MULTIPLE-WRITER ENTRY CONSISTENCY [*]

HARJINDER SANDHU [†], TIM BRECHT [‡], AND DIEGO MOSCOSO [§]

Abstract.
In this paper, we present the design, implementation and evaluation of a new distributed shared memory (DSM) coherence model called multiple-writer entry consistency (MEC). MEC combines the efficient communication mechanisms of Lazy Release Consistency (LRC) with the flexible data management of the Shared Regions [17, 11] and Entry Consistency (EC) models [5]. This is achieved in MEC by decoupling synchronization from coherence (in contrast to the tight coupling of synchronization and coherence present in EC) while retaining the familiar synchronization structure found in Release Consistent (RC) programs. The advantage of MEC is that it allows region-based coherence protocols (those that manage data at the granularity of user-defined shared regions) to be used along side page-based protocols within an application and within the RC framework. Our experimental evaluation on an 8 processor system shows that using MEC reduces parallel execution times by margins ranging from 5% to 46% in five of the six applications that we study. However, the parallel execution time of the LRC version of the remaining application is lower than the MEC version by 48%. We conclude that offering both page-based and region-based models for coherence within the same system is not only practical but necessary.

Key words. parallel programming, network of workstations, distributed shared-memory, coherence models, memory consistency.

AMS subject classifications. 68-04, 68N99, 68M20.

1. Introduction.

Within the realm of Distributed Shared Memory (DSM) systems, several divergent streams of models have emerged. Among the most common are (1) those that rely on pages as the underlying abstraction for data management and use Release Consistency (RC) as the synchronization model presented to the user [7, 13], and (2) those that use user-defined *shared regions* as the underlying abstraction for data management coupled with a synchronization model called Entry Consistency (EC) that explicitly binds synchronization primitives with the data they protect [5, 11, 16]. Each of these models has their limitations. In RC-based systems, the reliance on pages as the unit of data management often causes these systems to transmit more data than necessary in order to maintain coherence due to the effects of false sharing, or to transmit data using more messages than necessary when shared data structures span multiple pages. Sophisticated coherence protocols such as *lazy release consistency* (LRC) reduce the impact of false sharing but cannot eliminate it entirely [13]. In EC-based systems, the problems that stem from the mismatch between the system's view of sharing and the granularity of sharing within the application are avoided entirely by managing data at user-defined granularity. However, the EC synchronization model is more complex to program than RC and is often avoided as a result.

Ideally, a system should integrate both page-based and region-based protocols in a way that permits the programmer to begin with the simpler RC-based model,

[*] Appears in Parallel and Distributed Computing Practices (PDCP), NOVA Science Publishers, USA, Volume 2, Number 2, pp. 77-85, June 1999. This work was supported by the Natural Sciences and Engineering Research Council of Canada

[†] Department of Computer Science, York University, North York, Ontario, Canada M3J 1P3. email: hsandhu@cs.yorku.ca

[‡] Department of Computer Science, University of Waterloo, Waterloo, Ontario Canada N2L 3G1. email: brecht@cs.uwaterloo.ca

[§] Department of Computer Science, York University, North York, Ontario, Canada M3J 1P3. email: diego@cs.yorku.ca and then add EC annotations for just those data structures that are better suited to region-based rather than page-based data management. Unfortunately, RC and EC are difficult to integrate in this way due to the differences in their respective synchronization models. Adding EC annotations in an existing RC program requires modifying the program's synchronization structure and it may result in a program that contains too much synchronization. The study of LRC versus EC by Adve *et al.* [1], for instance, points out that their EC-based applications often suffered from this problem of over-synchronization.

In this paper, we present a new model called Multiple-Writer Entry Consistency (MEC). MEC manages data at a granularity defined by the user but, unlike EC, it decouples the primitives used for synchronization from those used for coherence. This decoupling of synchronization from coherence has two effects. The first is that it permits the efficient integration of both page-based and region-based protocols for coherence within the same RC framework. For synchronization, MEC retains the familiar *acquire* and *release* primitives of RC. Consequently, users can write programs for the simpler RC model, using an efficient page-based protocol such as LRC, and then, within the same synchronization framework, selectively add MEC annotations to those data structures that may benefit from region-based data management.

