An Effective Shared Memory Allocator for Reducing False Sharing in NUMA Multiprocessors

JongWoo Lee
Software Research & Development Center
Hyundai Electronics Industries, Korea
jwlee@ssrnet.snu.ac.kr

Yookun Cho
Department of Computer Engineering
Seoul National University, Korea
cho@ssrnet.snu.ac.kr

ABSTRACT Non-Uniform memory access (NUMA) time is an important issue in the design of large scale shared memory multiprocessors. One implication of NUMA architecture, however, is that locality of reference is crucial to the performance of the entire systems. So exploitation of locality of reference is necessarily supported by one or more system levels for efficient data sharing. Unfortunately, data sharing introduces a problem called false sharing which occurs when several independent objects which may have different access patterns are allocated to the same unit of movable memory (in our case, a page of virtual memory). In this paper we propose a simple and effective shared memory allocation mechanism for reducing the false sharing. Our design goal is to reduce the occurrences of false sharing misses by allocating independent objects that may have different access patterns to different pages. We use execution-driven simulation of real parallel applications to evaluate the effectiveness of our shared memory allocator. Our observation shows that by using our shared memory allocator, considerable amount of false sharing misses can be reduced and so the overhead of memory coherence protocol can also be reduced.

I. INTRODUCTION

NUMA architectures are becoming increasingly widespread because large scale shared memory multiprocessors can be built using them. Examples of current NUMA machines include BBN's Butterfly GP1000 and TC2000, and IBM's RP3[1]. In NUMA architectures, however, exploiting locality of reference is crucial to the entire system performance because access to a remote memory is much slower than access to a local memory1, and no attempt is made to hide this fact by hardware caching. A simple and fundamental mechanism for exploiting locality of reference is replicating or migrating frequently used pages from remote to local memory. But in case of page replication, the existence of multiple copies of the same page causes memory coherence problem. Whenever a processor modifies a local copy of a page, all processors that have the same page need to be informed of the change.

Memory coherence is accomplished through write-invalidation or write-update based protocol. In write-invalidation scheme, all copies of the page where the modified datum resides are invalidated. If a processor accesses an invalidated page, an invalidation fault occurs and then new copy can be replicated. In write-update scheme, update messages, which are composed of the updated location and its new value, are sent to all nodes that have local copy of the changed page, so that all processors that receive update message can modify their own corresponding local page to keep the page up-to-date. Traditionally, NUMA multiprocessors operating systems have used write-invalidation mechanism for its simplicity and ease of implementation. However, performance is not satisfactory mainly due to high page replication and invalidation costs and side effect called false sharing[3].

In write-invalidation protocol, all copies of the modified page are invalidated even though only one word in the page is changed. So when several independent objects which may have different access patterns are co-located in the same page, occurrences of unnecessary invalidation faults for keeping memory coherence are likely to increase. Since this unnecessary coherency overhead is not caused by the true sharing of data, it has been denoted false sharing[1, 4, 5]. Invalidation faults caused by false sharing do not affect the correct execution of parallel applications if they otherwise do not happen. One classic example of false sharing is when one part of a page is used exclusively by one pro-
cessor and another part is used exclusively by another processor, making the page appear shared even though all the data are used in a private fashion.

Eliminating this type of overhead induced by false sharing holds promise for performance improvements: execution time is reduced, and the decrease in coherency-related bus traffic enables the system to support more processors. The main goal of our work is to reduce the occurrences of useless invalidation faults caused by false sharing. To accomplish this we first define the false sharing reference to differentiate falsely-shared invalidation misses\(^2\) from truly-shared invalidation misses\(^3\). And we examine the number of false sharing misses happen during the execution of real parallel applications by tracing the memory access behaviors of them. And then we design and implement a simple and effective dynamic shared memory allocator to reduce false sharing misses. The reasons why we chose to optimize dynamic shared memory allocator for reducing false sharing are that this approach is transparent to the application programmers, and almost all false sharing misses happen in shared heap when multiple processes in an application communicate with each other using shared memory allocated by dynamic shared memory allocator. Support of transparency is very important for easy programming. In the absence of any system support, the programmer is forced to explicitly manage the false sharing problem, significantly increasing the development time for parallel applications.

