR coherence performs better than the compiler technique when the region size is larger than about 100 bytes.

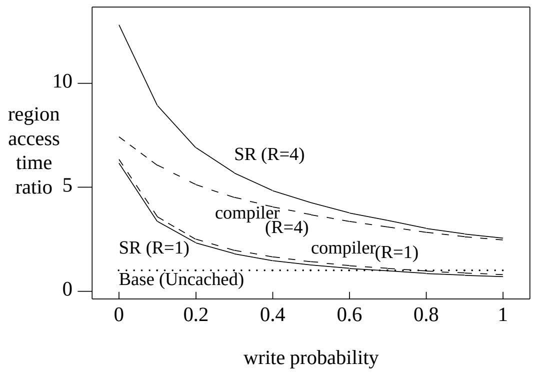
Figure 10(b) shows the effect of varying the write probability, with the default region size at 256 bytes. When the write probability is very low, SR coherence performs significantly better than the compiler with 90% efficiency when the reuse factor is 4. However, as the write probability approaches 1, both techniques perform similarly. The higher write probability induces more flush operations and decreases the benefits of the cache, so both techniques degrade for higher write probabilities.

Figure 10(c) shows the effect of varying the compiler efficiency from 0 to 1, with the region size again at 256 bytes and the write probability at 0.5. When the reuse factor is 1, the compiler requires 70% efficiency to perform better than SR coherence. When the reuse factor is 4, the compiler requires about 95% efficiency in order to perform better than SR coherence.

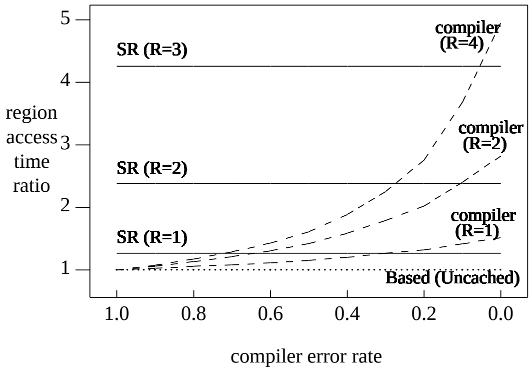
In summary, the analysis in this section shows that good performance with the SR Cache-Validate requires that regions be sufficiently large or that they have a high reuse factor. In such cases, compiler techniques will




(a) Region Size and Reuse (R)




(b) Region write frequency




(c) Compiler Error Rate

Figure 10: Comparison of SR Cache-Validate with compiler-based strategy

require a high degree of accuracy in order to be competitive. In the applications discussed in the previous section, regions were both large and the reuse factor was high, thereby contributing to good performance.

# 6 Related Work

Methods for annotating programs to elicit greater information about program behavior has been proposed in a number of different contexts. Jade [LR91] proposes *with* and *withonly* constructs that expose sharing characteristics to the systems, but unlike our approach, the motivations for these constructs in Jade is to permit the compiler to detect concurrency. The tokens used in the Jade constructs are not explicitly related to the data they encapsulate. In the approach proposed in [HLRW92], called Cooperative Shared Memory (CSM), users may optionally annotate their programs with explicit *check-in* and *check-out* of shared data, which serve as hints to the underlying system. CSM has the advantage over the Shared Region approach that the annotations are only advisory and thus need not be put in. However, because these annotations are only advisory, the CSM hardware still needs to implement a limited directory strategy to manage coherence. The advantage of the Shared Regions approach is that no hardware support is required, and a wide variety of coherence strategies can be implemented without being constrained to hardware determined fixed block sizes.

The approach proposed in [FL92], called Versioned Distributed Object Management (VDOM), bears a closer resemblance to our work. VDOM's notion of *fragmented objects* appears to be closely related to our notion of Shared Regions, and the primitives used to access fragmented objects, AcquireRead/AcquireWrite, appear identical to our ReadAccess/WriteAccess primitives. In both approaches, coherence is managed at the granularity of shared regions or fragmented objects, but the differing environments that they have been applied to lead to entirely different solutions and problems. For instance, in the VDOM case, the loosely coupled distributed environment means that higher overhead in accessing the fragmented objects can be tolerated; in the tightly-coupled shared-memory environment we have assumed, the overhead of the ReadAccess/WriteAccess calls themselves become a fundamental issue, along with managing coherence at that granularity.

# 7 Conclusions

In this paper, we introduced a technique for software cache coherence based on the integration of an abstraction for shared data, called *Shared Regions (SR)*, with software cache management. Algorithms for cache coherence which exploit knowledge of shared regions were discussed. These algorithms can be implemented easily and efficiently in software, without any special hardware support.