The second effect of decoupling synchronization and coherence in MEC is that user-defined shared regions can be concurrently modified by different processors. This presents greater flexibility than the single-writer multiple-reader behavior enforced by EC on regions defined by the user. Consider, for example, an application with interspersed fine-grained sharing among the elements of a large data structure. Under typical EC implementations, either each element would have to be bound to a separate synchronization object and guarded individually, or else the entire structure would have to be bound to a single synchronization object and guarded in a way that permits only one processor to modify it at a time. The former partitioning would be too fine-grained to be efficient in a DSM environment, while the latter solution would unnecessarily restrict concurrency.

We have implemented the MEC protocol within the Treadmarks distributed shared memory system (which itself implements page-based LRC) and conducted a detailed evaluation that compares the performance of LRC and MEC for six applications. The results of this investigation show that five of the six applications obtain better performance using MEC than LRC, with differences ranging from 5% to 46%. In the remaining application, however, there is a significant degradation (48%) using MEC. These results, which we consider to be among the key contributions of this paper, argue strongly in favor of using an integrated page-based and region-based solution as a platform for distributed shared memory.

An overview of this paper is as follows. Section 2 briefly describes the background to this work with a description of the RC, LRC, and EC models. Section 3 describes the MEC model and its implementation. Section 4 presents the results of a performance evaluation of LRC versus MEC, and is followed in Section 5 with a discussion of how these results relate to other work, including earlier work comparing LRC and EC. Section 6 presents the conclusions of this paper.

2. Background. Memory consistency models present a set of programming constraints to the user and guarantee correctness only for programs that obey those constraints. In the Release Consistency (RC) model [9] these constraints are (a) that only system recognized synchronization operations are used, (b) that all synchronization operations are labeled as *acquire* or *release* operations, and (c) that there are no data races in the program. Under RC, a processor may delay the propagation of coherence related information to other processors until it arrives at a *release* point in the program. A release point in the program indicates that the process is exiting a critical section, and that other processors must now be made aware of modifications to shared data within this critical section. A lazy implementation of release consistency (LRC) [13], however, takes advantage of the fact that a processor that wishes to use the data computed by another processor must first acquire a lock from the processor that has modified that data. Therefore, using LRC the propagation of coherence related information can be delayed until the acquire operation by another processor, rather than immediately upon a release operation.

DSM implementations using RC mitigate the impact of false sharing on pages both by delaying the propagation of coherence information and by permitting multiple processors to concurrently modify different portions of the same page. Munin, among the first DSM systems to use RC, introduced the notion of *twinning* and *diffing* to consolidate concurrent changes to a page from different processors [7]. Prior to modifying a page, a processor creates a twin (a local copy) of the page. Later, upon arrival at a release point, it compares the modified page to its unmodified twin and transmits to other processors only those portions of the page that have changed. Treadmarks' implementation of LRC delays the creation and transmission of *diff*s until modifications to the page are requested by another processor, thereby further reducing the number of messages required to maintain coherence.

Entry Consistency (EC) [5], and other models like it, such as the Shared Regions (SR) model [11, 17], avoid the problem of false sharing among virtual memory pages by managing data at a granularity defined by the user.[1] In SR, a *shared region* is a user-defined entity describing the granularity at which data is shared within the application. According to both EC and SR, each shared region is explicitly bound in the program to a token that is used for synchronization. In SR, the primitives *readaccess*, *writeaccess*, *readdone*, and *writedone* are used to guard access to a shared region. In addition to enforcing synchronization, the underlying system checks the state of the region when these primitives are invoked and, when necessary, initiates coherence actions to keep the region consistent. Each user-defined shared region follows a single-writer multiple-reader synchronization protocol. The EC or SR models have been used in a number of systems, including Midway [5], CRL [11], DiSom [16], ABC++ [4], Amber [8], and Hurricane [17].

