Dynamically Controlling False Sharing in Distributed Shared Memory

Vincent W. Freeh
Gregory R. Andrews

Dynamically Controlling False Sharing in Distributed Shared Memory

Vincent W. Freeh David K. Lowenthal Gregory R. Andrews

TR 96-10

Abstract

Distributed shared memory (DSM) alleviates the need to program message passing explicitly on a distributed-memory machine. In order to reduce memory latency, a DSM replicates copies of data. This paper examines several current approaches to controlling thrashing caused by false sharing in a DSM. Then it introduces a novel memory consistency protocol, *writer-owns*, which detects and eliminates false sharing at run time. In iterative computations, where the data is accessed similarly every iteration, the writer-owns protocol can have tremendous benefits because the overhead of eliminating false sharing is only incurred once. Performance results show that the writer-owns protocol is competitive with and often better than existing approaches.

June 7, 1996

Department of Computer Science
The University of Arizona
Tucson, AZ 85721

[1]First published in the "Proceedings of the Fifth Symposium on High-Performance Distributed Computing," IEEE, August, 1996.
[2]This work was supported by NSF grants CCR-9108412 and CDA-8822652.

1 Introduction

Distributed-memory multicomputers are used to achieve scalable high-performance computing. A convenient way to program such machines is to use a distributed shared memory (DSM), which provides logically shared memory and hence obviates the need to use explicit message passing. A DSM is responsible for transferring data between nodes and for maintaining consistent memory. However, because data travel on the interconnection network, there can be considerable delays in accessing non-local data. Therefore, modern DSM systems employ various techniques to reduce this latency, such as replicating local copies of data on many nodes. However, replicating data may result in false sharing, which in turn may cause thrashing.

False sharing occurs when two or more nodes access *distinct* data elements in the same unit of consistency (i.e., a DSM page) and at least one of them writes. Thrashing caused by false sharing can be controlled in several ways; the primary solutions are to reduce, avoid, or tolerate it. A DSM can reduce false sharing by using a small unit of consistency. Although thrashing may still occur, it is much less likely [BZS93, RLW94]. Second, a program can avoid false sharing by correctly placing data elements. This avoids thrashing but requires extensive static analysis [JE95]. Lastly, a DSM can tolerate false sharing by employing a weak memory consistency model in which there are multiple writers and hence the memory becomes inconsistent [GLL$^+$90]. Eventually, memory must be made consistent, however, and this can be costly.

This paper introduces a new memory consistency protocol, *writer-owns*, that at run time detects false sharing and then eliminates it. When false sharing is first detected, writer-owns tolerates it. At the next point at which the memory is made consistent, the system remaps the offending data to unused memory belonging to the writing node. This protocol has been implemented in the Filaments package [FLA94], and has very little overhead; programs using writer-owns are competitive with programs that avoid false sharing, and are better than ones that tolerate it.

The writer-owns protocol has several advantages over current methods. Because writer-owns dynamically detects false sharing, no static analysis is required. Moreover, it detects false sharing when it occurs; therefore, it discovers exactly every occurrence and no more. Because writer-owns eliminates false sharing, it incurs the overhead of detection and elimination only once; therefore, in iterative problems this overhead is amortized over all the iterations. Finally, writer-owns does not require any protocol-specific annotations or subroutine calls; consequently, writer-owns programs are simple and portable.

The remainder of this paper is organized as follows. Section 2 discusses false sharing in a DSM. Section 3 gives the details of the writer-owns protocol and compares it to a protocol that must avoid false sharing (write-invalidate) and one that can tolerate it (write-shared). It also describes an implementation of the writer-owns protocol. Section 4 compares the performance of write-invalidate, write-shared, and writer-owns on a number of application programs. Finally, Section 5 gives a few concluding remarks.

2 False Sharing in Distributed Shared Memory

The following example illustrates thrashing caused by false sharing in a DSM. We assume that the consistency unit is a page, data elements `x` and `y` are on the same page, there are two nodes A and B, and A and B execute the following:

```
Node A:              Node B:
  for ( ... )          for ( ... )
      x = ...              y = ...
```

