Region-Oriented Memory Management in Shared-Memory Multiprocessors

Harjinder S. Sandhu and Benjamin Gamsa and Songnian Zhou

Technical Report CSRI-269
April 1992

Computer Systems Research Institute
University of Toronto
Toronto, Canada
M5S 1A1

Region-Oriented Memory Management in Shared-Memory Multiprocessors

Harjinder S. Sandhu
Benjamin Gamsa
Songnian Zhou

Computer Systems Research Institute
University of Toronto

Abstract

Effective caching and memory locality are essential for good performance of parallel applications on non-uniform memory access (NUMA) shared-memory multiprocessors. The need for *multi-level memory coherence* requires that consistency of shared data be maintained both within each level and across adjacent levels of the memory hierarchy. Existing static program analysis techniques for managing shared data at the cache level are limited because dynamic program data sharing behaviour is difficult to determine at compile time. Traditional schemes for managing shared data at the main memory level are limited because they manage shared data at page granularity and without knowledge of application behaviour.

We introduce the notion of *shared regions* as the natural unit of data sharing in parallel applications, and show that this concept may be used in effective cache and main memory management. Algorithms for region-oriented cache and main memory management are presented. Region-oriented cache management has the advantage over other software cache coherence techniques in that shared regions are always *cacheable*. Region-oriented main memory management improves upon traditional NUMA memory management by maintaining coherence at region granularity. This eliminates page-level false sharing as a concern for coherence and allows for more effective replication strategies. The use of regions also permits the use of relaxed memory coherence protocols.

Implementation of region-oriented memory management was done on a hierarchical shared-memory multiprocessor called HECTOR. Experiments with a few sample applications show that region-oriented techniques applied to UMA programs can attain or even exceed the performance of hand-tuned NUMA applications.

1 Introduction

Effective caching and memory locality are essential for good performance of parallel applications on non-uniform memory access (NUMA) shared memory multiprocessors. On such architectures, accessing the local CPU cache requires only a few cycles, but the cost of a local memory access is often on the order of tens of cycles, and remote memory access may be as high as a few hundred cycles. Techniques for caching and NUMA memory management attempt to reduce access latency by moving data closer to the processor accessing it. For data that is shared among multiple processors, attempting to move data closer to each of the accessing processors introduces the problem of coherence. Although caches and main memory are both levels of the memory hierarchy, their functional differences have led to the development of two disjoint areas of research: cache coherence, and NUMA memory management.

Proposed solutions for cache coherence include hardware-based solutions such as snoopy caches and directory schemes, and software techniques requiring sophisticated program analysis. A problem with hardware solutions is that they are usually either non-scalable or require complex and expensive hardware [Ste90] [ASHH88]. Thus, software techniques that can guarantee consistency and achieve reasonable cache hit performance may be desirable. Unfortunately, traditional software techniques rely on static analysis of data dependence and data sharing within programs, something which is difficult for general purpose parallel programming paradigms. On multiprocessors without coherent caches, the task of enforcing consistency may be assumed by the memory management system, but this usually leads to poor cache hit performance if cacheability of data is determined on a per-page basis.

NUMA memory management algorithms use a variety of replication and migration techniques to move data closer to the accessing processor [JEK91][BSF+91]. While these techniques have had some degree of success for specific applications, they are limited by the fact that they operate on virtual memory pages and without any knowledge of application sharing behaviour. False sharing on pages has a severe impact on performance; pages may become unreplicatable or uncacheable, they may be migrated further from some processors accessing parts of the page, and in the worst case, false write sharing may induce page thrashing. Thus, much of the recent work has focused on mitigating the impact of false sharing, primarily through freezing and defrosting of pages [JEK91][BSF+91].

The fundamental problem is that *since data sharing in applications does not occur at page granularity, a page is the wrong abstraction for managing shared data.* Unfortunately, in the absence of knowledge about application sharing behaviour, there is no choice but to manage data at fixed size units such as cache lines or main memory pages. This means that in order to increase the efficiency of automatic cache and main memory management techniques, either the compiler or the user must provide more information about the sharing structure in the application.

System management of caches and main memory in NUMA multiprocessors may be improved by integrating these with the application's abstractions for shared data. The integration can be achieved by gradually migrating system data management facilities to the application level, or by exposing application sharing characteristics to the system and then relying on the system to optimize data management based upon this information. We view this process as being analogous to the evolution of threads and thread management facilities in shared-memory multiprocessors. Although originally the domain of the operating system, the cost of kernel-level thread management and the need for application specific customization led to the emergence of user-level thread packages, and eventually to the integration of user-level threads with system level support for thread management [ABLL91].

Applications typically use the abstraction of a *data object*, which can be any user-defined data structure. For instance, in a matrix multiplication, the matrices to be multiplied and the result matrix may be the principal objects. Munin, a distributed shared-memory (DSM) system, attempts to apply this notion of an object to managing shared data [CBZ91]. In Munin, when an object is declared, the user must specify the sharing patterns of the object according to the classifications recognized by the system. Munin then tries to optimize the coherence mechanisms for specific objects in the application based upon their sharing patterns.

While this approach may be effective in DSM systems, where the high communication cost allows for greater overhead in the software coherence mechanisms, we chose not to adopt this approach in the context of shared-memory multiprocessors. The following observations about parallel applications influence the approach we used:

1. shared data objects are not always statically defined, but may be dynamically created and destroyed during program execution;

2. sharing patterns are not static, but may change during program execution;

3. the granularity of sharing may be finer than user-defined objects;

4. parts of shared objects may exhibit different sharing patterns.

In attempting to define the granularity at which sharing occurs in parallel applications, we use the notion of a *shared region*:

> A *shared region* is a collection of shared data locations that are always accessed together and in the same mode [read and/or write] in a natural unit of computation.