3. Multiple-Writer Entry Consistency.

The Multiple-Writer Entry Consistency (MEC) model described in this paper combines elements of both RC and EC in the interface that is presented to the user, and is akin to LRC in its implementation. This section describes the MEC model and its implementation.

3.1. Model.

As in the Shared Regions (SR) model [17], we use the term *shared region* in MEC to denote a user-defined region of memory that is treated as a single unit for the purposes of coherence enforcement. Once a set of shared regions has been defined within the MEC framework, a set of operations, *readaccess* and *writeaccess*, are used in conjunction with a particular shared region to indicate when references to that shared region occur, and whether these references also modify that region. We refer to these operations generically as *access* operations.

From the programmers perspective, the main differences between MEC and the

[1] For the purposes of discussion, we borrow terminology from both the EC and SR models; semantically, these models are equivalent.

EC and SR models are twofold. First, the access operations in MEC enforce coherence but are non-synchronizing. For synchronization, MEC retains the familiar *acquire* and *release* operations of RC, and integrates the coherence enforced through the *access* operations with the synchronization performed by the *acquire/release* operations. The second difference between SR and MEC is that, in MEC, although the access operations are used prior to a series of references to a particular region, there is no need for corresponding operations to indicate the completion of this series of references to the region.

In MEC, regions defined by the user can be modified concurrently by different processors. Access operations are used to determine when to consolidate changes to a region. An *access* operation by processor P_i ensures that all modifications to region R *preceding* the *access* operation are seen by P_i. The use of the term *preceding* in MEC is equivalent to the *happened-before-1* relationship defined by Adve *et al.* [2]. That is, x can be said to *precede* y if: (a) x and y are on the same processor and x occurs before y in the order defined by the program, or (b) x is a release operation on one processor and y is an acquire operation on another processor, and y returns the value written by x. This relationship is transitive, so if x precedes y and y precedes z, then x precedes z in this partial order. Thus, modifications to R by processor P_i will be seen by processor P_j only when there is a transitive chain such that the modifications by P_i precede an *access* operation by P_j.

3.2. Implementation. Our implementation integrates MEC into the Treadmarks implementation of LRC [13]. In the standard Treadmarks implementation, a large shared data space is created in the initialization of the system, and all shared data is subsequently allocated within this space. A page table is used to keep track of the state of each page in this shared data area. The execution of each processor is divided into time intervals (where synchronization operations mark the beginning and end of an interval) and each processor maintains a list of the intervals it has seen from each of the other processors. On an acquire operation, a processor transmits its current interval time stamp and receives in return a list of intervals it has not seen and *write notices* containing the set of pages that were modified in that interval.

In our implementation of MEC, the shared data area is partitioned into two sections, a page-based area containing K virtual memory pages, and a shared region-based area containing the remaining space. The page table is converted into a page-region table, partitioned so that the first K entries keep track of pages in the page area, and the remaining entries keep track of user-defined regions in the shared region area. Shared data may be allocated in either area from within the program. Data allocated in the page-based area is managed using LRC in exactly the same way as in the unmodified Treadmarks system, using virtual memory protection to intercept faulting read/write references to pages and initiate coherence actions. For data in the shared regions area, page faults do not occur. Instead, *access* annotations in the program are used to determine when coherence actions are needed for a particular region.

In the integrated LRC/MEC implementation, both pages and shared regions are referred to by their index in the page-region table. On an acquire operation, a processor transmits its vector time stamp to the processor currently holding the lock, just as it does by default in LRC, but it receives in response *write notices* containing a list of both pages and regions that have been modified in intervals not seen by the acquiring processor. In MEC, a region is considered modified in interval q by P_i if P_i issued a *writeaccess* on that region in interval q.

The *diff* and *interval* management routines are identical for both page-based LRC and region-based MEC except for the granularity at which these actions are carried out. In our implementation, changes were made to the original Treadmarks *diff* management routines to support arbitrary regions of data, but the interval management routines were not modified from the originals. We also converted the Treadmarks communication layer from UDP to TCP to facilitate the transmission of large data regions in MEC. Comparisons between LRC in the original (unmodified) Treadmarks system and our modified version showed that neither of these changes had a significant impact on performance.