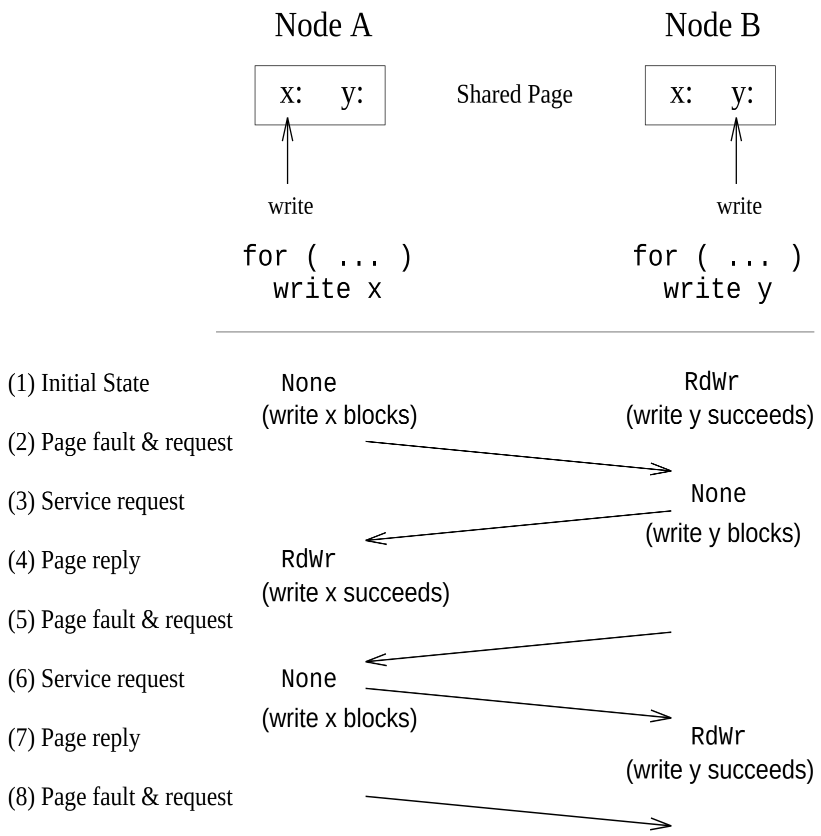

Figure 1: Thrashing example using write-invalidate.

In this example, there is false sharing because A and B update distinct locations on the same page. If there is only one copy of the page, there will be thrashing as the page bounces back and forth between the nodes.

Write-invalidate is a single-writer, multiple-reader protocol [LH89] that illustrates the issues involved. It provides *sequential consistency* in which memory updates must be made visible to all other nodes immediately [Lam79]. Write-invalidate (WI) is the canonical protocol and is the simplest to understand; consequently, it is the model programmers expect. In WI there can be multiple read-only copies or one writable copy of a page. The page remains consistent because updates occur only when a node exclusively owns the writable copy of the page.

Figure 1 shows how WI leads to thrashing in the above example. Initially, B has the single writable copy of the page. Node B is able to update `y`; however, when A attempts to update `x`, it must request the writable copy from B, which must invalidate its copy of the page before transferring it to A. Later, when B attempts to update `y`, it must request the single writable copy of the page from A. Since A and B are continuously updating `x` and `y`, the page will thrash.[1]

There are three methods for controlling thrashing caused by false sharing. The first uses a small unit of consistency in order to *reduce* the likelihood of false sharing. In the above example, choosing a smaller unit of consistency such that `x` and `y` are in distinct units will eliminate false sharing (and

[1]Thrashing also occurs even if A is only reading location `x`. Instead of transferring the page, node B sends a read-only copy to A and makes its local copy read-only. When B wants to update `y`, it sends an "invalidate" message to A before making its copy writable. The next time A reads `x`, it will re-request the page from B.

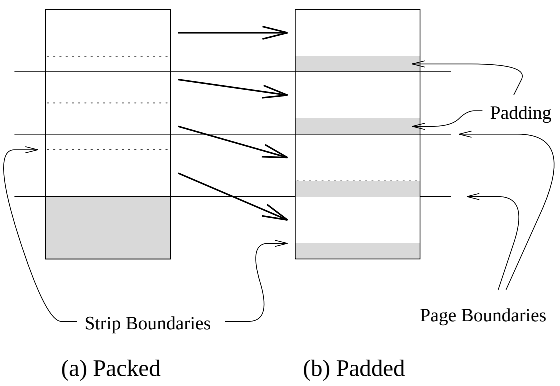

Figure 2: Padding a data object into four equal-sized strips.

hence eliminate thrashing). The size of the unit of consistency must be chosen carefully, however. If it is too large, there will be false sharing; if it is too small there will be extra overhead. Therefore, this method requires static analysis in order simultaneously to avoid false sharing and minimize overhead.

The second method places data statically in order to *avoid* false sharing. In particular, the program must place data so that each node has exclusive access to the pages it updates between synchronization points. We refer to this placing of data as *padding*. Consider a 2-dimensional array that will be updated in parallel by $\mathcal{N}$ nodes. The simplest way to pad the array is to partition it (and the tasks) into $\mathcal{N}$ strips, and assign one to each node (this is *block* distribution). Instead of laying the array out contiguously in memory, there is a break between the strips, so that each strip resides on a distinct set of pages. Figure 2 shows how data could be padded when a data object is partitioned into four strips. Each strip is three-quarters of a page; therefore, the object spans three pages when packed and four when padded. Avoiding false sharing in this manner can require extensive static analysis [JE95].