A region may encompass an entire user-defined object, it may be only part of a user-defined object, or it may comprise parts of multiple user-defined objects. For example, in the matrix multiplication example, the individual rows or columns of the matrices may be defined as regions. In order to use the concept of regions for system data management, the regions in the application must be exposed to the system. Once the system becomes aware of shared regions, cache and main memory coherence may be enforced at region granularity, thus overcoming the limitations of traditional cache and memory management techniques.

This paper discusses a system for Region-Oriented Memory Management (referred to as ROMM). Section 2 discusses how the regions in the application may be exposed to the system. Sections 3 and 4 examine techniques for region-oriented cache coherence and region-oriented memory management. Section 5 discusses experiments with ROMM using several applications running on a hierarchical shared-memory multiprocessor called HECTOR. Section 6 presents an overview of related work, and Section 7 presents the conclusions of this paper.

2 Regions and Coherence

2.1 Region specification

Region specification requires that: a) region boundaries be clearly defined, and b) access to regions be controlled through access control primitives. While compiler techniques that could adequately identify the regions and operations on them in a program would be ideal, a study of such techniques is beyond the scope of this paper. Instead, we focus upon the performance implications of exposing shared regions to the run-time system. The application programmer is responsible for region specification through a set of primitives made available in the C++ language. Figure 1 illustrates region specification in C++ and contrasts this with the use of semaphores.

Semaphore approach	Regions Approach

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

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

In Figure 1, the *BindRegion* function defines the boundaries of an array of regions. [1] A process must explicitly request read or write access to that region before accessing it. This is different from traditional semaphore-type synchronization mechanisms in which a process requests access to a critical section of code.

Access control operations on regions are the following:

- *ReadAccess:* access region for reading and allow multiple concurrent read accesses.
- *WriteAccess:* exclusive access to region for write operations.
- *ReadDone:* relinquish a region that was acquired via ReadAccess.
- *WriteDone:* relinquish a region that was acquired via WriteAccess.

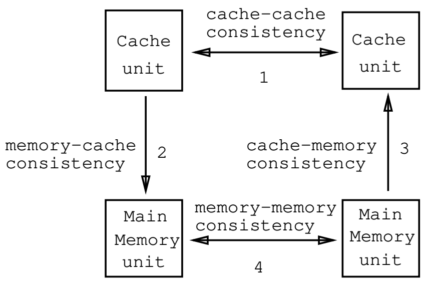
These access control primitives have two components: 1) a locking mechanism that provides synchronization between processes sharing a region, and 2) cache and main memory control operations that interact with the system to enforce coherence. In most cases, region access control

[1] In this example, the *BindRegion* function specifies a contiguous region comprising one row of the matrix. Non-contiguous regions may be specified by: a) including additional parameters in the BindRegion function indicating a stride length and the number of strides, or b) using a function called *BundleRegions*, which allows multiple regions to be referenced as a single one.

replaces existing synchronization constructs (such as semaphores, as illustrated in Figure 1), so the region locking mechanisms do not impose additional overhead and the effort to annotate a program with region specifications is usually minimal. For cases in which the region locking mechanisms are redundant, access control is still required for coherence, but the locking mechanism may be turned off if the application programmer chooses to do so.

2.2 Regions and multi-level coherence

Coherence of regions in ROMM is enforced at the region access control points in the application. Within the memory hierarchy, the types of coherence mechanisms used depend on the relationship of a particular level in the hierarchy to other levels in the hierarchy. Different levels in the hierarchy maintaining multiple copies of data introduces the need for *multi-level memory coherence*. This is illustrated in Figure 2 for the common two level cache/main memory hierarchy.



	cache	memory
cache	1 caches consistent with each other	3 data in cache reflects latest version in memory
memory	2 updates in cache reflected in memory	4 memory units consistent with each other

Figure 2: Multi-level memory coherence.

In architectures without hardware cache coherence, caches may not have the ability to communicate directly, so the only way to keep caches consistent with each other is through memory. The *region-mapped memory* M_i^R, is defined as the memory unit(s) containing the copy of region R that is mapped into cache C_i. Multi-level coherence may be maintained by applying the following rules to regions at access control points:

(a) *memory-cache:* at WriteDone time, updates to region R in cache C_i are reflected in M_i^R.

(b) *memory-memory:* between Done and Access time, the copy of region R at M_i^R is made consistent with the latest version of R in main memory.

(c) *cache-memory:* at region Access time, the region R in C_i is made consistent with the copy of the region in M_i^R.

Condition (a) and (c) are rather strong, but they simplify main memory coherence algorithms considerably by allowing cache coherence and main memory coherence algorithms to be independent of each other. For instance, without condition (a), it may be the case that the latest version of a region does not exist anywhere in main memory when main memory coherence mechanisms are initiated. Without condition (c), the memory coherence algorithms may be restricted to making all replicas of regions consistent at Done time, depending on the cache coherence algorithm.

3 Region-Oriented Cache Management

3.1 Algorithms for region-oriented cache management

An implementation of region-oriented cache management requires three basic cache control primitives that operate on contiguous regions. Non-contiguous regions require multiple calls to the same cache control function. The three primitives are:

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

For *write-through* caches, both the *cache-copyback* and the *cache-flush* operations become unnecessary. For the purpose of generality, however, we will assume that the cache is not write-through. Additionally, we assume that regions are aligned to cache line boundaries; the issue of false sharing within cache lines poses a problem for most software cache management techniques and is not addressed in this paper.

Two algorithms for region-oriented cache management are shown in Table 1. These are the Cache-Flush algorithm and the Cache-Validate algorithm. In the Cache-Flush algorithm, coherence is enforced at region Done time by invalidating the region in the cache at ReadDone time, and flushing the region at WriteDone time. Using the Cache-Validate algorithm, the region remains in the cache at Done 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.

Access Control Primitive	ROMM Cache Management Algorithm	
	Cache-Flush	Cache-Validate
ReadAccess or WriteAccess	-	if status[MyCache] == HI cache-invalidate(region) status[MyCache] = HV
ReadDone	cache-invalidate(region)	-
WriteDone	cache-flush(region)	cache-copyback(region) foreach Cache $\neq$ MyCache & status[Cache] == HV status[Cache] = HI

Table 1: Region-Oriented cache management strategies.

