CSTallocator: Call-Site Tracing Based Shared Memory Allocator for False Sharing Reduction in Page-Based DSM Systems

Jongwoo Lee[1], Sung-Dong Kim[2], Jae Won Lee[3], and Jangmin O[4]

[1] Dept. of Multimedia Science, Sookmyung Women's University, Seoul 140-742, Korea
bigrain@sookmyung.ac.kr
[2] Dept. of Computer Engineering, Hansung University, Seoul 136-792, Korea
sdkim@hansung.ac.kr
[3] School of Computer Science and Engineering, Sungshin Women's University, Seoul 136-742, Korea
jwlee@cs.sungshin.ac.kr
[4] NHN corp., 9th Fl. Venture Town Bldg. 25-1 Jungja-dong Bungdang-gu, Gyunggi-do, 463-844, Korea
jmoh@nhncorp.com

Abstract. False sharing is a result of co-location of unrelated data in the same unit of memory coherency, and is one source of unnecessary overhead being of no help to keep the memory coherency in multiprocessor systems. Moreover, the damage caused by false sharing becomes large in proportion to the granularity of memory coherency. To reduce false sharing in page-based DSM systems, it is necessary to allocate unrelated data objects that have different access patterns into the separate shared pages. In this paper we propose *call-site tracing-based shared memory allocator*, shortly *CSTallocator*. CSTallocator expects that the data objects requested from the different call-sites may have different access patterns in the future. So CSTallocator places each data object requested from the different call-sites into the separate shared pages, and consequently data objects that have the same call-site are likely to get together into the same shared pages. We use execution-driven simulation of real parallel applications to evaluate the effectiveness of our CSTallocator. Our observations show that our CSTallocator outperforms the existing dynamic shared memory allocator.

Keywords: False Sharing, Distributed Shared Memory, Dynamic Memory Allocation, Call Site Tracing.

1 Introduction

In distributed shared memory (DSM) systems, efficient data caching is critical to the entire system performance due to their non-uniform memory access time characteristics. Because the access to a remote memory is much slower than the access to a local memory, reducing the frequencies of the remote memory accesses with efficient caching can lead to decrease of the average cost of memory accesses, and subsequently improve the entire system performance [1]. A simple and widely

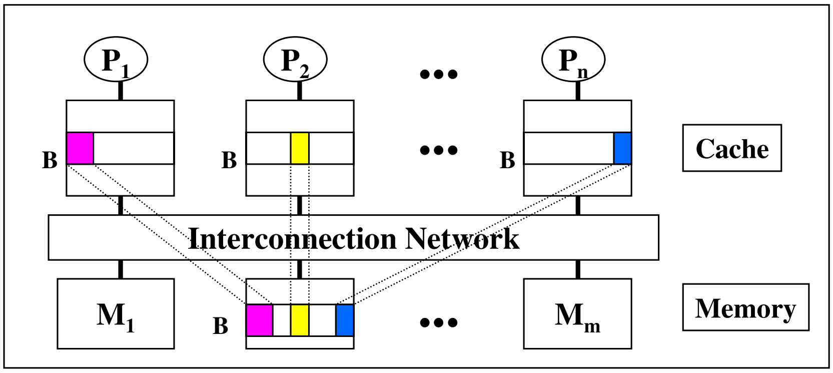


Fig. 1. Example of page replication in DSM systems

used mechanism for exploiting locality of reference is to replicate or migrate frequently used pages from remote to local memory [2]. But in case of page replication, the existence of multiple copies of the same page causes memory coherence problem (Fig. 1).