The third method employs a weak memory model in order to *tolerate* false sharing. One such example is the *release consistent* memory provided by the *write-shared* (WS) protocol [CBZ91]. It is a multiple-writer protocol, where each writer records its updates to a page locally; later, these changes are merged into a consist copy of the page. When the node first acquires a page that has multiple writers, the permission is set to read-only. The first update of the page generates a fault, at which time the page is "cloned" and made writable. All subsequent updates to the page succeed without delay. When the page is released, the clone is compared to the current page; all differences are stored in a "diff-list" and propagated to the other nodes sharing the page. Because all nodes have cloned the last consistent copy of the page, all diff-lists reflect changes to the same copy. After merging the diff-lists, each node will now have the same, consistent image of the page in memory.

Figure 3 shows false sharing in WS. The example begins with the same initial state as the WI example: node B exclusively owns the page. When node A requests the page, B returns a writable copy of the page. Node A immediately creates a clone because it is updating the page. Node B also creates a clone so that it can update the page. At this point both nodes have a writable copy of the page. If either node updates its copy of the page, the page is inconsistent. At the next consistency point, each node creates a diff-list for every cloned page and exchanges them with the other nodes.

The write-shared protocol has two significant drawbacks. First, false sharing is not eliminated; therefore, in iterative problems WS incurs the overhead of tolerating false sharing every iteration.

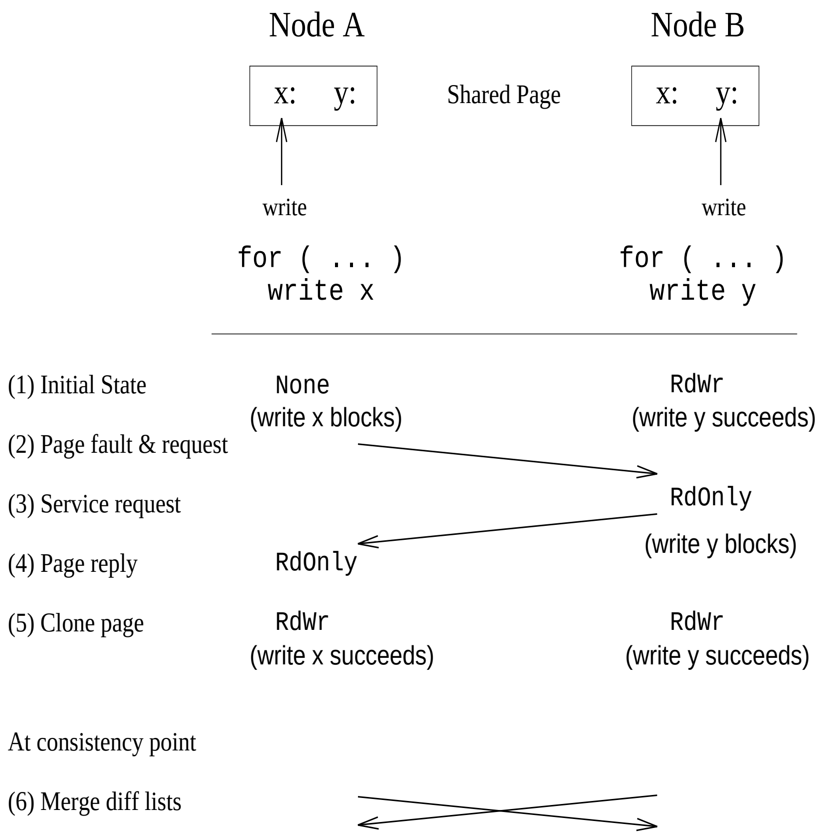

Figure 3: Thrashing example using write-shared.

Second, the number of messages required grows as the square of the number of nodes accessing a shared page because every node requires all the diff-lists.

3 The Writer-Owns Protocol

Writer-owns (WO) dynamically detects false sharing and then eliminates it by migrating data. In iterative computations, where the data is accessed similarly every iteration, eliminating false sharing can have tremendous benefits—not only is it eliminated, but the cost of eliminating it can be amortized over the subsequent iterations. Moreover, WO imposes very little overhead, so it is efficient even in non-iterative computations.

3.1 The General Writer-Owns Protocol

The writer-owns protocol has six steps:

1. *detect* false sharing,
2. *tolerate* false sharing until the next consistency point,
3. *create* the migration list of (falsely) shared pages,
4. *migrate* any data causing false sharing,

5. *disseminate* migration information to all nodes, and

6. *remap* user data structures to reflect the migration.