The Cache-Flush algorithm is simple to implement and has the advantage that the actions of each cache are independent of others, thus allowing for greater scalability. However, in many instances, a region is unnecessarily invalidated from the cache using this algorithm. Invalidations from processor P_i are *necessary* only when P_i accesses region R, and P_j $(j \neq i)$ modifies R before P_i accesses it again. In all other cases, the region is unnecessarily invalidated from the cache and incurs both the cost of performing the invalidation and the cost of reloading the region back into the cache. Thus, the Cache-Flush algorithm is expected to perform well only if the sharing patterns of the application are such that necessary invalidations are far more common than *unnecessary* invalidations.

The Cache-Validate algorithm performs only necessary invalidations and thus has an advantage over the Cache-Flush algorithm. However, in cases where an invalidate after a WriteDone operation was necessary, the Cache-Flush algorithm performs both the cache-copyback and the cache-invalidate as a single cache-flush operation at WriteDone time, whereas the Cache-Validate algorithms performs a cache-copyback at WriteDone and a cache-invalidate at Access time. Since doing these two operations separately is inherently more expensive than a single cache-flush (see Table 3 in Section 5.1), performance for the Cache-Validate algorithm will degrade if such cases are common.

To implement the Cache-Validate algorithm, a state vector is associated with each region. This state vector contains information about the current state of all caches in the system for a particular region. The possible states are:

- NH : This cache does *Not Have* a copy of the region.
- HV : This cache *Has* a *Valid* copy of the region.
- HI : This cache *Has* an *Invalid* copy of the region.

For either Read or Write Access calls, the region's cache state vector is checked. If the current state of cache C_i for region R is HI, R has been modified by another processor since the last Done operation by processor P_i. Thus, R must be invalidated from C_i before P_i may continue. For a WriteDone operation on R by P_i, the state of all caches (except C_i) in R's state vector for which the current state is HV is set to HI to indicate that the region has been modified.

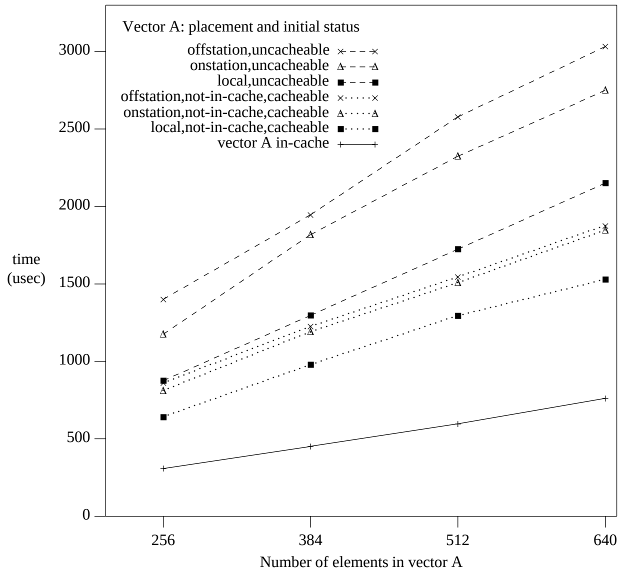
3.2 The benefits of region-oriented cache management

Region-oriented cache management algorithms can be easily and efficiently implemented in software. These techniques have two important advantages over other software cache management techniques:

1. In ROMM, shared regions are *always cacheable.*

2. Using the Cache-Validate algorithm, regions are never ejected from the cache unless they have become invalid. Thus, this technique may offer high cache reuse.

A high cache reuse rate is critical to performance. However, even when data in the cache is not reused, the ability to cache that data may be a significant advantage in ROMM. *Cacheability* for a variable implies that keeping that variable in the local cache and performing read or write operations on it will not compromise consistency. Static program analysis techniques may be unnecessarily pessimistic when determining when shared data is cacheable. Page-based run-time cache management techniques disable caching whenever false or real read-write sharing occurs. Figure 3 illustrates the impact of cacheability on HECTOR, a hierarchical shared-memory multiprocessor




```
Test Code
float A[N];

for (i=0; i<; i++)
      A[i] = A[i] + q;
```

```
Pseudo-Assembly Code
loop:
  load A[i]
  add A[i],q
  store A[i]
  inc i
  exit i=N
  goto loop
```

Figure 3: Measurements of the impact of caching and cacheability on HECTOR.

(described in Section 5.1) in which four processor-memory pairs comprise a station [VSLW91], and multiple stations are connected by a hierarchy of rings.

In Figure 3, the array A is accessed and updated. The time required to update A when A is in-cache and when A is not in the cache at the beginning of the computation is compared. If A is not in the cache, it's placement may be: a) in local memory, b) in a remote memory on the same station (on-station), or c) in a remote memory on a different station (off-station). Additionally, a comparison is made between the cases when A is cacheable, and when A is not cacheable according to the coherence policy. The results are somewhat surprising. Having A remote but cacheable appears better in this particular case than having it local but uncacheable, even though the data is used only once. There are two reasons why cacheability makes a difference:

1. Cache lines: cache line fills can be used when data is cacheable but single word fetches must be used when data is uncacheable.

2. Write buffering: with ROMM, the store operations in the above example are done in cache, and the whole vector would be copied back to memory at some later time. This is beneficial if the hardware supports efficient block copyback or if the region is updated multiple consecutive times from the same processor.

In comparison to some of the hardware directory-based schemes that have been proposed [Ste90], there are several reasons why ROMM may be preferable:

1. Memory requirements: A common hardware directory solution is to associate cache status bits with each memory block [LLG+90]. The memory requirements for such a scheme is on the order of MC, for C caches and a main memory size M. In the ROMM Cache-Flush

scheme, no additional bits of memory are required. For the ROMM Cache-Validate scheme, two bits for each cache are required per region, with a total space overhead of $2RC$ for R user-defined regions. It is expected that the number of user-defined regions is much less than the number of memory blocks in the system (i.e., $R \ll M$). Thus, the space overhead for this algorithm should be comparatively small.