In DSM systems, ***false sharing*** happens when several independent objects, which may have different access patterns, share the memory coherency unit. Memory faults or misses caused by false sharing do not affect the correct executions of the parallel applications. As a result, we can say that false sharing is one major source of unnecessary overhead to keep the memory consistent [3, 4, 5, 6]. Especially, the problem becomes severe in PC-NOW DSM systems where the memory coherency unit is very large (generally, one virtual page). They say that the false sharing misses occupy 80% or so of the shared memory faults in page-based DSM systems [3, 4, 5, 6]. It means that the false sharing is the major obstacle for improving the memory performance in page-based DSM systems. In this paper, we present an efficient dynamic shared memory allocator for false sharing reduction in DSM system. 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 the false sharing misses happen in shared heap when multiple processes in a parallel application communicate with each other using shared memory allocated by dynamic shared memory allocator. The prediction of the future access patterns of each data object is necessary to reduce the false sharing misses caused by the data object. To accomplish this, we classify the data objects such that data objects requested at different locations in parallel program codes should not be allocated in the same shared page by tracing the call-site(object request location in parallel program codes). This is based on the idea that data objects requested at the different locations in program codes will show different access patterns in the future. Though the prediction technique of the access patterns we use is not always correct, we find out that our call-site tracing prediction technique could reduce the false sharing in comparison with other existing techniques. In order to measure the frequencies of page faults caused by false sharing(shortly false sharing misses), we use SPLASH and SPLASH II as a parallel application benchmark, and MINT as a multiprocessor architecture simulator.

In section 2, we review the related works. Section 3 explains the design and implementation of the call-site tracing-based shared memory allocator. We present the results of performance evaluation in section 4, and section 5 draws the conclusions.

2 Related Works

In this paper, we focus on the page-based DSM systems that keep the memory coherency in unit of a virtual memory page. The dynamic shared memory allocator for the page-based DSM systems has to decide where the requested data objects are placed. If the dynamic shared memory allocator knows the characteristics and access patterns of the requested data objects in advance, the allocator can easily place the data objects into the appropriate shared page with removing the causes of the false sharing. For example, the allocator can reduce the false sharing misses by placing the objects with much different access patterns to the different shared pages, or not placing non-related data objects into the same shared page. But, the dynamic shared memory allocator cannot know the characteristics and access patterns of the requested objects in advance. Therefore, the *typed allocation* is proposed in [7] where the clues provided by the programmers are used. In this typed allocation, the programmer must specify the memory access type through the allocation function arguments, such as Read-Only, Write-Mostly, and Lock types. Thus, the data objects with different types could be placed in the different shared pages. But, this scheme needs to additional overheads that user interfaces of the dynamic shared memory allocator have to be changed, and in turn the modification of the application source code is unavoidable. Moreover, it is not an easy job for the programmers to know in advance the access types of each shared data object. Our work assumes that there is no change in the API of the dynamic shared memory allocator.

Per-process allocation scheme assigns the different cache lines to the data objects requested by the different processes [3]. In this scheme, the data objects requested by the different processes are placed in the separate cache lines, so that it could reduce the possibility that data objects without relationships or with different access patterns are placed in the same cache line. This technique is effective where multiple processes request shared memory allocation evenly, but is likely to be ineffective where a dedicated process has the full responsibility of shared memory allocation [8]. In all the parallel applications used in our experiments, a dedicated process is also used for shared memory allocation, so it is inappropriate to compare this scheme with our approach.

Sized allocation scheme is proposed in [5, 6, 8], where the different-sized objects are prohibited from being placed in the same shared page. That is, by placing only the same-sized objects in the same shared page, this method tries to minimize the co-location of heterogeneous data objects. They say that, by using the object-size information for the prediction of the future access patterns, the transparency of the allocator API could be kept and the false sharing misses could be reduced simultaneously. Particularly according to [8], *allocation with separated tag* scheme and *minimizing the multi-page spanning* scheme could additionally reduce the false sharing misses. But this sized allocation is not enough to exactly predict the future access patterns of the shared data objects because the object size may not sufficiently represent the future access patterns. In our work, we will compare our call-site tracing technique with this sized allocation technique because these two methods have the similar assumptions that the interface of the dynamic shared memory allocator must not be modified and the shared memory allocation must be effective regardless of the existence of the allocation-dedicated process.