The first two steps occur for each falsely-shared page; the last four are performed by every node at the next memory consistency point. Each node determines which locations have been updated for each shared page in order to create a *migration list*, which is a list of the migrating data. Using the migration list, each node migrates the data it has updated to pages it owns. Then, migration information is collected and disseminated to all nodes so that the nodes can remap user data structures.

Data migrate to new locations in order to eliminate false sharing. An *atom* is the smallest unit of data that can migrate, the size and details of which depend on the implementation. However, an atom must able to be remapped and migrated.

Occurrences of false sharing are detected and tolerated while servicing DSM page requests. If a node receives a request for a page that it is updating, there is false sharing. At this point it changes permission to allow the update to proceed; however, the local updates must be recorded. Cloning is a simple and efficient method of recording local updates; therefore, each node that is updating a shared page clones the current (consistent) copy of the page before updating it. Later, the node determines the updates that were made by comparing the page to the clone.

Instances of false sharing that were discovered are eliminated at the next consistency point. After arriving at this point, the node determines what atoms were updated by comparing the cloned, consistent copy of the page to the current updated page. This operation is very similar to what is done in WS; however, because the updates are migrated and not merged, the *migration list* of WO is much smaller than the diff-list of WS. A migration list has an entry for each atom that has been updated, whereas a diff-list contains an entry for each location that was updated. A migration list cannot have more elements than a corresponding diff-list; in general, it will have many fewer.

Next, WO migrates data onto new pages in order to eliminate false sharing. All atoms that were updated are moved to pages that are owned by the writer of the data. In order to eliminate false sharing, each atom must only have a single writer. Otherwise, two nodes would try to migrate the same atom to different pages.

The fifth step is to collect migration lists from all nodes, merge them into a single list, and disseminate this list to all nodes. During this step WO discovers if an atom has more than one writer. An atom cannot be migrated to both writers. However, there are two reasonable actions that can be taken. First, the system could abort, alerting the user to the problem. The expectation is that the programmer would then re-write the program. Second, the system could attempt to merge the changes in a way similar to that taken in the write-shared system described above.

In the last step, each node remaps user data structures so that the program will access the data in the new locations. The specifics of this operation depend on how the atom is implemented.

In WO, false-sharing causes a sequence of events that is very similar to that shown for WS in Figure 3. The difference is the actions at the consistency point. In WS, each node merges the diff-lists from others nodes and applies the changes to its copy of the page. In WO, one node merges migration lists and disseminates it to the other nodes, which then migrate data and remap data structures.

3.2 An Implementation of Writer-Owns

The writer-owns protocol exists as a user-defined page-consistency protocol in the Filaments DSM. This section describes the implementation. It focuses on the critical design decisions.

The implementation of the atom has the most wide-ranging effects. An atom has at least one level of indirection (implemented by a pointer), so that remapping the atom only requires changing the value of the pointer. For example, a 2-dimensional array is implemented as an array of pointers to arrays of elements. To remap, we execute a statement similar to the following:

```
a[i] = new;
```

This remaps the `i`th atom (row) in the user data structure `a` so that it points to the location (`new`) into which the original contents have migrated.

Detecting and tolerating false sharing is straightforward in our implementation. Detection occurs when a page request is made. If the request is for a page that has been updated, there is false sharing. In order to tolerate it, WO makes a clone of the page. Then the page is made writable, which allows multiple writers. There are two choices for cloning the page: clone it as soon as sharing is detected (eager) or clone it later when the page is first updated (lazy). Our implementation uses lazy cloning. After sending a copy of the shared page to the remote node, the permissions on the local copy of the page are set to read-only. If the node tries to update a location on that page, it will generate a page fault; at this point the clone is made. If there is a subsequent update of the page, then lazy cloning requires an extra page fault over eager cloning. However, if all updates have already been applied, then eager cloning will compare identical pages when creating the migration list and lazy will not.

In order for WO to remap user data structures and create the migration list, the programmer must use the allocation routines provided by WO. These routines maintain a database of all user data objects; the database contains the location of the base pointer and information about the structure (including sizes) of each object. Using the database, WO can follow the base pointer to any atom pointer, and hence remap the atom. Similarly, WO uses the atom pointer and size information from the database to determine the extent of an atom.

The WO system creates the migration list by comparing the clone to the current copy of the page. For every difference between the two, WO inserts an entry into the migration list to identify the atom that was modified. Thus, the migration list contains an entry for every atom that was modified by this node since the last consistency point. Once an atom is inserted into the migration list, WO skips forward in memory to the beginning of the next atom; therefore, creating the migration list is most expensive when the current page and the clone are identical because then every location on the page must be checked.

After the migration list is constructed, all the atoms in it are migrated to unused, locally owned pages. Our current implementation moves each atom onto its own exclusive page (or set of pages). Although this can waste space, it reduces the chance of further false sharing. An alternative is to conserve space by packing atoms next to each other on the new pages.