4. Performance Evaluation. In order to assess the performance of MEC as well as the value of integrating both MEC and LRC within the same system, we compared the performance of six application kernels executed first using LRC, and then with key shared data structures identified as shared regions in order to use MEC. These applications are: Blocked Matrix Multiplication (MM), Successive Over-Relaxation (SOR), Blocked-Contiguous LU-Decomposition (LU), Traveling Salesman Problem (TSP), Integer Sort (IS), and Floyd-Warshall (FLOYD). TSP, SOR, and IS are from the suite of applications used in earlier TreadMarks studies [13, 14, 1], LU is from the Splash-2 benchmark suite [18], and MM and FLOYD were written locally.

All experiments were conducted on an 8 node cluster of workstations connected together by a Fore ATM switch. The workstations are IBM RISC System/6000s each with a 133 MHz PowerPC 604 processor, 16 KB L1 instruction cache, 16 KB L1 data cache, a 4-way associative 512 KB L2 cache, and 96 MB of memory. Virtual memory pages are 4 KB in size. Table 1 shows the problem sizes, execution times, and speedups of these applications running on this workstation cluster.

program	problem size	1 proc	8 proc time		speedup	
		time	LRC	MEC	LRC	MEC
MM	1536x1536 (int) matrices , block size 32	257.7	58.1	50.1	4.4	5.1
SOR	2000x2000 (float) matrix, 100 iterations	108.5	34.9	18.8	3.1	5.8
LU	1536x1536 (double) matrix, block size 64	97.9	35.8	21.2	2.7	4.6
TSP	19 city tour	56.2	23.3	44.5	2.4	1.3
FLOYD	512 node graph	61.9	13.8	12.5	4.5	5.0
IS	2^{22} keys, bucket size 2^9, 10 iterations	24.9	4.0	3.9	6.2	6.3

TABLE 1
Problem sizes, execution times (in seconds), and speedups on 8 processors.

4.1. Summary of Results. Figure 1(a) shows the normalized parallel execution times of the applications used in this study. Execution times are divided into four components: (1) sync - synchronization time, (2) traps - time spent handling traps (protection traps to pages in LRC, *access* traps in MEC), which includes time spent fetching diffs and creating twins, (3) remote - time spent handling remote requests initiated from other processors, including time spent creating diffs on demand, (4) base - the time remaining, once sync, traps, and remote costs are accounted for. The base time is largely time spent performing computation in the application, although because trap and remote costs begin timing after asynchronously entering the signal handler, the base time will also include the cost of entering the signal handler. The accompanying graphs in Figure 1 show the number of messages communicated and the number of bytes transferred relative to one processor.