In [9], the hybrid allocation technique is proposed, which combines *per-processor allocation* and *minimizing the multi-page spanning* scheme. In this hybrid scheme, data objects requested by the different processors are placed in the different pages only when the object size is smaller than the page size. When the size of the data objects is bigger than the page size, on the other hand, they try to minimize the multi-page spanned allocations by prohibiting a shared object from being allocated in the page boundary. This technique could reduce the false sharing misses a little more by only combining the existing methods. But it is insufficient to accept this technique as a new prediction model of the future access patterns.

We find out by reviewing the previous works that the effective prediction of the future access patterns to be applied to the shared object allocations is an important factor to reduction of the false sharing misses. The shared objects which may have different access patterns must be placed in the different memory coherency unit. In this paper, we present ***call-site tracing based shared memory allocator***, shortly called ***CSTallocator***, where the future access patterns are predicted by the shared objects' request location(call-site) in the program codes. That is, the prediction is based on the instruction pointer from which the shared object allocation is requested. We hope that the objects with different call-sites may have the different access patterns in the future. By using the implicit information inherent in the program codes, our method not only keeps the API transparency, but also does not burden the programmers with the additional access type information. The call-site information of a shared object could be a useful clue for predicting the future access patterns because most parallel application programs call the allocation functions at different locations according to the object usage plans. Of course, the call-site tracing cost is more expensive than the cost of getting static information such as the allocation size passed via parameters or processor/process ID calling the function. Nevertheless, we can say that the call-site tracing overhead is not quite large because a call-site tracing procedure happens at a time only when the new call-site appears.

3 Design and Implementation of CSTallocator

With the information about objects' request locations in the program codes, we can infer the object's usage more accurately than with the object-size because multiple processes(or threads) cannot help to call the allocator at the different call-site according to the object's future usage. We expect that the future access patterns of the shared objects requested at different call-sites will be different even though the object sizes are the same. The only case that our expectation becomes wrong is when the usage of data objects requested at the same call-site changes abruptly and/or frequently. But it is difficult for the usage of the specific part of the program code to be dynamically changed, so we can use the object request call-sites as a clue for predicting the object's future access patterns.

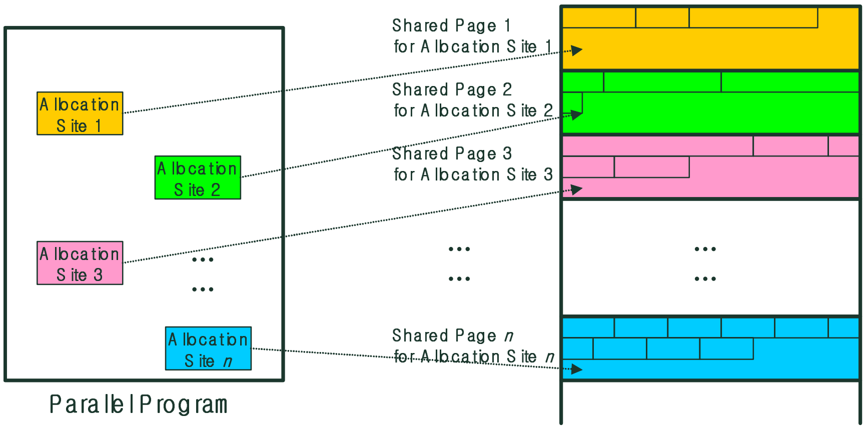

Allocation Site 1
Allocation Site 2
Allocation Site 3
Allocation Site *n*
Parallel Program
Shared Page 1 for Allocation Site 1
Shared Page 2 for Allocation Site 2
Shared Page 3 for Allocation Site 3
Shared Page *n* for Allocation Site *n*

Fig. 2. Shared objects allocation example according to the call-sites in CSTallocator

Fig. 2 shows an example of the call-sites of each shared object in a parallel application program. In this figure, the shared objects are placed in the separate pages according to their allocator call-sites. The key idea is to prevent the shared objects requested at the different call-sites from being placed in the same shared page, while the different sized objects are allowed to be in the same page if the objects are requested at the same call-site. In our experiments, we intentionally allow this situation for the exact one-to-one comparison with the sized allocation scheme. In addition, we exclude the mixture scheme of call-site tracing and sized allocation for the accurate comparison of the two methods. Though the mixture technique considering both the call-site and object size is expected to show better performance, we do not discuss about the mixture technique here, and leave it as a future work.