2. Cost of invalidation: For a region of size S and cache line size L, a directory-based scheme requires $\left\lceil \frac{S}{L} \right\rceil$ invalidation messages to each of the P processors with valid copies of the shared data. With the ROMM Cache-Flush algorithm, no communication with other processors is required. With the Cache-Validate algorithm, the state vector must be altered, but this does not require communicating with other caches.

3. Serialization: Directory schemes using a write-invalidate scheme require invalidation of all caches to be performed at the time of a write operation. In ROMM, the invalidation of the regions is performed independently by each processor before or after it accesses that region.

4 Region-Oriented Main Memory Management

4.1 Algorithms for main memory management

Using region-oriented memory management, locality in main memory may be improved by replicating regions to memory local to the accessing processors. Coherence is maintained at region granularity at region access control points in the application. There are two issues in NUMA main memory management: a) when and where to replicate regions, and b) how to maintain coherence of replicas. Currently, regions that are identified by the application as candidates for replication are replicated to memory local to the accessing processor at Access time. Automatic techniques for determining when and where to replicate regions are desirable, but they are investigated in this paper. We focus instead on techniques for maintaining coherence.

As with region-oriented cache management, several algorithms for main memory coherence are possible. Consider, for example, the two algorithms shown in Table 2. An implementation of these algorithms requires two conceptual operations:

- *push-update*(region, j): a calling processor P_i with an up-to-date memory copy of a region at M_i initiates the update of an out-of-date replica of the region at memory M_j.
- *pull-update*(region): a calling processor P_i with an out-of-date replica of the region use this operation to make its own replica up-to-date. For regions comprising multiple pages, an optimized implementation of this operation may defer the update of each page of the region to when the page is accessed.

In the Region-Update algorithm, replicas of a region are *push-updated* at the time of a WriteDone operation, while the Region-Invalidate algorithm uses invalidation at WriteDone time, and enforces coherence at Access time. With the Region-Invalidate algorithm, a replica state vector similar

Access Control Primitive	ROMM Memory Management Algorithms	
	Region-Update	Region-Invalidate
ReadAccess or WriteAccess	-	if status[MyReplica] == HI pull-update(region) status[MyReplica] = HV
ReadDone	-	-
WriteDone	foreach region-replica push-update(region)	foreach region-replica RR $\neq$ MyReplica status[RR] = HI

Table 2: Region-Oriented main memory management strategies.

to the cache state vector used for the Cache-Validate algorithm is associated with each region. The replica state vector indicates the status of each replica of the region in main memory, and invalidation is performed simply by setting the entry for a particular replica in this vector to HI.

The choice of algorithm for a particular region will depend on the access patterns to that region, since each of these algorithms has its benefits and its limitations. For instance, the Region-Update algorithm has the advantage that replicated regions are available in local memory at Access time. Additionally, if there are many processors requesting concurrent read access to a region, there may be considerable contention for the region after a WriteDone operation. If an efficient implementation of push-update is used (such as one using logarithmic broadcast of shared regions), the cost of updating regions may be reduced considerably.

The Region-Invalidate algorithm may be preferable in many cases since, in the Region-Update algorithm, replicas may be updated unnecessarily. In our current implementation of ROMM, the user is able to specify which main memory coherence algorithm to apply to a particular region.

4.2 A simple Region-Update implementation

A simple implementation of Region-Update was done in HURRICANE [SUK92], an operating system developed for the HECTOR multiprocessor [VSLW91]. HURRICANE provides basic support for page-level replication and consistency management. Because there is no hardware support in HECTOR for replicating individual regions, we adopted a hybrid page-region implementation which uses the pages as the basic containers of regions, but maintains consistency at region granularity.

The default HURRICANE mechanisms for triggering replication and maintaining consistency were disabled, and support for ROMM consistency management was added. When replication for a particular region is required, a call is made to HURRICANE at the time of an Access operation to replicate the page(s) on which the region exists. For consistency management, the boundaries of the region are sent as parameters to a HURRICANE system call, and HURRICANE initiates the push-update of all out-of-date replicas of the region. The push-update implementation copies portions of pages specified by the region boundaries to all replicas of the pages on which the region lies.

This implementation strategy was chosen for its simplicity, and it allowed us to test the performance benefits of ROMM memory management on different applications with minimal effort. However, the use of pages as the unit of replication introduces some undesirable side-effects, such as the potential for accidental replication of another region on the same page as the one for which replication is called. Our current solution is to ensure that only regions with the same sharing patterns are allocated on the same page. We are working on a purely region-based implementation.

4.3 The benefits of region-oriented main memory management

Region-oriented main memory management has several advantages compared to traditional NUMA memory management schemes:

1. Since consistency is maintained at region granularity, the false sharing of pages should not be a concern. Our current implementation does not entirely overcome the false sharing limitation since pages are still used for replication, but future purely region-oriented implementations should.

2. The cost of maintaining consistency is lower with ROMM since only the modified region needs to be updated. Updating whole pages implies that more data is being transferred than may be required.

3. The access control to regions permits the use of relaxed consistency protocols for managing replicated data, such as the two algorithms that were discussed, as a means to improving performance.

4. Policies in traditional NUMA memory management schemes are triggered by page faults. With ROMM, replication and consistency management may be triggered by the access control points in the application. This gives greater flexibility in the choice of replication strategies.

5. Different replication algorithms can be used for different regions. The replication policies may be based upon measured access patterns to the regions or on user (or compiler) supplied hints.