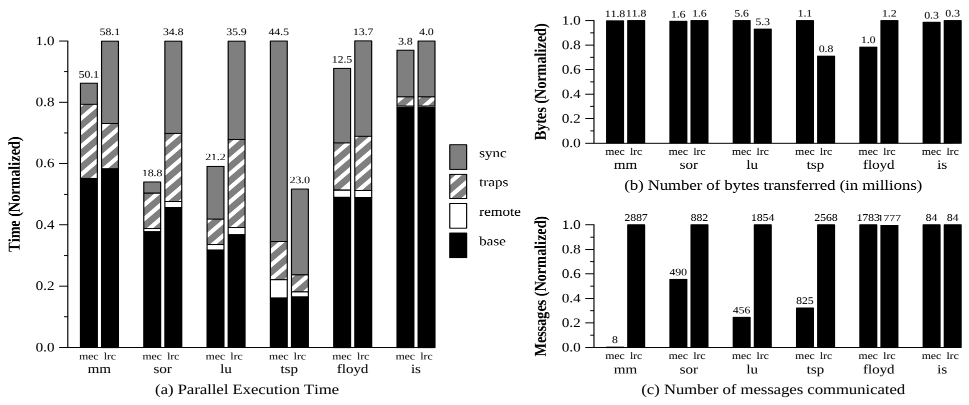

FIG. 1. *Eight processor performance of applications under LRC and MEC.*

In the MEC version of our applications, shared data structures were partitioned into shared regions in what seemed to be the most obvious and natural fashion for each application. For the problem sizes used, MEC executions times are lower than LRC in MM, SOR, LU, FLOYD, and IS, by margins ranging from 5% in IS to 46% in SOR (Figure 1(a)). For MM, SOR, and LU, this difference is due largely to a reduction in the number of messages (Figure 1(c)), while in FLOYD, the difference is due to a reduction in the number of bytes (Figure 1(c)). In TSP, MEC execution time is significantly worse than LRC, by a margin of 48%, due to the overly simplistic way regions are used in the MEC version. Here, the number of messages is reduced using MEC by defining a shared region that spans many pages, but false sharing within this shared region leads to a significant increase in the number of bytes transmitted.

The performance of MEC in these applications highlights the need for providing both page-based and region-based protocols within the same system. MEC requires more effort to program, and an improvement of 5% is not likely to justify this effort, whereas some of the larger improvements gained in other applications makes the corresponding effort appear justified. In order to provide insights into the underlying influences on the performance of MEC and LRC in these six applications, each of the applications is discussed in more detail below.

4.2. The Applications in Detail. **MM** implements a blocked matrix multiplication $C = A \times B$. The A and C matrices are partitioned into P contiguous sections among the P processors. In LRC the portions of A and C belonging to each processor, as well as all of B, are fetched one page at a time. B occupies 2304 pages, while the portions of A and C accessed by each processor occupy 288 pages each.[2] Fetching A, B, and C therefore requires on the order of 2800 messages in LRC. In MEC, on the other hand, each processor fetches exactly three regions, one containing matrix B, and two others containing the portions of A and C to be used by that processor, using a corresponding small number of messages. However, the advantage gained in MEC by reducing the number of messages is significantly hampered by increased contention that results from the size of the regions (9 MB for B and over 1 MB for each portion of A and C) and the fact that they are all accessed synchronously from

[2]The number of pages occupied by B can be calculated as the size of the matrix (1536 x 1536 x 4) divided by the size of a page (4096 KB), and the number of pages occupied by the portions of A and C assigned to each processor are 1/8th of this each.

one processor prior to beginning the computation. In the LRC version of MM, the interleaving of page requests with computation causes page requests to be distributed more randomly and therefore these requests experience less contention. Consequently, the dramatic reduction in the number of messages from several thousand to a handful in MEC results in a decrease in execution time of only 14%.

SOR uses a Red-Black Successive Over-Relaxation algorithm to solve partial differential equations. Sharing between processors occurs along the boundary rows of two $M \times N$ matrices. In the MEC version, a separate shared region is used for each row. For $N = 2000$, each row spans two pages. Rows of this size are page-aligned in LRC using the default Treadmarks memory allocation algorithm for page-based data, but unaligned in MEC using the memory allocation algorithm we implement for region-based data (where page alignment is unnecessary). The page-alignment in LRC means that false sharing plays no role in SOR's performance for this problem size. Most of the difference in execution time between MEC and LRC is due to the fact that each MEC region comprises two pages, so that the number of messages in MEC are half of those in LRC. Interestingly, using page-alignment in LRC degrades the cache performance of SOR, resulting in some loss in performance even before communication costs are accounted for. Not having page alignment in LRC, however, has more serious implications on performance (as discussed in Section 4.3).