An entry in the migration list has three fields:

`obj`	`atom`	`loc`

The first field, `obj`, identifies the user object and the second field, `atom`, identifies the atom within it. These fields are set when the entry is initially created. The third field, `loc`, identifies the new location to which the atom has migrated.

Figure 4 shows the effect of migrating and remapping a two-dimensional array, in which the rows are the atoms. The migration list consists of two elements; the first maps atom 3 of object 0 to page 11, the second atom 5 of object 0 to page 10. The effect of the migration is that two atoms (rows) migrate to new pages. Additionally, the top-level pointers are remapped to reflect this migration.

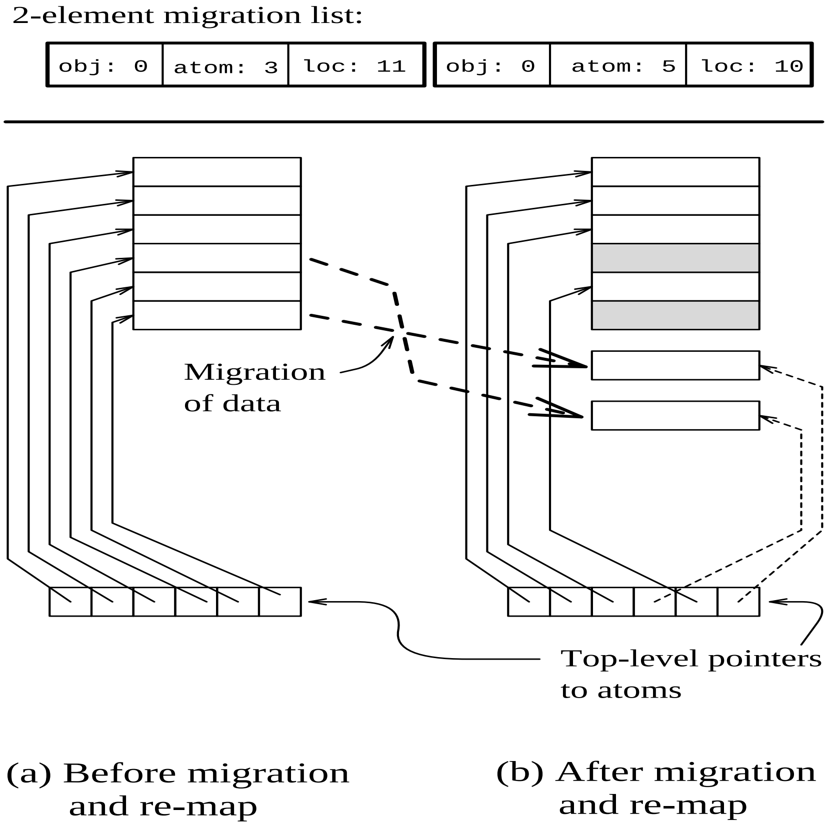

Figure 4: Migrating and remapping in a two-dimensional array.

In our implementation, the consistency points are *barriers*. A barrier is a point at which all nodes must arrive before any are allowed to proceed past. WO piggybacks both the local migration lists and the complete migration list on the barrier messages. Therefore, the migration information is transferred without any additional messages. The barrier messages become longer, but only when there is remapping. Furthermore, their length depends only on the number of atoms that are remapped, not on the size of the data or the number of nodes.

There are some limitations imposed by our design. First, because there must be a level of indirection to the remappable unit, certain data structures are not supported. For example, the individual elements of an array cannot be remapped. A simple technique to overcome this is to use pointers to the elements of the array; however, this can result in too much overhead. (Fortunately, the current solution works for a large number of applications.) Second, because an atom is the smallest data unit that can migrate, only one node can update an atom between consistency points. A programmer must ensure that a program obeys this exclusive-writer rule.

4 Performance

This section compares the writer-owns protocol, which dynamically eliminates false sharing, to two protocols: one that statically avoids false sharing and one that tolerates it. We consider four applications: Jacobi iteration, Tomcatv, matrix multiplication, and LU decomposition. For each we created three programs to evaluate the three page consistency protocols (PCP) discussed in this paper: write-invalidate, write-shared, and writer-owns. The only differences in the programs are the initialization of the protocol and the placement of the data structures in memory. The write-invalidate programs statically pad data structures; the write-shared and writer-owns programs pack data structures contiguously in memory. All protocols are implemented in the Filaments package.

Below, we briefly describe the four applications, present the results of runs on 1, 2, 4, and 8 nodes, and examine the parallel speedup. All tests were run on an isolated cluster of 8 Sun IPCs connected by a 10Mbs Ethernet. We used the `gcc` compiler, with full optimization. The execution times reported are the average of at least three test runs, as reported by `gettimeofday`. The tests were performed when the only other active processes were Unix daemons. For each application, we also created a purely sequential program in order to compute the speedup of the parallel programs. The speedup reported in this paper is the ratio of the sequential program time over the parallel program time.