The remainder of this paper is organized as follows: section II describes the definition of false sharing that can be used to differentiate between false sharing misses and true sharing misses; the overhead of useless invalidation misses caused by false sharing is presented in section III; section IV explains the related works; section V and section VI present our dynamic shared memory allocator and its performance results respectively. Finally, section VII presents our conclusion.

II. Definition of False Sharing

A precise and formal definition and quantification of the false sharing have proven to be elusive\(^5\). The definition of false sharing, however, must have the minimal property that it is able to draw a distinction between false sharing misses and true sharing misses. To satisfy this property, we define false sharing reference which means a memory reference that cause a false sharing miss. We assume that a memory reference is made in word-by-word fashion. And let \(r_{\text{page}, z}(a)\) and \(w_{\text{page}, z}(a)\) be read and write access to address \(a\) in page \(z\) by processor \(p\) respectively. Then a false sharing reference by a processor \(p\) is defined as follows. In the following definition, \(\text{reftime}(X)\) means the point of time when a reference \(X\) happens.

- \(r_{\text{page}, z}(a)\) or \(w_{\text{page}, z}(a)\) is a false sharing reference if the following two conditions are met simultaneously.

1. There exists a \(w_{\text{page}, z}(b)\) which precedes \(r_{\text{page}, z}(a)\) or \(w_{\text{page}, z}(a)\) (i.e., \(\text{reftime}(w_{\text{page}, z}(b)) < \text{reftime}(r_{\text{page}, z}(a)\text{ or } w_{\text{page}, z}(a))\)). Here, \(p \neq q\) and \(a \neq b\).

2. There does not exist any \(r_{\text{page}, z}(\ast)\) or \(w_{\text{page}, z}(\ast)\) during the interval between \(\text{reftime}(w_{\text{page}, z}(b))\) and \(\text{reftime}(r_{\text{page}, z}(a)\text{ or } w_{\text{page}, z}(a))\) where \(\ast\) means any address.

The above definition says that in a processor’s point of view the first invalidation miss caused by an access to a modified (by other processors) page is a false sharing miss if it is not caused by true sharing. We can also obtain the definition for true sharing reference by simply substituting \(a = b\) for \(a \neq b\) in the above definition. Figure 1 shows examples of false sharing and true sharing references according to our definition.

In this paper, we differentiate between false sharing misses and true sharing misses based on the above definition, and our performance related data we present in next sections is also measured using this definition.

III. Impact of False Sharing on Memory Performance

To examine the impact of false sharing on memory performance, we measure the ratio of false sharing misses to the total misses during one execution cycle of parallel applications. Typically, this ratio depends highly on the page size of systems that applications run. Since the false sharing is a product

\(^2\)simply, false sharing misses
\(^3\)simply, true sharing misses
application. In this graph, cold start misses and capacity misses are not included in the total number of misses for more exact comparison of false sharing misses between true sharing misses. As we can see in this figure, for most applications more than 90% of all invalidation misses are false sharing misses. It means that performance of a memory coherency protocol (in our case, write-invalidation scheme) may be significantly degraded by unnecessary overhead like false sharing misses. Other authors have also presented similar results in their papers [7, 8]. Particularly in [7], the authors have shown that the fraction of useless update messages are more than 90% of all messages in write-update protocol.

![Graph](image)

Figure 1: Examples of false sharing and true sharing references: according to our definition, $v_{page..1(1)}$ and $r_{page..1(0)}$ are false sharing references, and $r_{page..1(1)}$ is a true sharing reference.

![Graph](image)

Figure 2: Impact of page size on number of false sharing misses (using 32 processors).

Figure 2 also implies that false sharing may significantly degrade the performance of most systems that use 4K or larger page size.

Figure 3 shows the ratio of false sharing misses to the total misses during one execution cycle of each application. In this graph, cold start misses and capacity misses are not included in the total number of misses for more exact comparison of false sharing misses between true sharing misses. As we can see in this figure, for most applications more than 90% of all invalidation misses are false sharing misses. It means that performance of a memory coherency protocol (in our case, write-invalidation scheme) may be significantly degraded by unnecessary overhead like false sharing misses. Other authors have also presented similar results in their papers [7, 8]. Particularly in [7], the authors have shown that the fraction of useless update messages are more than 90% of all messages in write-update protocol.