LU decomposes a matrix into upper and lower triangular matrices. Sharing occurs along the blocks of an $N \times N$ matrix. As in the original SPLASH-2 version of this algorithm, data is allocated so that all of the blocks modified by a particular processor are allocated contiguously. Each 64 by 64 block, defined as a single region in the MEC version, occupies 32 KB, or 8 pages. Consequently, the LRC version uses 8 times as many messages to fetch a single block. False sharing exists within each block for both LRC and MEC but is limited by the page size in LRC. Thus, while MEC reduces the number of messages, it also transmits more bytes of data due to false sharing within blocks. In this case, however, the increase in the number of bytes due to false sharing in MEC, about 6%, is less significant than the decrease in the number of messages. As a result, the execution time using MEC in LU is 41% lower than when using LRC.

TSP solves the traveling salesman problem using a branch and bound algorithm. There are two data structures of interest in TSP, each of which is defined as a single shared region in the MEC version. The first (about 800 KB in size) is used to hold the partially evaluated tours, the priority queue consisting of pointers to tours in the pool, as well as associated data structures. The second (about 132 bytes in size) is used to keep track of the current shortest path. The size of the first region in MEC causes the number of messages to be reduced significantly compared to the LRC version. However, the number of bytes transmitted is about 40% higher in MEC because of the false sharing within regions. In TSP, this latter factor dominates, as execution time using LRC is 48% better than using MEC. Breaking up the larger data structure into smaller regions in the MEC version of TSP can avoid this performance degradation, but this requires more programming effort than the more obvious (albeit naive) way to use regions in this application.

FLOYD uses a dynamic-programming algorithm to solve the all-pairs shortest path problem on a directed graph. The distances and paths between each pair of nodes are computed and stored in separate two dimensional matrices. Each row of this matrix is defined as a region in MEC. Two regions, each equal to the size of one row, are also created and used to store a temporary copy of the current row of the distance and path matrix. The size of each row is less than one page. The number of messages is about the same for both MEC and LRC, though LRC suffers from some false sharing which causes it to transmit 27% more bytes than MEC. This results in a difference in execution times of about 9% in favor of the MEC version of FLOYD.

IS ranks an unsorted sequence of keys using bucket sort. In the MEC version only one region is created and is used for the shared array of buckets. We sort 2^{22} keys ranging from 0 to 2^9 for 10 iterations. With this problem size, a region occupies about half of a page, but this page contains no other data. As a result, the number of messages and bytes communicated is the same for both MEC and LRC (except for the first transfer of that data wherein LRC transfers the entire page unnecessarily), and execution times using either LRC or MEC are nearly identical.

4.3. Data Size and Alignment Sensitivity. Variations in problem size and data alignment have a significant impact on the relative behavior of LRC and MEC. We examined three of the applications (LU, SOR, and IS) in further detail by varying the problem sizes and thus changing the page alignments of the data structures in these applications (Figure 2). In each of the three applications examined, LRC and MEC execution times are nearly identical when data is page aligned and fits on one page, but small changes in the problem size cause data alignment to change in LRC and can result in significantly worse execution time for LRC. MEC behavior is insensitive to the alignment of data to pages. Consequently, in an integrated page-based and region-based system, the decision to use MEC in an application cannot be judged solely upon the performance of the application on a particular problem size. Rather, the performance of MEC versus LRC on a range of problem sizes must be taken into consideration, and the effort required to add MEC annotations for a particular data structure must be weighed against the effort required to avoid degradation in performance due to the misalignment of data and false sharing in LRC.

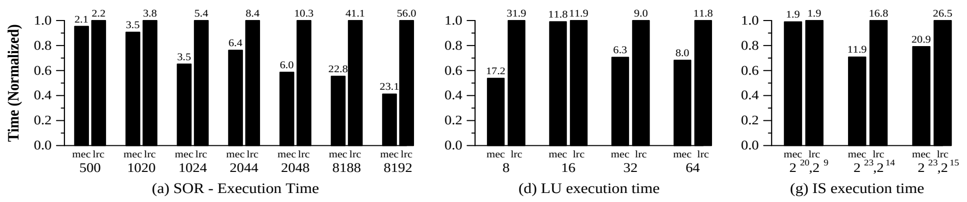

FIG. 2. *Varying input data characteristics in SOR, LU, and IS. In SOR,* $M = 500$ *and* N *is varied. In LU, the matrix size is fixed at 1024 and block size is varied. In IS, the number of keys and bucket sizes are both varied.*