In addition to overall times for all four applications, we present times for individual iterations in Jacobi iteration in order to show the overhead of migrating data in WO. To further evaluate the overhead of WO, we show the results of matrix multiplication on two different sized problems: one requires that WO remap data and the other does not.

4.1 Jacobi Iteration

Laplace's equation in two dimensions is the partial differential equation

$$\frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2} = 0.$$

Given constant boundary values for a region, its solution is the steady-state values of the interior points. These values can be approximated numerically by using a finite difference method such as Jacobi iteration, which uses the following equation to compute the $(k+1)$th approximation of U from the kth approximation:

$$u_{i,j}^{(k+1)} = \frac{1}{4}(u_{i-1,j}^{(k)} + u_{i+1,j}^{(k)} + u_{i,j-1}^{(k)} + u_{i,j+1}^{(k)}). \tag{1}$$

Thus the new value is the average of four neighboring points from the old grid. There are two $n \times n$ grids in the Jacobi iteration programs. On each iteration, the points in one grid are updated using the points in the other. For the next iteration, the roles of the grids are reversed. The algorithm iterates until the solution converges; i.e., until the maximum change that occurs at any point is less than some threshold.

We use a fine-grain Jacobi iteration program to compare the PCPs. A very lightweight thread (filament) is created for each of the n^2 points in the grid. Each filament updates the new grid point by averaging the values of the neighboring points in the old grid. These filaments are distributed among the nodes such that each owns one horizontal strip of the grid. After all grid points are updated, the program checks for convergence. If all the points have converged, the program terminates; otherwise it executes another iteration. Although there is very little work per filament per iteration, the program is still efficient, because the Filaments package efficiently implements fine-grain parallelism.

On every iteration, each node updates all the points in its strip of the new grid. This necessitates reading its strip of the old grid, plus reading a row from the strip of each neighbor. Thus, each strip of the new grid is written exclusively by the owner; the boundary rows of the old grid are read-shared between two neighboring nodes; and the interior of the old grid is read exclusively by the owner.

Nodes	1	2	4	8
Write-inv. Time	165.2	86.48	49.88	32.70
Write-inv. Speedup	0.99	1.89	3.28	5.01
Write-shared Time	165.9	88.45	52.51	43.73
Write-shared Speedup	0.99	1.85	3.12	3.75
Writer-owns Time	165.2	86.42	48.62	31.89
Writer-owns Speedup	0.99	1.89	3.37	5.14
Sequential Time	163.8			

Table 1: Jacobi iteration, 500×500 (times in seconds).

Iteration	1	2	3-100
Write-inv.	0.9175	0.4817	0.4958
Write-shared	0.9545	0.5865	0.5283
Writer-owns	0.9522	0.5300	0.4809

Table 2: Jacobi iteration on four nodes, 500×500 (seconds per iteration).

Since the old grid becomes the new grid on the next iteration, the boundary rows of the new grid are shared at the beginning of most iterations. In WI, the owner must invalidate remote copies of the boundary rows in order to perform the update. Furthermore, the nodes must receive the boundary rows of the old grid, which were updated in the previous iteration, from each neighbor. Therefore, on each iteration, each node in the WI program sends an invalidation message to its neighbors and requests a page from each of them. (Additionally, it receives an invalidation message and services a page request from each neighbor.)

In the WS programs, the boundary rows of the new grid have multiple writers; therefore, diff-lists must be exchanged between neighbors. Furthermore, the boundary rows of the old grid are shared by multiple readers only (no writers). Therefore, the nodes exchange empty diff-lists. On every iteration, each node must create, exchange, and merge the diff-lists.

The WO program performs identically to the WI program on the later iterations: for each neighbor, a node sends and receives an invalidate message, and it generates and services a page request. However, on the first two iterations, the WO program also must detect, tolerate, and eliminate false sharing. It takes two iterations to eliminate false sharing because false sharing only occurs while the grids are being updated and the second grid is not updated until the second iteration.

The WS program must exchange and merge diff-lists on each iteration; this accounts for the overhead relative to the other programs. The number of messages sent per iteration grows with the number of nodes. When there are 8 nodes, the number of messages saturates the network and results in a large increase in overhead relative to the other programs.