An implementation of the Shared Region approach was done on a shared-memory multiprocessor, and experiments with a few sample applications show favorable results. Measurements of the overheads of software region management for these applications show that most of the time spent in region code is spent performing operations such as cache-invalidation and cache-copyback, which would be required using any cache management strategy; the actual overhead particular to shared regions is relatively low. Comparisons with other software coherence strategies such as user-controlled caches, and operating system based-coherence, show that the Shared Regions approach is able to deliver better performance with little programming complexity. Comparisons with a compiler-based cache management strategy show that the Shared Region approach performs well except when the region is very small or is only used in the cache once or twice. In most other cases, a compiler strategy would have to make fewer than 5% errors in order to be competitive.

Work is currently in progress to compare, using a simulation model, the Shared Regions approach to hardware directory strategies such as the full-map scheme used in DASH [LLG$^+$90], and the LimitLESS scheme proposed in [CKA91], as well as strategies that require special hardware architectures or hardware support, such as VMP [CGB89] which relies on a hierarchical cache organization. Additionally, although the 16-processors experiments in this paper give some indication of the scalability of the Shared Regions approach, work is also in progress to analyze, via simulation, its behavior on much larger systems.

# References


[AAHV91] S. Adve, V. Adve, M. Hill, and M. Vernon. Comparison of hardware and software cache coherence schemes. In *18th International Symposium on Computer Architecture*. ACM, May 1991.

[AH90] S. Adve and M. Hill. Weak ordering - a new definition. In *17th International Symposium on Computer Architecture*. ACM, May 1990.

[CGB89] D. Cheriton, H. Goosen, and P. Boyle. Multi-level shared cacheing techniques for scalability in VMP-MC. In *16th Intl. Symposium on Computer Architecture*, May 1989.

[CKA91] D. Chaiken, J. Kubiatowicz, and A. Agarwal. LimitLESS directories: A scalable cache coherence scheme. In *4th Intl. Conference on Architectural Support for Programming Languages and Operating Systems*. ACM, April 1991.

[CKM88] R. Cytron, S. Karlovsky, and K. McAuliffe. Automatic management of programmable caches. In *1988 International Conference on Parallel Processing*, 1988.

[CV90] H. Cheong and A. Veidenbaum. Compiler-directed cache management in multiprocessors. *IEEE Computer*, 1990.

[FL92] M. Feeley and H. Levy. Distributed shared memory with versioned objects. In *Conference on Object-Oriented Programming Systems, Languages, and Applications*. ACM, October 1992.

[GKM+82] A. Gottlieb, C. Kruskal, K. McAullife, L. Rudolph, and M. Snir. The NYU Ultracomputer – designing an MIMD shared memory parallel computer. In *9th Intl. Conference on Computer Architecture*, 1982.

[GLL+90] K. Gharacharloo, D. Lenoski, J. Laudon, P. Gibbons, A. Gupta, and J. Hennessy. Memory consistency and event ordering in scalable shared-memory multiprocessors. In *17th International Symposium on Computer Architecture*. ACM, May 1990.

[HLRW92] M. Hill, J. Larus, K. Reinhardt, and D. Wood. Cooperative shared memory: Software and hardware for scalable multiprocessors. In *5th Intl. Conference on Architectural Support for Programming Languages and Operating Systems*. ACM, October 1992.

[LLG+90] D. Lenowski, J. Laudon, K. Gharachorloo, A. Gupta, and John Hennessy. The directory-based cache coherence protocol for the DASH multiprocessor. In *17th International Symposium on Computer Architecture*. ACM, 1990.

[LR91] Monica Lam and Martin Rinard. Coarse-grain parallel programming in Jade. In *3rd Symposium on Principles and Practices of Parallel Programming*. ACM, April 1991.

[MOR86] S. McGrogan, R. Olson, and N. Roda. Parallelizing large existing programs - methodology and experience. In *1986 Spring COMPCON*, 1986.

[OA89] S. Owicki and A. Agarwal. Evaluating the performance of software cache coherence. *ACM*, pages 230–242, 1989.

[PBG+85] G. Pfister, W. Brantley, D. George, S. Harvey, W. Kleinfelder, K. McAliffe, E. Melton, V. Norton, and J. Weiss. The IBM research processor prototype (RP3); introduction and architecture. In *1985 International Conference on Parallel Processing*, 1985.

[RL80] M. Reiser and S. Lavenberg. Mean value analysis of closed multichain queueing networks. *JACM*, April 1980.

[Ste90] Per Stenstrom. A survey of cache coherence schemes for multiprocessors. *IEEE Computer*, June 1990.

[SUK92] Michael Stumm, Ronald Unrau, and Orran Krieger. Clustering micro-kernels for scalability. In *Microkernels and Other Kernel Architectures Workshop*. USENIX, April 1992.

[VSLW91] Z.G. Vranesic, M. Stumm, D.M. Lewis, and R. White. Hector - a hierarchically structured shared-memory multiprocessor. *Computer*, January 1991.