5 Experimental Results

5.1 System architecture and environment

Implementation and experimentation of ROMM was done on HECTOR, a shared-memory multiprocessor developed at the University of Toronto [VSLW91]. Each processor of HECTOR is a 16 MHz MC88100 CPU with separate 16 Kbyte data and instruction caches, and 4 Mbyte local memory. HECTOR is hierarchical in design; several processor-memory pairs on a bus comprise a station, and multiple stations are connected by a hierarchy of bit-parallel rings. Our experiments were conducted on a 16-processor, 4-station configuration. Figure 4 shows the organization of HECTOR, as well as representative read latencies (for a 16-byte cache line fill) for local, on-station, and off-station memory units. It is worth noting that, in the absence of contention, the "NUMAness" of

HECTOR is mild compared to other NUMA multiprocessors [BBN88, LLG+90]. Thus, the benefits of localizing and caching data shown in our measurements would likely be greater in other NUMA architectures.

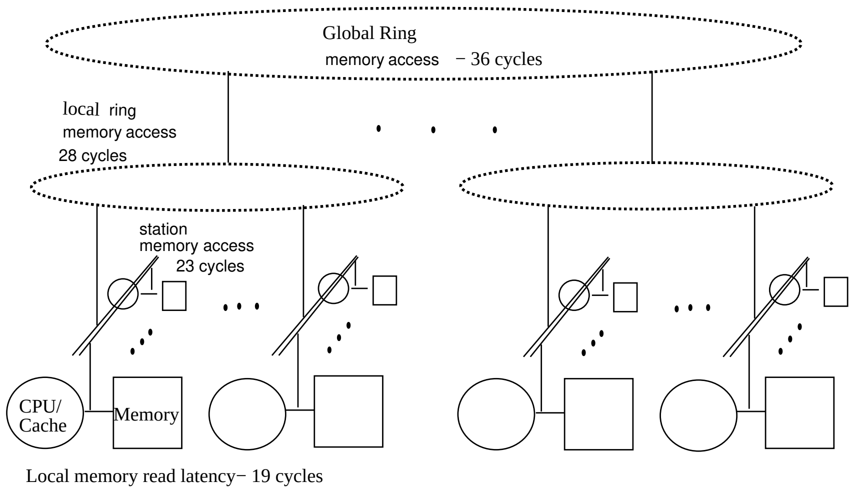


Figure 4: HECTOR: A hierarchical shared-memory multiprocessor.

HECTOR runs a locally developed operating system called HURRICANE [SUK92]. Cache coherence is enforced by the system memory manager at page granularity, by allowing only unshared and read-shared pages to be cacheable. HURRICANE supports replication and migration of pages, but disables replication when a page is write-shared.

For our implementation of ROMM, we disabled the standard HURRICANE cache and memory coherence mechanisms and used the ROMM cache and main memory management techniques described in earlier sections. Shared data that uses the region abstraction is allocated in a separate memory pool, and coherence of this pool is controlled through region access control operations. 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 data management facilities.

To assess the costs of using region-oriented memory management, we measured the time to perform the primitive cache and memory operations used by ROMM. This is shown in Table 3 for various region sizes. The costs of cache line invalidation is negligible, whereas those for copying cache contents back to main memory is low but significant, especially when the region-mapped memory is non-local. The cache operation costs have to be incurred even without the use of regions, and ROMM cache management may avoid many of the cache operations, such as unnecessary invalidations and subsequent reloading of the cache, using Access and Done information. The memory update costs are significantly higher than cache operations, suggesting that suggesting that main memory management techniques should only be applied to data that will be repeatedly accessed and for which caching is not effective.

The costs in Table 3 are not linear with the size of the region being operated upon, due to operating system call and other fixed overheads. The benefits of using regions may be offset by region overheads if regions are defined on data shared in fine grains (e.g., cache line sizes), and when Access and Done operations are performed frequently. For the applications to be discussed below, this has not been a problem.

Size	Cache Invalidate	Cache Copyback/Flush		Memory Push Update	
		Local Memory	On-Station	One Replica	Two Replicas
1 cache line	19	22	26	930	965
10 cache lines	29	41	66	990	1100
1K bytes	27	129	283	1350	1770
4K bytes	28	425	1035	2550	4170

Table 3: Timings for primitive Region operations (in microseconds).

5.2 Experiments

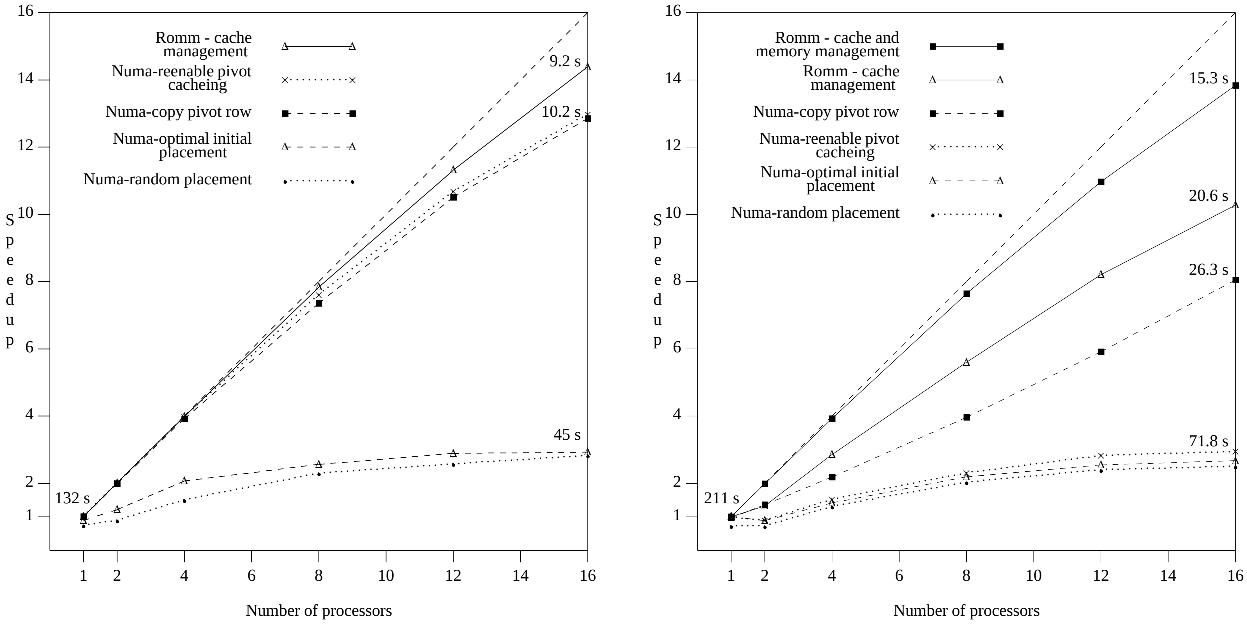
We used a few sample applications to assess the performance of ROMM. For each application, we implemented two versions based on a naive uniform memory access (UMA) implementation of the application. The version with shared region specifications, referred to as the ROMM version relies on ROMM cache and (optionally) main memory management. For the version without region specifications, called the NUMA version, we began with the naive UMA program and then tried various hand-coded optimizations on the standard HURRICANE system to improve performance. For each application, we discuss the optimizations for the NUMA version that lead to significant improvements in performance, and compare these with the ROMM version.