Table 1 shows the results of Jacobi iteration programs on a 500×500 matrix. The WO program is slightly *faster* than the WI program. Because of this surprising result, we evaluated the times of the individual iterations of the program, as reported in Table 2. There are 100 iterations in the test; the times for the first two iterations are shown along with the average of the last 98 iterations. The first iteration is always longest because the cache is cold. Because there are two grids in Jacobi

Nodes	1	2	4	8
Write-inv. Time	391.1	202.0	109.7	73.75
Write-inv. Speedup	0.99	1.94	3.57	5.30
Write-shared Time	392.0	204.8	114.7	81.50
Write-shared Speedup	1.00	1.91	3.41	4.80
Writer-owns Time	391.5	202.2	110.2	64.45
Writer-owns Speedup	1.00	1.93	3.55	6.07
Sequential Time	391.1			

Table 3: Tomcatv, 250×250 (times in seconds).

it takes two iteration for the programs to reach a steady state. During the first two iterations, the WO program tolerates and eliminates the instances of false sharing.

It is surprising that the WO program is faster than the WI because WO has more overhead to detect and eliminate false sharing. However, the programs employ different data placements, which results in slightly different memory references and different cache utilization. It turns out that the final placement arrived at by WO utilizes the cache more efficiently than the block distribution of WI. The WS program is always slower because it is cloning pages and merging diff-lists every iteration.

4.2 Tomcatv

Tomcatv is a mesh-generation algorithm obtained from the SPEC benchmark. The Fortran program in the benchmark was translated into a Filaments program. This application uses 7 shared $n \times n$ matrices, and there are two phases in the computation. The first phase computes residuals; the second uses the residuals to update a tridiagonal matrix. There is a barrier after the first phase, and a reduction, to obtain the maximum residual, after the second phase. The program iterates until the maximum residual is small enough. The sharing patterns in Tomcatv are very similar to Jacobi iteration. The big difference is there are seven shared matrices in Tomcatv, whereas Jacobi iteration only has two.

Table 3 contains the results for a 250×250 matrix. Again the WS program has the most overhead, which increases with the number of nodes. The WI and WO programs are very similar except for the 8-node test, which we cannot explain.

4.3 Matrix Multiplication

The programs compute $C = A \times B$, where A, B, and C are $n \times n$ matrices. Each node computes a horizontal contiguous strip of the rows of the C matrix. The nodes exclusively access portions of A and C; however, every node reads every element of B. Each node initializes part of the A matrix; one node initializes B, and the others acquire the pages of B through page faults.

Because matrix multiplication is not iterative, WO is not able to amortize the overhead of eliminating false sharing. However, WO is still competitive with WI, as shown in Table 4. Our implementation of WS requires that nodes exchange diff-lists for all shared pages, even those which are not updated. Because all 489 pages of the B matrix are shared by all nodes, there is tremendous overhead. A more elaborate WS implementation, such as the lazy release consistency protocol [KDCZ94], does not require nodes to exchange diff-lists unless the data is changing. The results

Nodes	1	2	4	8
Write-inv. Time	227.5	114.5	58.26	30.80
Write-inv. Speedup	0.97	1.93	3.80	7.19
Write-shared Time	224.2	118.0	61.65	38.44
Write-shared Speedup	0.99	1.88	3.59	5.76
Writer-owns Time	228.7	117.1	59.44	31.39
Writer-owns Speedup	0.97	1.89	3.73	7.06
Sequential Time	221.5			

Table 4: Matrix Multiplication, 500×500 (times in seconds).

Nodes	1	2	4	8
Write-inv. Time	245.7	125.5	63.71	33.31
Write-inv. Speedup	0.97	1.90	3.74	7.15
Write-shared Time	245.7	126.2	64.08	35.02
Write-shared Speedup	0.97	1.84	3.72	6.80
Writer-owns Time	245.6	125.5	63.67	33.26
Writer-owns Speedup	0.97	1.90	3.74	7.16
Sequential Time	238.1			

Table 5: Matrix Multiplication, 512×512 (times in seconds).

shown are for a hybrid program: The A and B matrices use the WI protocol, and only C uses WS. Consequently, only $\mathcal{N} - 1$ diff-lists must be exchanged. The hybrid program has message traffic that is similar to lazy release consistency.

Table 5 shows results for the special case where there is no false sharing. In a 512×512 matrix of type `double`, the rows require 4096 bytes, which is the size of a page in our system. Therefore, every row begins and ends on a page boundary, and the padded data layout is the same as the packed layout. The WS program does not exchange diff-lists and the WO program does not migrate data. Thus it is no surprise that the times of the three programs are nearly identical.

4.4 LU Decomposition

LU decomposition is used to solve the linear system $Ax = b$. It is an example of an application in which the load is not balanced. After a row is pivoted, it is never accessed again; on iteration i, only an $(n-i+1)$ by $(n-i+1)$ sub-matrix is accessed. On each iteration, the workload decreases by one row and every node must read the pivot row (row i), which is written by its owner. Communication is constant over all data placements. Therefore, strict block decomposition leads to severe tail-end load imbalance, because only one node is working during the last $n/\mathcal{N}$ iterations. To assure good load balance, many strips of contiguous rows of the matrix are distributed cyclicly among the nodes.