![Graph](image)

Figure 3: The number of false sharing misses and true sharing misses that happen during executions of each parallel application (cont’d).

**IV. RELATED WORKS**

To alleviate the overhead caused by false sharing, several researchers have proposed some hand-
and how the data copies are organized. As a result, the coherence unit and the coherence protocol are both defined by the application. An application can tailor the unit of coherency to the data, thereby avoiding false sharing. It can also "schedule" writes to the global data space in order to alleviate contention. Finally, the application can change the data layout in order to reduce the number of replacement misses.

A major advantage of the above techniques is that performance gain can be definitely expected because such optimizations are performed directly by the application programmer in a customized hand-tuning fashion. There are, however, several drawbacks in the above techniques such as the followings: First, application programmers are heavily burdened with doing every optimizations by themselves. It is very difficult for programmers, even though they are very skilled in parallel programming, to know in advance access patterns of shared data. Second, application programmers must know the architecture related information, such as cache line size, for correct optimizations. And finally, the above techniques are not applicable to the parallel applications that multiple processes of one application communicate with each other using shared memory allocated by dynamic shared memory allocator (for example, share_malloc()). Since dynamic shared memory is allocated at run-time when each process needs it, the above hand-tuning methods cannot be used. Moreover, many parallel applications use a dynamic shared memory allocator for data sharing instead of light-weight thread model due to the application’s portability.

Our approach for reducing false sharing is to optimize dynamic shared memory allocation mechanism in a way that prohibits independent data objects from being co-located in the same shared page. And this approach is fully transparent to the application programmer, and can be applicable to many parallel applications that use shared memory allocated by dynamic shared memory allocator.

A. DRAWBACK OF EXISTING DYNAMIC SHARED MEMORY ALLOCATOR

The existing dynamic shared memory allocator serves allocation requests of applications based on sequential allocation scheme. In sequential allocation scheme, each allocated space that have different sizes may be co-located in a shared heap page. So the possibility that unrelated data objects reside on the same page becomes relatively high. Figure
4 shows an example of this sequential allocation scheme that all objects are mixed in a shared page.

Sequential allocation scheme, of course, have several obvious advantages such as minimal space overhead and ease of implementation. But in this scheme, many independent objects may reside on the same shared page, and so the false sharing misses are likely to increase.

Example Allocation Stream(Processor, Request Size): (p1, 256) (p2, 512) (p3, 200) (p2, 300) (p1, 256) (p2, 512) (p1, 512) (p2, 256)

Figure 4: An example of sequential allocation for a requests stream.

To overcome the drawback of sequential allocation scheme, Roger and Ellis have proposed "typed allocation" technique[1]. Typed allocation technique allows application programs to specify future reference patterns of each space when they request it. In this technique reference patterns that can be specified include read-mostly, write-mostly, and lock types. And then the data spaces that have same reference type are allocated to the same shared page. We think, however, that this technique is neither transparent to the programmers, nor it is easy for programmers to know in advance the future access patterns of each data space they request.

V. NUMAmalloc with Sized Allocation Feature

In this section we propose a simple and effective dynamic shared memory allocator to reduce false sharing misses. Our main design goals are as follows:

- **Minimal Co-location**:
  The cases that unrelated or independent objects reside on the same shared page must be minimized.

- **Application Transparency**:
  No application's intervention should be required for support of minimal co-location.

- **Space Efficiency**:
  The additional space required to support minimal co-location should be small.

In the following, we call our dynamic shared memory allocator NUMAmalloc.

**A. APPLICATION TRANSPARENCY**

In NUMAmalloc, we do not modify user interface of the allocator. So what the application programmers have to do to use our allocator is to modify the function name from the traditional to our own name.