5.2.1 LU-decomposition

Row-oriented LU-decomposition decomposes a dense N×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. At each phase of the computation, the pivot row is reduced by the processor that "owns" it, and the pivot row is then used by all processors to reduce their own rows.

Figure 5 shows the results of two sets of experiments, for LU-decomposition using matrix sizes 512×512 and 600×600. These are referred to as the 512-case and the 600-case. (In these and all subsequent figures, speedup curves are annotated with response times for the optimal sequential case and the 16-processor cases). In the 512-case, each row of 512 double precision floating-point numbers occupies exactly one HECTOR virtual memory 4 Kbyte page and the rows are page-aligned. In the 600-case, the rows are also contiguously allocated but are not page-aligned.

The naive UMA version is not shown in Figure 5 because it resulted in a slowdown. To improve performance, the first step is to reduce memory contention by distributing the pages of the matrix



a) 512×512. b) 600×600.

Figure 5: LU-decomposition on 512×512 and 600×600 matrices.

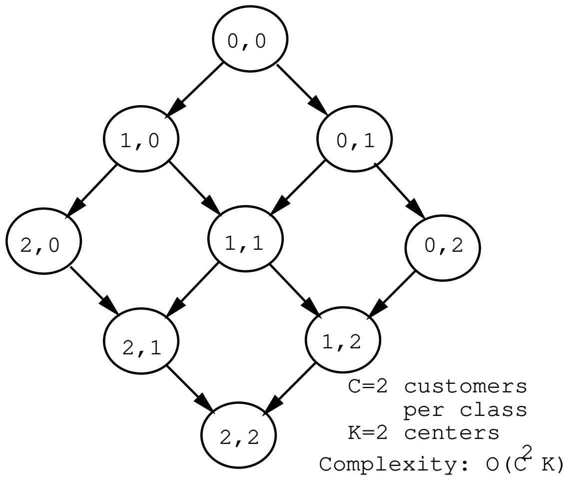
uniformly across all memory units. This results in moderate speedups for both the 512- and the 600-cases. Next, data locality is improved by allocating the pages containing the rows onto the local memories of their owners (optimal initial placement). In the 512-case, all of the rows are completely localized. In the 600-case, however, the mis-alignment of rows on pages reduces the amount of memory locality that can be achieved in this fashion. Surprisingly, however, little improvement results even with the 512-case. This is because the pivot row, once reduced, is not cacheable according to the write-sharing protocol of HURRICANE, and the pivot row accounts for half of all accesses to the matrix A. Thus, as the next optimization, an explicit call is made to HURRICANE to re-enable caching for the page on which the pivot row exists, after this row has been reduced. This yields reasonable performance for the 512-case, but no real improvement for the 600-case, where false sharing due to the mis-alignment of rows on pages makes them remain uncacheable. As a final optimization, the pivot row is manually copied to each of the accessing processors after the pivot reduction. This does not help the 512-case much (because pivot caching was already re-enabled), but does help the 600-case significantly. Note that explicitly copying the pivot row is a deviation from the shared-memory programming paradigm, and may only work in restricted cases such as LU-decomposition.

The first ROMM version uses the Cache-Validate algorithm for cache management and tries to place each row local to the process that owns it. This version performs better than all of the hand-coded optimizations for the NUMA case, and shows the enormous benefits of good automatic cache management. The speedup for the 512-case is close to linear. For the 600-case, the rows cross page boundaries, and when parts of two rows exist on the same page, they can only be local to one of the two processors sharing it. Thus, in the second ROMM version, replication of the rows (regions) is used, ensuring that that the processor modifying a row will have a completely local copy of that row. For the 600-case, this resulted in performance comparable to the 512-case.

The results for LU-decomposition show a clear need for using a region-oriented approach to memory management. There are two important observations: a) a UMA program annotated with region specifications and using region-oriented memory management may give significantly better performance than an optimized NUMA version of the same program, and b) whereas many previous studies of speedups of various algorithms used data sizes resulting in perfect page alignment, ROMM is able to provide equally good performance for non page-aligned data.

5.2.2 Mean value analysis (MVA)

MVA calculates mean performance measures such as average customer response times and queue lengths for a 2-class closed queueing network. We use the algorithm proposed by Reiser and Levenburg [RL80]. This algorithm 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 $(n_1, n_2 - 1)$. Therefore, solving a system with C customers in each class requires first solving all systems with fewer customers. Figure 6.a shows the task execution graph for the case of $C = 2$. Each task requires the queue length vectors from its two parent tasks in order to proceed. Figure 6.b shows a skeleton of the code for the task (n_1, n_2), together with the six lines of region access control code.



a) Task execution graph.

```
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();
}
```

b) Skeleton code.