3.1 Call-Site Tracing Technique

To accomplish the call-site tracing based allocation, firstly we have to identify the call-site where the shared memory allocation function is called in the program codes. The identification procedure must be done dynamically and transparently in the shared memory allocation function without additional formal parameters. For this purpose, we embed a module called *call path back tracker*, into the shared memory allocator. By back tracking the activation records accumulated in the process's (or thread's) stack, we could identify the call path from *main()*, the starting point of the program, to the current call-site. A return address has to be stored in the activation record for returning from the function call, and we could get this return address by identifying the size of local variables and the parameters used in the function. The stack back-tracking repeats till the *main()* function. For example, if we get "share_malloc() ← B() ← A() ← main()" from the stack back-tracking at a certain call-site, the ID of this call-site is represented as "A→B". The *share_malloc()* and *main()* functions are excluded in the call site ID representation because they always

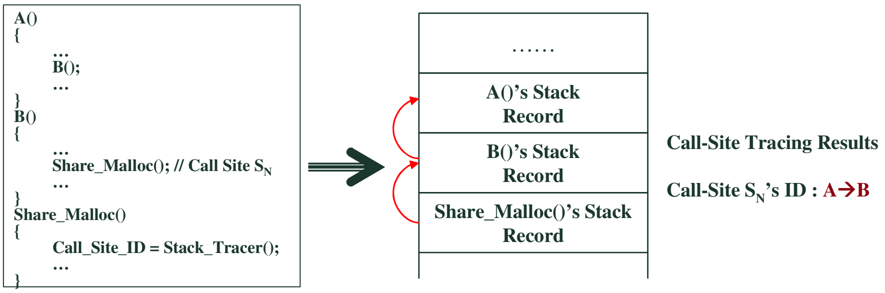

A()
{
...
B();
...
}
B()
{
...
Share_Malloc(); // Call Site S_N
...
}
Share_Malloc()
{
Call_Site_ID = Stack_Tracer();
...
}
......
A()'s Stack Record
B()'s Stack Record
Share_Malloc()'s Stack Record
Call-Site Tracing Results
Call-Site S_N's ID : A→B

Fig. 3. An example of identifying call-site ID by call-site tracing

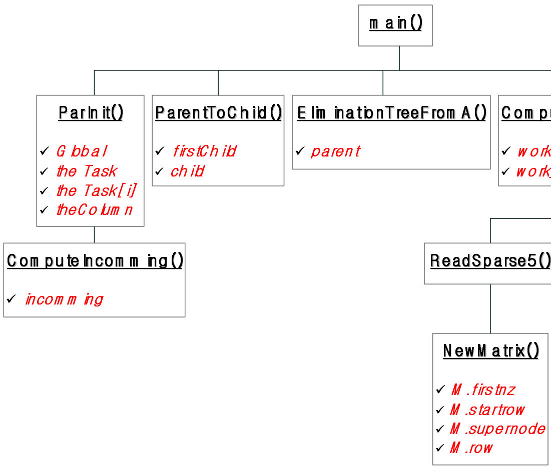
appear in every call-site ID. Fig. 3 shows an example of call-site tracing. In this figure, the call-site, S_N, is identified and registered with a call-site ID "A→B". And then, the shared objects with different call-site IDs are allocated in the different shared pages.