**B. SUPPORT OF MINIMAL CO-LOCATION**

To avoid co-location of independent objects in the same page, we use "request size for allocation" as a hint for determining from which page the allocation requests are served. By allocating data spaces with the same size to the same page if possible, we try to approximate the minimal co-location of unrelated data objects. This approximation, of course, may lead to a misjudgement of placing independent data objects to the same page because the data objects with the same size may have different access patterns between each other, and the data objects with different size may have similar reference patterns. But we thought that the sizes of allocation requests could play an important role in predicting the near future reference patterns of data objects. Thus we assume that data spaces with different sizes would be likely to store different data structures, and so allocating data objects with different sizes to different pages will help us to reduce false sharing misses.

We could find out by simulation that this assumption holds for most parallel applications used in our experiments(section VI).

Several allocation mechanisms that use the allocation request sizes as a hint have been already suggested in [10, 11, 12, 13]. But, the major goals of these previous works are to improve the performance of dynamic storage allocator itself, or processor cache affinity.

Figure 5 shows an example of allocations for the same requests stream as Figure 4 when our sized
allocation technique is used. As we can see in this figure, since data objects that have different size are not mixed in a page, reduction of false sharing misses can be expected. In addition to this intuition, we can also look forward to the reduction of false sharing misses according to the following analytical modelling method[14].

![Diagram of allocation technique](image)

Figure 5: An example of allocations using sized allocation technique.

The unit of memory coherency is a page which is a set of words:

\[ p_j = \{ w_i | \text{word } i \text{ is part of page } j \} \]  
\[ p_j \cap p_k = \emptyset, \ j \neq k \]  
(1)  

And the processor set of a word is simply the set of processors that reference the word over the time interval of interest:

\[ W_i = \{ \text{processors referencing } w_i \} \]  
(2)  

The processor set of a page is the union of the processor sets of the words in the page:

\[ P_j = \bigcup_{w_i \in p_j} W_i \]  
\[ = \{ \text{processors referencing } p_j \} \]  
(3)  

Using these definitions, we can derive an equation for the degree of false sharing FS(i, j) that can be attributed to a particular \( w_i \) in \( p_j \):

\[ FS(i, j) = 1 - \frac{|W_i|}{|P_j|} \]  
(4)

The key idea of equation (4) is that the greater the difference between the word’s processor set size and the page’s processor set size is, the greater the degree of false sharing associated with that word becomes. And we can say that \( |P_j| \) of a page \( j \) containing many different size of data objects would be greater than \( |P_k| \) of a page \( k \) containing only the unique size of data structures.

\[ |P_j| > |P_k| \]  
(5)

So,

\[ FS(i, j) > FS(i, k) \Rightarrow FS(j) > FS(k) \]  
(6)  

Here, \( FS(j) = \sum_{w_i \in p_j} FS(i, j) \)  
(degree of false sharing associated with a given page \( j \))

C. SPACE EFFICIENCY OF NUMAMALLOC

In order to support minimal co-location, there is a trade-off between space efficiency and the rate of false sharing misses. By comparing the two allocation mechanisms (figure 4 and figure 5), we can observe that NUMAMalloc requires more pages than sequential allocation technique for the same requests stream. But, we can also observe by the following calculations that the additional number of pages required by our NUMAMalloc is at most less than or equal to the number of unique sizes that each application requests.

For a set of allocation requests \( S \),

\[ S = \{ s_1, s_2, \ldots, s_n \} \]  
(7)  

\( s_i = \text{request size of } i\text{th allocation.} \)  
\( (1 \leq i \leq n) \)  
\( n = \text{total \# of requests.} \)  

the total number of pages required to service \( S \) in sequential allocation mechanism can be calculated by using the following expression.

\[ \# \text{ of pages required} = \left\lfloor \frac{\sum_{i=1}^{n} s_i}{\text{pagesize}} \right\rfloor \]  
(8)

In NUMAMalloc, on the other hand, a set of allocation requests can be expressed as follows if requests’ order is ignored.

\[ S = \{ S_{size1}, S_{size2}, \ldots, S_{sizek} \} \]  
(9)  

\( S_{sizek} = \text{set of allocation requests with size } sizek. \)  
\( S_{size1} \cap S_{size2} \cap \cdots \cap S_{sizek} = \emptyset \)  
\( RS = \{ size1, size2, \ldots, sizek \} : \)  
set of unique requests sizes.