Figure 6: Parallel mean value analysis.

We adapted a parallel implementation of MVA by Zoharjan and McCann [ZM90] for large populations by creating a thread for each left-inclining row of tasks (those with the same n_2 value). Figure 7 shows the performance of the NUMA and ROMM versions of the program for a system with $C = 200, K = 400$. For this system, a queue length vector is 1600 bytes. Since each occupies only about 1/3 of a page, multiple vectors become dynamically allocated onto the same page. As is the case with LU-decomposition, false sharing makes the vectors uncacheable in the NUMA version. After annotating the program with region Access and Done operations for each of the queue length vector (see Figure 6.b), all the data becomes cacheable, yielding significant performance improvement. This is especially interesting given that each task reads the queue length vector of each of its parents only once, and has only a 50% chance of being able to use its parents' cached data. Because of the pair-wise sharing characteristics and the effectiveness of caching in MVA, main memory locality is not a significant issue, and only ROMM cache management is used.

It is worth noting that, for MVA and the other applications we have experimented with, annotating the program with region specifications was relatively simple, and the changes to the application program are minimal.

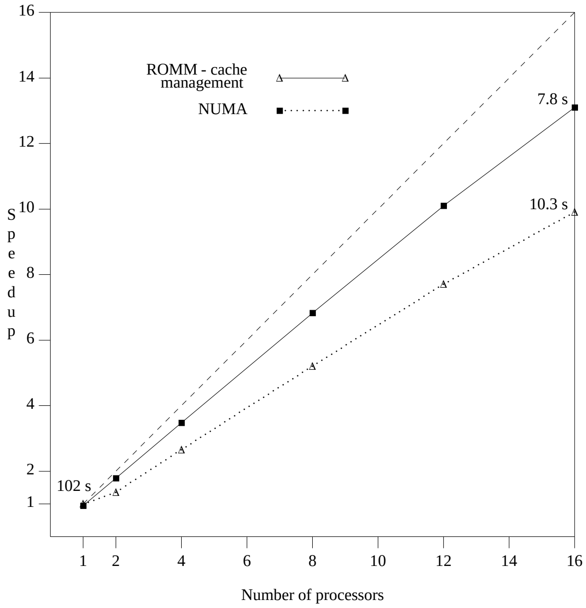


Figure 7: MVA for $C = 200$, $K = 400$.

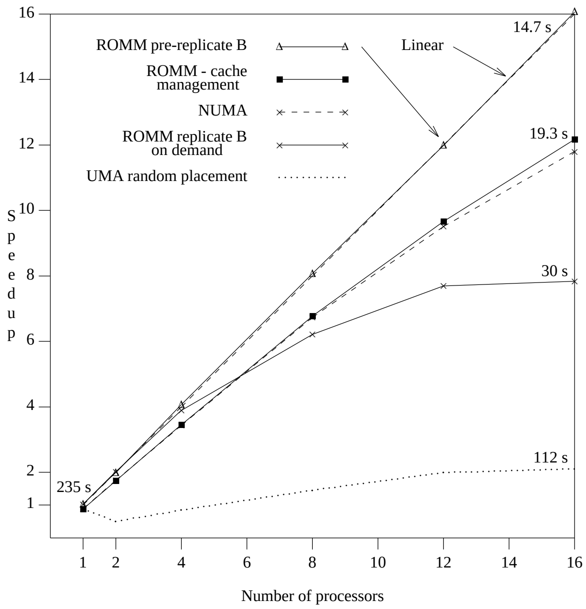


Figure 8: Matrix multiply: 460x460 matrices.

5.3 Matrix Multiply and other applications

Multiplication of two matrices A and B to produce a result matrix C is done in parallel by assigning to each processor a group of rows of the matrices A and C. the whole B matrix is read-shared by all processors. Figure 8 shows the results of matrix multiply on 460×460 matrices. We again compare the NUMA version with a ROMM version of the program.

In the NUMA case, each processor was assigned a group of contiguous rows of the C matrix, in order to reduce the effects of false sharing. Unfortunately, when the row size is not a multiple of the page size, parts of the first and last rows in each group cross page boundaries and end up on the same pages as the perimeter rows of other processors. Pages for which such overlap occurs end up uncached due to the HURRICANE coherence mechanism. However, since these are only perimeter rows, the impact on performance is not substantial. Thus, although the ROMM version with using random data placement and region-oriented cache management improves performance over this

NUMA version, the difference is small.

The next ROMM version replicates the rows of B on demand, in order to improve locality. For matrices A and C, memory locality is not needed because of good cache performance. While replicating B leads to better performance for a small number of processors, for six or more processors, sending replicas of B from one to all fo the other processors causes the sending processor to become a sequential bottleneck and causes performance to degrade. In the pre-replicate ROMM case, in which the replication of B was performed before the computation started (analogous to optimal initial placement of data), we were able to achieve linear speedup. This result shows that, in the current ROMM implementation, the only bottleneck for this application is the cost of replication. This cost may be reduced by using more intelligent replication strategies (for instance, logarithmic broadcast).

It is also worth noting that, in the ROMM version, any strategy for assigning rows to processors may be used without a performance penalty. For example, we assign rows of C to each processor in a round-robin fashion. In the NUMA version, a block of contiguous rows had to be allocated to each processor in order to minimize the impact of false sharing. Thus, ROMM provides greater flexibility in the choice of algorithms by relieving the user of concerns about false sharing.

We are currently experimenting with region-oriented memory management techniques for a variety of applications. Experiments thus far with other applications such as traveling sales person, merge sort, and fast fourier transform have all shown favorable results for the ROMM approach.