For the performance trade-offs of the stack back tracking, we have to consider the *back tracking depth of function call paths*. As a rule, a call-site ID can be defined after the back tracking to *main()* is completed. But in some parallel applications, we could identify all the call-sites without back tracking to *main()*. Therefore, we may decrease the overhead caused by the redundant stack back tracking if we could choose dynamically between the deep tracing and the shallow tracing. But the implementation of the dynamically depth-controlled back tracking is impossible because we cannot know the appropriate back tracking depth in advance to identify all the call-sites in a parallel application. So in our experiments, we will statically measure the effect of the back tracking depth adjustment on the performance. To do this, we define *length-N call chain*, which is the first *N* call paths from *share_malloc()* to *main()*. For example, length-1 call chain identifies only function *B()* which calls *share_malloc()*. In the same way, length-2 call chain includes function *B()*, which calls *share_malloc()*, and function *A()*, which calls *B()*, in the call-site ID. The longer the length of call chain, the deeper back tracking is to be done. In the prospect of the call chain length, we can expect that the possibility of false sharing would drop when using the longer length of call chain because it could identify the call-sites minutely.

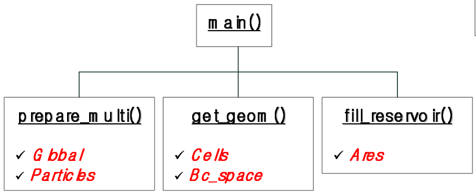
3.2 Examples of Call-Site Tracing in Parallel Application Programs

Fig. 4 shows the call-site tracing results for the parallel applications used in our experiments. This figure summarizes all the paths from the *main()* to the call-sites identified during the applications' run-time. As we can see in this figure, cholesky(fig. 4(a)) and volend(fig. 4(d)) have a relatively large number of call-sites, on the other hand, mp3d(fig. 4(b)) and barnes(fig. 4(c)) have much smaller number of call-sites than the others. For convenience, we exclude the functions that have no call-site in this figure.

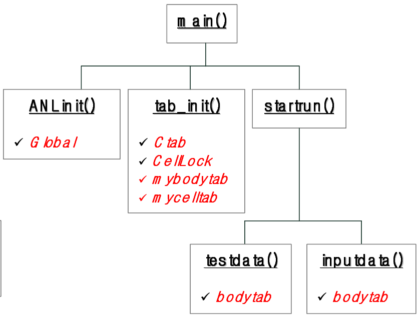
As we can see in this figure, most parallel application programs request shared memory at various locations in the program codes. In addition, we expect that the future access patterns to the shared objects allocated from the different call-sites are



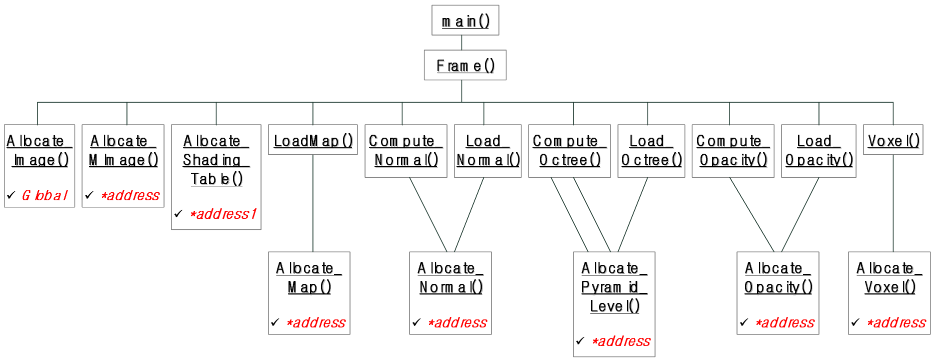
(a) Result of call-site tracing in Cholesky (maximum call chain length = 2).



(b) Result of call-site tracing in Mp3d (maximum call chain length = 1)



(c) Result of call-site tracing in Barnes (maximum call chain length = 1)



(d) Result of call-site tracing in Volrend (maximum call chain length = 2)

Fig. 4. Call-site tracing results in the four parallel application programs

likely to differ from each other because the different call-sites mean the different programmer's intention. We are sure that our approach can reduce the false sharing misses because it prevents the shared objects with different access patterns from being allocated in the same shared page. Of course, the cost of stack back tracking increases in proportion to the length of call chain, but we can find out that the lengths of call chains are not altogether long in most applications as shown in fig. 4. The maximum length of call chain in the four parallel application programs used in our experiment is only 2, so we could identify all the call-sites by using shallow tracing only.