Table 6 shows the results using 160 strips of four rows each. Because each strip of four rows occupies exactly five pages, there is no false sharing. Consequently, the WS program does not exchange diff-lists and the WO program does not migrate data.

We further tested the programs using 320 strips of two rows each. In this case, there is false

Nodes	1	2	4	8
Write-inv. Time	281.8	151.8	85.53	53.94
Write-inv. Speedup	0.99	1.84	3.27	5.19
Write-shared Time	282.1	157.2	90.32	58.04
Write-shared Speedup	0.99	1.78	3.10	4.82
Writer-owns Time	288.5	154.0	86.11	56.13
Writer-owns Speedup	0.97	1.82	3.25	4.99
Sequential Time	280.0			

Table 6: LU decomposition, strip size of 4 rows, 640×640 (times in seconds).

Nodes	1	2	4	8
Write-inv. Time	281.8	151.9	85.49	54.02
Write-inv. Speedup	0.99	1.84	3.28	5.18
Writer-owns Time	288.5	154.2	86.18	56.01
Writer-owns Speedup	0.97	1.82	3.25	5.00
Sequential Time	280.0			

Table 7: LU decomposition, strip size of 2 rows, 640×640 (times in seconds).

sharing between two nodes on one out of every five pages. The WS program must exchange diff-lists for all of these pages on every iteration. There are 800 pages, so initially there are 160 shared pages. The WS program takes a tremendously long time and is not reported. The performance of the WI and WO programs using a strip size of two rows is shown in Table 7; it is very similar to the performance using a strip size of four. The benefit of slightly better load balancing is offset by the less efficient use of the cache.

5 Conclusions

This paper has shown that false sharing can be detected and eliminated at run time and has presented an efficient implementation of the writer-owns (WO) protocol. WO migrates data to new locations in order to eliminate false sharing. In our implementation, eliminating false sharing has minimal overhead. Moreover, because false sharing is eliminated, the overhead can be amortized over subsequent iterations.

We have shown that the writer-owns protocol has advantages over the write-invalidate and write-shared protocols. With write-invalidate, avoiding false sharing requires detecting and eliminating it statically. In contrast, writer-owns detects all false sharing as it occurs; hence, detection is simple and exact. The write-shared protocol tolerates false sharing, but differences between pages have to be resolved after *every* iteration of an iterative application. Consequently, WO requires significantly fewer consistency operations in iterative computations than WS. We have shown that programs using WO perform as well as those using write-invalidate and better than those using write-shared.

References

[BZS93] Brian N. Bershad, Matthew J. Zekauskas, and Wayne A. Sawdon. The Midway distributed shared memory system. In *COMPCON '93*, pages 528–537, 1993.

[CBZ91] John B. Carter, John K. Bennett, and Willy Zwaenepoel. Implementation and performance of Munin. In *Proceedings of 13th ACM Symposium On Operating Systems*, pages 152–164, October 1991.

[FLA94] Vincent W. Freeh, David K. Lowenthal, and Gregory R. Andrews. Distributed Filaments: Efficient fine-grain parallelism on a cluster of workstations. In *First Symposium on Operating Systems Design and Implementation*, pages 201–213, November 1994.

[GLL+90] Kourosh Gharachorloo, Daniel Lenoski, James Laudon, Phillip Gibbons, Anoop Gupta, and John Hennessy. Memory consistency and event ordering in scalable shared-memory multiprocessors. In *Proceedings of the 17th International Symposium on Computer Architecture*, pages 15–26, May 1990.

[JE95] Tor E. Jeremiassen and Susan J. Eggers. Reducing false sharing on shared memory multiprocessors through compile time data transformations. In *Fifth ACM SIGPLAN Symposium Principles and Practice of Parallel Programming*, pages 179–188, July 1995.

[KDCZ94] Pete Keleher, Sandhya Dwarkadas, Alan Cox, and Willy Zwaenepoel. TreadMarks: Distributed shared memory on standard workstations and operating systems. In *Proceedings of the 1994 Winter Usenix Conference*, pages 115–131, January 1994.

[Lam79] Leslie Lamport. How to make a multiprocessor computer that correctly executes multiprocess programs. *IEEE Transactions on Computers*, C-28(9):690–691, September 1979.

[LH89] Kai Li and Paul Hudak. Memory coherence in shared virtual memory systems. *ACM Transactions on Computer Systems*, 7(4):321–359, November 1989.

[RLW94] S. K. Reinhardt, J. R. Larus, and D. A. Wood. Tempest and Typhoon: User-Level Shared Memory. In *Proc. 21st Annual Symposium on Computer Architecture*, pages 325–336, May 1994.