378
And we can calculate the total number of pages required in NUMAmalloc by the expression (10).

\[
\text{# of pages required} = \sum_{\text{sizek} \in RS} \left[ \frac{S_{\text{sizek}} \times \text{sizek}}{\text{pagesize}} \right]
\] (10)

The number of additional pages required in NUMAmalloc can be obtained by subtracting expression (8) from (10). The important fact we can observe from the two expressions is that the additional pages required in NUMAmalloc depend only on how many times the ceiling happens. It also implies that the space overhead (in number of pages) of NUMAmalloc is less than or equal to the number of unique request sizes (|RS| in definition (9)).

Space Overhead =

\[
\left( \sum_{\text{sizek} \in RS} \left[ \frac{S_{\text{sizek}} \times \text{sizek}}{\text{pagesize}} \right] \right) - \left( \frac{n}{\sum_{i=1}^{n} \frac{s_i}{\text{pagesize}}} \right) \leq |RS| \] (11)

The actual |RS| values of the parallel applications used in our experiments will be presented in section VI.

D. IMPLEMENTATION OF NUMAmalloc

Implementation is accomplished by adding page management part for every allocation sizes to the existing module. The free data chunks with the same size are linked with each other in a doubly linked free list so that allocations and deallocations can be performed easily. An allocated data chunk is removed from the corresponding free list, and a freed chunk is inserted to the head of its free list. To find the corresponding free list of a request size easily, we use head pointer indicating the location of the first chunk with the size. If a request size is the first one that has never been requested before, a new page is allocated from the shared heap, and the corresponding free list and its head pointer are newly created. The cases that there are no more free chunk in a free list are treated in the similar fashion without creation of a head pointer.

VI. EXPERIMENTATION

In this section, we describe the simulation environment, characteristics of parallel applications used in our simulation, and show the simulation results.

A. SIMULATION ENVIRONMENT

We use an execution-driven simulation technique to simulate shared memory NUMA multiprocessors with up to 32 nodes. The simulator we use consists of two parts: a front end, MINT (Mips INTERpreter)[13, 16], which simulates the execution of the processors, and a back end that simulates the memory system. The front end interprets object code of simulated application and calls the back end on every data reference. We exclude instruction fetches from memory references stream. At the present, MINT supports MIPS II instruction set only. The memory management policies and coherence protocols are actually simulated by the back end simulator. The back end simulator we implement includes modules to support the "page replication policy using adjustable DELAY counter"[6] and write-invalidation protocol, and some instrumentation modules for measuring false sharing misses. No modifications are made to the MINT. Our NUMAmalloc routines are not included in the back end simulator, but linked statically with parallel applications at compile time.

B. PARALLEL APPLICATIONS SUITE

Our applications consist of four parallel programs (BARNES, CHOLESKY, RADIX, WATER-SPATIAL). All programs are from SPLASH[17] and SPLASH2[18] suite. Table 1 shows total number of memory references and working set size of each application. BARNES simulates the evolution of a system of bodies under the influence of gravitational forces. CHOLESKY and RADIX perform parallel Cholesky factorization of a sparse positive definite matrix and parallel integer radix sort on Connection Machine CM-2 respectively. And WATER-SPATIAL simulates the evolution of a system of water molecules.

C. RESULTS

To measure the impact of dynamic shared memory allocator on false sharing, we compare the number of false sharing misses when the two allocation techniques (the existing one and our NUMAmalloc) are used.

Figure 6 shows how many false sharing misses are reduced when our NUMAmalloc is used by each parallel application. In this graph, y-axis represents
Table 1: Characteristics of Applications used in simulation (Using 32 processors).