4 Performance Evaluation

This section explains the experimental environments and shows the results of the false sharing misses measurement, comparing with the performance of the two allocators, CSTallocator and the sized allocator.

4.1 Experimental Environments

We use the execution-driven technique to simulate a DSM system consisting of 16 nodes. The simulator consists of the front-end and the back-end simulators. The front-end simulator interprets the execution codes of the parallel application program binaries and simulates the executions of the processors. We use MINT(Mips INTerpreter) [10, 11] as a front-end simulator. The back-end simulator simulates the policies of the memory management system using MINT's outputs. MINT interprets the execution codes and calls functions provided by the back-end simulator in every memory reference. The back-end simulator implements the memory management policies and the memory coherence protocols to be simulated.

We use cholesky, mp3d, barnes, and volrend as parallel application program suites. These parallel applications are randomly selected from the Stanford's SPLASH [12] and SPLASHII [13]. We compare the number of false sharing misses when using the two allocation schemes, CSTallocator and sized allocation scheme. We also measure the effects of the length of call chain, *N*, on the number of false sharing misses when using CSTallocator.

4.2 Experimental Results

Table 1 shows how many false sharing misses are reduced in each parallel application when using our CSTallocator. The number of buckets in the second column is the number of the unique allocation slots found during the repeated shared memory allocation function calls. It represents the number of object sizes when using the sized allocation scheme, and the number of call-site IDs when using our CSTallocator respectively. Both the shared memory allocators manage the allocated objects as a linked list using the separate pointers for each bucket. The shared pages with the same bucket pointers are assigned to data objects with the same call-site ID or object size. Thus, the more buckets are found, the more sophisticated classification has been done. In general, the false sharing misses will decrease when the number of buckets increases.

Table 1. Results of performance comparison of CSTallocator and sized allocation (page size = 4KB, N = length of call chain)

(a) Cholesky

Measure / Allocator	# of buckets	# of false sharing misses	Reduction rate(%)
Sized	10	44,717	
Call-Site-Tracing (N = 1)	15	40,921	8.5
Call-Site-Tracing (N = 2)	17	36,599	18.2

(b) Mp3d

Measure / Allocator	# of buckets	# of false sharing misses	Reduction rate(%)
Sized	8	6,147,589	
Call-Site-Tracing (N = 1)	5	5,754,143	6.4

(c) Barnes

Measure / Allocator	# of buckets	# of false sharing misses	Reduction rate(%)
Sized	27	5,805,705	
Call-Site-Tracing (N = 1)	7	5,104,413	12.1

(d) Volrend

Measure / Allocator	# of buckets	# of false sharing misses	Reduction rate(%)
Sized	11	953	
Call-Site-Tracing (N = 1)	8	931	2.3
Call-Site-Tracing (N = 2)	12	883	7.3

From the result of table 1, we can see that our CSTallocator is much more effective for the false sharing reduction than the existing sized allocation scheme for all the parallel application programs used in our experiment. This observation indicates that the object request location in program codes, that is call-site, can be a better clue than the object size for predicting the objects' future access patterns. For example, we find out that the number of false sharing misses rather decreased for mp3d and barnes in which the sized allocation scheme uses more buckets. To our expectations, the false sharing misses reduction ratios of cholesky and volrend becomes larger in proportion to the length of call chain. This means that the future access patterns of the objects could be predicted more accurately with the fine-grained call-site identification. Moreover, the fact that the false sharing misses decrease even though the number of buckets decreases supports that our CSTallocator is also more effective in space efficiency than the sized allocation scheme.

4.3 Analysis of Space Efficiency

For the strict performance evaluation, we need to analyze space overheads caused by CSTallocator and the sized allocation scheme. The space overhead is the amount of memory used additionally by the proposed methods. For more accurate space efficiency analysis, we need to analyze the time efficiency in conjunction with space efficiency. But in our experiments, it is impossible to measure the actual execution time of the allocation functions because we use the simulation method instead of real executions. So we do not discuss about the time efficiency here, and leave it as a future work.