6 Related Work

An application-oriented approach to data management has been applied to Distributed Shared Memory (DSM) systems by Bennet *et al.* (Munin) [CBZ91] and Bryant *et al.* (Structured Shared Virtual Memory – SSVM) [BCaCR92]. In Munin, the user explicitly specifies the shared data objects, and how they are shared. Munin then applies different coherence mechanisms for different types of shared objects, choosing the optimal coherence strategy for a predetermined class of sharing behaviour. In ROMM, on the other hand, data sharing behaviour does not need to be identified in the application, but the specification of regions is essential. In SSVM, a synchronization primitive is provided that allows the user to specify when a data object is to be flushed from the cache.

There is an obvious distinction between cache and memory management mechanisms for Shared Memory Multiprocessors and Distributed Shared Memory systems. The communication latencies in DSM are such that greater overhead in the software coherence policies can be tolerated. For example, in Munin, false sharing of objects is avoided by allowing objects that have been declared as write shared to be concurrently updated, and then later merging the pages together using a byte-by-byte comparison. Such a technique would not work well on a shared-memory multiprocessor where accessing uncached shared data is not as expensive and the granularity of sharing is likely to be finer. In ROMM, false sharing is avoided by managing data at region granularity. Further, in DSM, there is no choice but to replicate or migrate all shared data since only a local copy of the data is directly addressable. In shared-memory multiprocessors, replicating or migrating data is only a performance optimization.

7 Conclusions

In this paper, we discussed techniques for region-oriented memory management (ROMM). We introduced the notion of *shared regions* in an attempt to characterize the natural granularity of data sharing that occurs in parallel applications. We then discussed how memory coherence techniques may exploit this concept for applications annotated with region specifications. Region-oriented techniques provide an effective solution to the *multi-level memory coherence* requirement, which encompasses cache coherence, main memory coherence, and the interaction between levels of the memory hierarchy in order to maintain memory coherence.

We discussed algorithms for region-oriented cache management. These algorithms can be implemented easily and efficiently in software. Region-oriented cache management has the advantage over other software cache management techniques in that it can be applied to any parallel programming paradigm with the addition of region specifications. Additionally, with ROMM, shared regions are always cacheable, so that performance may be significantly better for applications where cacheability of shared data is a critical factor. Compared with hardware directory schemes, ROMM cache management induces very little memory overhead, and less coherence traffic since coherence is enforced at region granularity and does not require communicating with other caches in the system.

We also discussed region-oriented techniques for main memory management. ROMM memory management has the advantage over traditional NUMA memory management in that, since shared data is managed at application-specified region granularity, the false sharing of data on hardware determined fixed-sized pages is eliminated. This allows ROMM to *localize* shared data in cases where traditional page-based schemes cannot, and it reduces the cost of maintaining consistency since updates are also performed at region granularity. Additionally, because of the access control applied to regions, ROMM has the ability to use relaxed consistency models and a number of different replica management algorithms, including *Region-Update* and *Region-Invalidate*, analogous to the *write-invalidate* and *write-update* algorithms applied in hardware cache coherence schemes.

Experiments with ROMM on HECTOR, a hierarchical shared memory multiprocessor. Our results show that ROMM programs perform comparably to hand-tuned versions of the same program in cases when data can be page-aligned, and better when data cannot be page-aligned. Additionally, the reduced sensitivity of page-alignment issues in ROMM gives more flexibility to the implementation of a particular algorithm. For instance, in matrix multiply, rows can be assigned to processors in any fashion without a performance penalty, whereas, without ROMM, programs must assign groups of contiguous rows to each processor in order to achieve the best results.

Research dealing with a number of issues in region-oriented memory management is currently in progress. This includes 1) an implementation and measurement study of hybrid page-region oriented approaches for memory management as well pure region-based approaches, 2) a performance study of ROMM in large shared-memory systems, and 3) a study of various optimizations on region-specified programs. Optimizations include prefetching regions, intelligent initial placement of regions, and replication strategies for regions in large shared-memory systems.

8 Acknowledgements

We are indebted to the Hector and Hurricane groups for their work, most notably: Michael Stumm, Ron Unrau, Orran Krieger, and Ron White. This paper would certainly not have been done without their support. Thanks also to Tim Brecht, Brian Carlson, and Michael Stumm for reviewing the paper and giving many useful comments.

References

[ABLL91] Thomas E. Anderson, Brian N. Bershad, Edward D. Lazowska, and Henry M. Levy. Scheduler Activations: Effective kernel support for the user-level management of parallelism. In *Proceedings of 13th ACM Symposium on Operating Systems Principles*, pages 95–109. Association for Computing Machinery SIGOPS, October 1991.

[ASHH88] Anant Agarwal, Richard Simoni, John Hennessy, and Mark Horowitz. An evaluation of directory schemes for cache coherence. In *15th International Symposium on Computer Architecture*. ACM, May 1988.

[BBN88] BBN. Butterfly parallel processor overview. Technical Report No. 6148, BBN Laboratories, Cambridge, Mass., 1988.

[BCaCR92] Ray Bryant, Paul Carini, Hung ang Chang, and Bryan Rosenburg. Supporting structured shared virtual memory under Mach. In *Usenix Mach Symposium*. Usenix, January 1992.

[BSF+91] W. J. Bolosky, M. L. Scott, R. P. Fitzgerald, R. J. Fowler, and A. L. Cox. NUMA policies and their relation to memory architecture. In *Fourth Conference on Architectural Support for Programming Languages and Operating Systems*. ACM, April 1991.

[CBZ91] John Carter, John Bennet, and Willy Zwaenepoel. Implementation and performance of Munin. In *Thirteenth Symposium on Operating Systems Principles*. ACM, October 1991.

[JEK91] Richard LaRowe Jr, Carla Schlatter Ellis, and Laurence S. Kaplan. The robustness of NUMA memory management. In *Thirteenth Symposium on Operating Systems Principles*. ACM, October 1991.

[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.

[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.

[ZM90] J. Zahorjan and C. MaCann. Processor scheduling in shared memory multiprocessors. In *Proc. ACM SIGMETRICS Conference*. ACM, May 1990.