<table>
<thead>
<tr>
<th>Application's Name</th>
<th>Memory Reference Length(10^6)</th>
<th>Working Set Size(MB)</th>
</tr>
</thead>
<tbody>
<tr>
<td>BARNES</td>
<td>58.716</td>
<td>1.20</td>
</tr>
<tr>
<td>CHOLESKY</td>
<td>39.341</td>
<td>2.88</td>
</tr>
<tr>
<td>RADIX</td>
<td>33.107</td>
<td>3.76</td>
</tr>
<tr>
<td>WATER-SPATIAL</td>
<td>334.054</td>
<td>1.36</td>
</tr>
</tbody>
</table>

the number of false sharing misses that happen during one execution cycle of each application. And our experiments are performed for the widely used page sizes (2048KB, 4096KB, 8192KB). As we can see in this figure, a considerable number of false sharing misses are reduced for most parallel applications. The reduction rate (percentage), which is presented at the top of each bar chart, ranges from 1.7% to 74.8%.

To evaluate the space efficiency of NUMAmalloc, we first trace allocation requests stream of each application. By tracing them, we can find out [RS] of each application and the additional number of pages required in NUMAmalloc. For each application, table 2 shows the allocation sizes and their request counts. The [RS] values of each application, which can be obtained by simply counting the number of rows in each table, are 6(BARNES), 10(CHOLESKY), 7(RADIX), and 6(WATER-SPATIAL) respectively. And the additional number of pages required can be calculated using requests stream data (table 2) and expression (8) and (10). Results of these calculations are presented in table 3. The important fact we can observe from table 3 is that the space overhead is relatively small compared to the total number of pages required (only 0.98% ~ 4.60%, or 16KB ~ 24KB in bytes).

VII. CONCLUSIONS

In this paper, we propose a simple and effective dynamic shared memory allocator to reduce false sharing misses in NUMA multiprocessors. The false sharing is one of the major overheads that may degrade the entire system performance. Actually we can observe by simulation that the ratio of false sharing misses to the total misses is more than 90% for most parallel applications used in our experi-

![Figure 6: Reduced number of false sharing misses when NUMAmalloc is used.](image-url)

380
Table 2: Allocation requests streams of each parallel application.

<table>
<thead>
<tr>
<th>BARNES</th>
<th>CHOLESKY</th>
</tr>
</thead>
<tbody>
<tr>
<td>Request Size(byte)</td>
<td># of Req.</td>
</tr>
<tr>
<td>544</td>
<td>1</td>
</tr>
<tr>
<td>8008</td>
<td>1</td>
</tr>
<tr>
<td>12296</td>
<td>1</td>
</tr>
<tr>
<td>163848</td>
<td>1</td>
</tr>
<tr>
<td>262152</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>RADIX</th>
<th>WATER-SPATIAL</th>
</tr>
</thead>
<tbody>
<tr>
<td>Request Size(byte)</td>
<td># of Req.</td>
</tr>
<tr>
<td>328</td>
<td>1</td>
</tr>
<tr>
<td>4104</td>
<td>1</td>
</tr>
<tr>
<td>131208</td>
<td>1</td>
</tr>
<tr>
<td>144</td>
<td>2</td>
</tr>
<tr>
<td>1048584</td>
<td>2</td>
</tr>
<tr>
<td>264</td>
<td>3</td>
</tr>
<tr>
<td>8200</td>
<td>32</td>
</tr>
</tbody>
</table>

Table 3: Space efficiency of NUMA malloc.

(A : When using existing allocation technique, B : When using NUMA malloc)

<table>
<thead>
<tr>
<th>Application's Name</th>
<th># of Pages Required</th>
<th>Space Overhead of NUMA malloc</th>
</tr>
</thead>
<tbody>
<tr>
<td>BARNES</td>
<td>136</td>
<td>2.94 KB</td>
</tr>
<tr>
<td></td>
<td>140</td>
<td>16</td>
</tr>
<tr>
<td>CHOLESKY</td>
<td>387</td>
<td>1.55 KB</td>
</tr>
<tr>
<td></td>
<td>393</td>
<td>24</td>
</tr>
<tr>
<td>RADIX</td>
<td>610</td>
<td>0.98 KB</td>
</tr>
<tr>
<td>WATER-SPATIAL</td>
<td>87</td>
<td>4.60 KB</td>
</tr>
<tr>
<td></td>
<td>91</td>
<td>16</td>
</tr>
</tbody>
</table>

REFERENCES