At first, we analyze the general shared memory allocator, which does not use the buckets such as object size or call-site ID. In the general shared memory allocator, the objects can be mixed up into the same shared page according to the sequence of requests. So in the general allocator, the allocation requests stream, *S*, is represented as:

$$S = \{s_1, s_2, ..., s_n\} \tag{1}$$

where s_i = requested size of i-th allocation ($1 \le i \le n$), n = total # of requests.

The number of pages needed to accept the above allocation requests stream is as follows:

$$\#\text{ of pages required} = \left\lceil \frac{\sum_{i=1}^{n} s_i}{\text{page size}} \right\rceil \tag{2}$$

On the other hand, when using CSTallocator or sized allocation scheme, the allocation request stream can be represented as follows without considering the order of requests:

$$\begin{aligned}
&S = \{S_{bucket_1}, S_{bucket_2}, ..., S_{bucket_k}\},\\
&S_{bucket_k} = \text{set of allocations with bucket ID } bucket_k,\\
&S_{bucket_1} \cap S_{bucket_2} \cap ... \cap S_{bucket_k} = \varnothing,\\
&BS = \{bucket_1, bucket_2, ..., bucket_k\}: \text{set of unique bucket IDs.}
\end{aligned} \tag{3}$$

And the number of pages needed to accept the above stream is as follows:

$$\#\text{ of pages required} = \sum_{bucket_k \in BS} \left\lceil \frac{|S_{bucket_k}| \times AvgSize_{bucket_k}}{\text{page size}} \right\rceil, \tag{4}$$

where $AvgSize_{bucket_k}$ is average size of each allocation request heading for $bucket_k$.

In comparison of the equation (2) with (4), the difference lies in the number of ceiling function. In equation (2), the ceiling function is applied at once, while it is applied as many as the size of the set BS (|BS|) in equation (4). This means that the maximum additional number of pages is limited to the number of the unique allocation sizes in sized allocation scheme and the number of call-site IDs in CSTallocator respectively. Thus, the following is valid:

$$\text{Space Overhead} = \left(\sum_{bucket_k \in BS} \left\lceil \frac{|S_{bucket_k}| \times AvgSize_{bucket_k}}{\text{page size}} \right\rceil \right) - \left\lceil \frac{\sum_{i=1}^{n} s_i}{\text{page size}} \right\rceil \leq |BS| \qquad (5)$$

The obvious fact we can obtain from the above equations is that the shared page overhead is no more than the number of buckets regardless of which bucket classification methods are used. Table 2 shows the comparison results about the space efficiency measured by equation (5). As we can see in this table, the space efficiency of CSTallocator is better than that of the sized allocation scheme for mp3d and barnes, but is a little worse for cholesky and volrend. It means that the space efficiency gap between the two schemes is not quite large. Moreover, nowadays the space overhead such like the above can be surely tolerable if the memory specification of the current computer systems is taken into account.

Table 2. Space efficiencies of the two schemes (Page size = 4KB. In CSTallocator, maximum length of the call chain is used)

Parallel application programs (total # of pages needed in general allocation method)	# of additional pages (space overhead (%))	
	Sized allocation	CSTallocator
Cholesky (738)	10 (1.36)	17 (2.30)
Mp3d (553)	8 (1.45)	5 (0.90)
Barnes (308)	27(19.85)	7 (5.15)
Volrend (441)	11 (2.49)	12 (2.72)

5 Conclusions and Future Works

This paper presents an efficient shared memory allocation method for parallel applications which communicate via dynamically allocated shared memory in DSM systems. Without modifying the user interface of the shared memory allocator, the proposed call-site tracing-based allocator, called CSTallocator, can reduce the false sharing misses more effectively than the existing sized allocation scheme. Our CSTallocator prevents the shared objects with different call paths being allocated in the same shared page by tracing the object request location in the application program codes. We use the call-site as a clue for predicting the programmer's intention, and find out by simulation that the call-site help to predict the future access patterns of the shared objects more accurately than the existing sized allocation scheme. The CSTallocator additionally spends pages only as many as the number of unique call-sites in the applications. That is, our method could reduce more false sharing misses with a moderate space overhead than the sized allocation scheme. We are sure that our CSTallocator can contribute to both reduction of the false sharing misses and reduction of the cost on keeping the memory coherency in DSM systems.