5. Related Work. Adve *et al.* [1] conducted a performance comparison of lazy release consistency [9] and entry consistency [5] and conclude that there is no clear winner in terms of performance. They also point out that the performance of EC is hurt by extra synchronization and lock rebinding because all accesses to regions of shared memory may only occur after acquiring the corresponding lock. In contrast, the MEC model proposed in this paper uses no extra synchronization and requires no lock rebinding, yet retains the ability to manage data at region granularity. Further, in the three applications (SOR, TSP, and IS) that are common to both our study and the study by Adve *et al.*, our results show that variations in problem size have a significant impact on the difference between page-based and region-based data management. In SOR, for example, for the problem size used by Adve *et al.* (1000x1000 matrices), there is no significant difference between LRC and EC in their study and no significant difference in performance between LRC and MEC in our study. However, larger problem sizes in SOR favor MEC over LRC in our study.

The use of larger virtual memory pages for obtaining data aggregating effects in LRC has been previously considered by Amze *et al.* [3]. While larger virtual memory pages improves LRC performance for many applications, its effectiveness is limited by the increase in false sharing in others. On the other hand, MEC is capable of simultaneously eliminating false sharing along page boundaries and achieving much larger data aggregation, on the order of megabytes in some of the MEC applications we examine, in contrast with the 16 KB pages used in the study by Amze *et al.*.

Neves *et al.* [16] conduct a comparison of TreadMarks (LRC) and their system, called DiSOM, which implements EC. Their results show that EC can send significantly fewer messages and less data and thus obtain substantial reductions in execution times compared to LRC. The study by Neves *et al.* differs from our own in several respects, aside from the differences between MEC and EC already noted. The DiSOM system uses an object-oriented framework to avoid write trapping and exercise fine-grain control over communication, and their comparison between LRC in Treadmarks and EC in DiSOM is between two different DSM implementations that may differ in other respects as well. In contrast, our study compares MEC and LRC, both implemented within the Treadmarks framework.

Monnerat and Bianchini [15] introduce a protocol (ADSM) that dynamically and automatically associates locks with specific shared pages, in effect achieving part of the behaviour of entry consistency without the need for annotations, but still managing data at page granularity. Brecht and Sandhu [6] and Itzkovitz and Schuster [10] introduce two different techniques for managing data at region granularity without the need for annotations, using a novel pointer swizzling and page fault handling strategy in the former case, and conventional virtual memory fault handling in the latter. Both of these techniques improve the value of the MEC protocol described in this paper. Finally, Keleher [12] examined the performance impact of single versus multiple-writer protocols in a page-based release consistency context and found that multiple-writer protocols provide small improvements but considerably increase complexity. In MEC, using a multiple-writer protocol is intended primarily to allow a simpler programming style (since objects can be modified concurrently by different processors) than the single-writer EC protocol, and to allow applications to mix page and object based protocols without changing the synchronization model.

6. Conclusions. In this paper, we have presented a new model called Multiple-Writer Entry Consistency (MEC). The value of MEC is in its use of an RC-based synchronization structure while managing data at a granularity defined by the user. Thus, MEC permits the efficient integration of both page-based and region-based coherence protocols within the same RC framework. In this integrated environment, a program can be written initially for the RC model using page-based data management, and then MEC annotations added for those data structures which may obtain a performance advantage from region-based data management. MEC also presents a significant amount of flexibility in the way shared regions are defined by the user, by allowing a single region to be modified concurrently by different processors.

Our experimental evaluation of the performance of MEC and LRC shows that MEC improves execution times in five of the six applications that we study, by margins ranging from 5% to 46% for the default problem sizes, while leading to a degradation in performance of 48% in one case. These results, the first to compare page-based and region-based data management in an environment in which all other aspects of the system and the programs in the two models are nearly identical, suggests that integrating both page-based and region-based coherence protocols into one DSM framework is both practical and necessary.