In the future, we will verify how many false sharing misses can be reduced when using the mixture scheme of the sized allocator and our CSTallocator. And to measure the time efficiency as well as space efficiency, we will try to use the real DSM systems as a test bed instead of simulation environments.

References

[1] Andrew S. Tanenbaum. *Distributed Operating Systems*, chapter 6, pages 333-345. PRENTICE HALL, 1995.

[2] Jongwoo Lee, Yookun Cho. Page Replication Mechanism using Adjustable DELAY Counter in NUMA Multiprocessors. *Journal of the Korean Institute of Telematics and Electronics B*, 33B(6), pp.23-33, June 1996.

[3] Josep Torrellas, Monica S. Lam, and John L. Hennessy. Shared Data Placement Optimizations to Reduce Multiprocessor Cache Miss Rates. In *Proceedings of the 1990 International Conference on Parallel Processing*, volume II(Software), pages 266-270, August 1990.

[4] Susan J. Eggers and Tor E. Jeremiassen. Eliminating False Sharing. In *Proceedings of the 1991 International Conference on Parallel Processing*, volume I(Architecture), pages 377-381, August 1991.

[5] Jongwoo Lee, Yookun Cho. Shared Memory Allocation Mechanism for Reducing False Sharing in Non-Uniform Memory Access Multiprocessors. *Journal of Korean Information Science Society(A): Computer Systems and Theory*, 23(5), pp.487-497, May 1996.

[6] JongWoo Lee and Yookun Cho. An Effective Shared Memory Allocator for Reducing False Sharing in NUMA Multiprocessors. In *Proceedings of 1996 IEEE 2nd International Conference on Algorithms & Architectures for Parallel Processing(ICA^{3}PP '96)*, pages 373-382, June 1996.

[7] Roger L. Adema and Carla Schlatter Ellis. Memory Allocation Constructs to Complement NUMA Memory Management. In *Proceedings of the 3rd IEEE Symposium on Parallel and Distributed Processing*, December 1991.

[8] Jongwoo Lee, Moonhee Kim, Janghee Han, Daeku Ji, Jongwan Yoon, Jangseon Kim. Effects of Dynamic Shared Memory Allocation Techniques on False Sharing in DSM Systems. *Journal of Korean Information Science Society(A): Computer Systems and Theory*, 24(12), pp.1257-1269, December 1997.

[9] Boohyung Han, Seongje Cho, Yookun Cho. Techniques for Eliminating False Sharing and Reducing Communication Traffic in Distributed Shared Memory Systems. *Journal of Korean Information Science Society(A)*, 25(10), pp.1100-1108, October 1998.

[10] J. E. Veenstra. MINT Tutorial and User Manual. Technical Report TR452, Computer Science Department, University of Rochester, July 1993.

[11] J. E. Veenstra and R. J. Fowler. MINT: A Front End for Efficient Simulation of Shared-Memory Multiprocessors. In Proceedings of the Second International Workshop on Modeling, Analysis and Simulation of Computer and Telecommunication Systems(MASCOTS '94), pages 201-207, January-February 1994.

[12] J. P. Singh, W. Weber, and A. Gupta. SPLASH: Stanford Parallel Applications for Shared-Memory. *ACM SIGARCH Computer Architecture News*, 20(1):5-44, March 1992.

[13] Steven Cameron Woo, Moriyoshi Ohara, Evan Torrie, Jaswinder Pal Singh, and Anoop Gupta. The SPLASH2 Programs: Characterization and Methodological Considerations. In *Proceedings of the 22nd Annual International Symposium on Computer Architecture*, pages 24-36, June 1995.