7. Acknowledgements. We thank Ken Sevcik and the other members of the SIGPOW project at the University of Toronto for providing us with access to their cluster of workstations, which were used to perform our experiments. Also thanks to Paul Lu for providing us with the information needed to utilize the cluster.

REFERENCES

[1] S.V. Adve, A.L. Cox, S. Dwarkadas, R. Rajamony and W. Zwaenepoel, "A Comparison of Entry Consistency and Lazy Release Consistency Implementations", Proceedings of the 2nd International Symposium on High-Performance Computer Architecture, pp. 26-37, February, 1996.

[2] S.V. Adve and M.D. Hill, "Weak Ordering – A New Definition", Proceedings of the 17th Annual International Symposium on Computer Architecture, pp. 2-14, May 1990.

[3] C. Amza, A.L. Cox, K. Rajamani, and W. Zwaenepoel, "Tradeoffs between False Sharing and Aggregation in Software Distributed Shared Memory", Proceedings of the Sixth Conference on Principles and Practice of Parallel Programming, pp. 90-99, June 1997.

[4] E. Arjomandi, W. O'Farrell, I. Kalas, G. Koblents, F. Eigler and G. Gao, "ABC++: Concurrency by Inheritance in C++", IBM Systems Journal, Vol. 34, No. 1, pp. 120-137, 1995.

[5] B. Bershad, M. Zekauskas and W. Sawdon, "The Midway Distributed Shared Memory System", Proceedings of COMPCOM '93, pp. 528-537, February, 1993.

[6] T. Brecht, H. Sandhu "The Region Trap Library: Handling Traps on Application-Defined Regions of Memory", Proceedings of the 1999 USENIX Technical Conference, to appear.

[7] J. Carter, J. Bennett and W. Zwaenepoel, "Implementation and Performance of Munin", Proceedings of the 13th Symposium on Operating Systems Principles, pp. 152-164, October, 1991.

[8] M. Feeley and H. Levy, "Distributed Shared Memory with Versioned Objects", Proceedings of the Conference on Object-Oriented Programming Systems Languages, and Applications, October, 1992.

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

[10] A. Itzkovitz and A. Schuster, "MultiView and Millipage – Fine-Grain Sharing in Page-Based DSMs", Proceedings of the 3rd Symposium on Operating Systems Design and Implementation (OSDI '99), February, 1999.

[11] K. Johnson, F. Kaashoek and D. Wallach, "CRL: High-Performance All Software Distributed Shared Memory", Proceedings of the 15th Symposium on Operating Systems Principles, pp. 213-228, December, 1995.

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

[13] P. Keleher, A. Cox, S. Dwarkadas and W. Zwaenepoel, "TreadMarks: Distributed Shared Memory on Standard Workstations and Operating Systems", Proceedings of the Winter 1995 USENIX Conference, pp. 115-131, 1994.

[14] H. Lu, S. Dwarkadas, A.L. Cox, and W. Zwaenepoel, "Message Passing Versus Distributed Shared Memory on Networks of Workstations", Proceedings of Supercomputing '95, December, 1995.

[15] L.R. Monnerat and R. Bianchini, "ADSM: A Hybrid DSM Protocol that Efficiently Adapts to Sharing Patterns", Federal University of Rio de Janerio, COPPE Systems Engineering Computer Science Department, Technical Report ES-425/97, March, 1997..

[16] N. Neves, M. Castro, and P. Guedes, "A Checkpoint Protocol for an Entry Consistent Shared Memory System", Proceedings of the 13th Annual ACM Symposium on Principles of Distributed Computing August, 1994.

[17] H. Sandhu, B. Gamsa and S. Zhou, "The Shared Regions Approach to Software Cache Coherence on Multiprocessors", Proceedings of the Fourth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, pp. 229-238, May, 1993.

[18] J.P. Singh, W.-D. Weber and A. Gupta, "SPLASH: Stanford Parallel Applications for Shared-Memory", Computer Architecture News, Vol. 20, No. 1, pp. 5-44, March, 1992.