SCSTallocator: Sized and Call-Site Tracing-based Shared Memory Allocator for False Sharing Reduction in Page-based DSM Systems⋆

Jongwoo Lee[1], Youngho Park[1], and Yongik Yoon[1]

[1] Dept. of Multimedia Science, Sookmyung Women′s University, Seoul 140-742, Korea
{bigrain, yhpark, yiyoon}@sookmyung.ac.kr

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 *sized and call-site tracing-based shared memory allocator*, shortly *SCSTallocator*. SCSTallocator expects that the data objects requested from the different call-sites may have different access patterns in the future. So SCSTallocator 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. At the same time SCSTallocator places each data object that has different size into different shared pages to prohibit the different-sized objects from being allocated to the same shared page. We use execution-driven simulation of real parallel applications to evaluate the effectiveness of our SCSTallocator. Our observations show that our SCSTallocator outperforms the existing dynamic shared memory allocators. By combining the two existing allocation technique, we can reduce a considerable amount of false sharing misses.
Keywords. False Sharing, Distributed Shared Memory, Dynamic Memory Allocation, Sized Allocation, Call Site Tracing

1 Introduction

In distributed shared memory (DSM) systems, efficient data caching is crucial to the entire system performance due to the 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 used mechanism for exploiting locality of reference is to replicate or migrate frequently used pages from

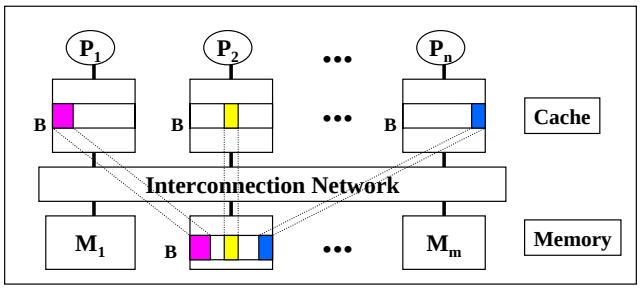

Fig. 1. Example of page replication in DSM systems

⋆ This Research was supported by the Sookmyung Women′s University Research Grants 2006.

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]. Especially, the problem becomes severe in PC-NOW DSM systems where the memory coherency unit is very large (generally, one virtual page). They find out that the false sharing misses occupy 80% or so of the shared memory faults in page-based DSM systems [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. 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.

To reduce the false sharing misses in a shared memory allocator, a prediction of the future access patterns of each data object is necessary. To predict the future access pattern of each data object at the shared memory allocators run-time, several techniques are suggested [5, 6, 8, 9]. Among them, [5, 6, 8] and [9] motivated us to expect that, by combining the two techniques, more performance improvements can be possible. In [5, 6, 8] and [9], the sized allocation technique and call-site tracing based allocator(shortly CSTallocator) are proposed respectively. As we can see from the name of the techniques, sized allocator uses the request size as a clue to predict the future access pattern of each data object. The sized allocator expects that data objects of different sizes are likely to show different access patterns in the future. In CSTallocator, on the other hand, by tracing the call-site(object request location in parallel program codes), data objects requested at different locations should not be allocated in the same shared page. This is based on the idea that data objects requested at the different call-sites will show different access patterns in the future.

In section 2, we review the related works. Section 3 explains the design and implementation of the combined version (SCSTallocator). 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 to blockade the false sharing misses. 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 should be no changes 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, 14, 15, 16]. 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 allocator (Fig. 2) 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. 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.

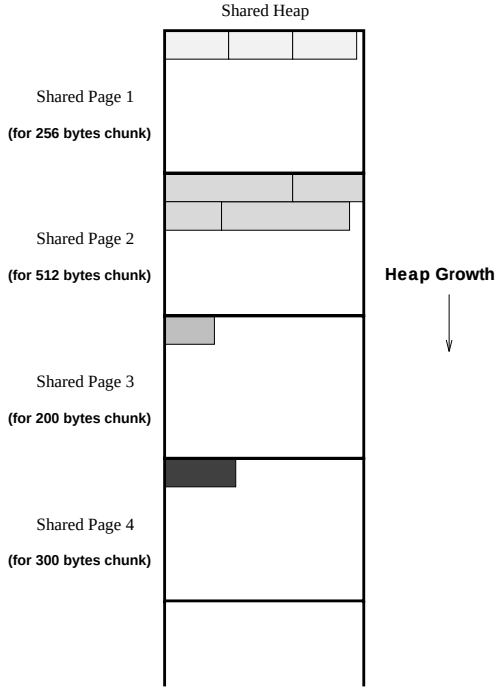

Fig. 2. Shared objects allocation example according to the object size in sized-allocation scheme

In [9], call-site tracing based shared memory allocator, shortly called *CSTallocator*, is presented(Fig. 3). In this technique, the future access patterns are predicted by the shared objects' request location(call-site) in the program codes. That is, the prediction is performed based on the instruction pointer from where the shared object allocation is requested. They 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, CSTallocator tries not only to keep the API transparency, but also to unburden the programmers' efforts. There, authors claim that

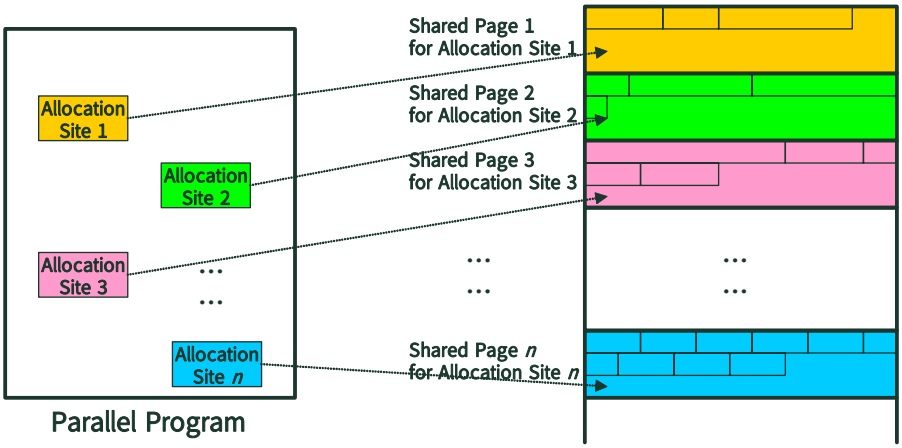

Fig. 3. Shared objects allocation example according to the call-sites in CSTallocator

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, they admit that the call-site tracing cost is more expensive than the cost of getting static information such as the request size passed via parameters or processor/process ID calling the function.

We find out by reviewing the previous works that the effective prediction of the future access patterns is a very important factor to reduction of the false sharing misses. And we can also find out that the sufficient false sharing reduction can't be obtained using just one prediction technique. These observations motivated our work to combine the existing prediction techniques(sized allocation scheme and CSTallocator) to simultaneously use them at the dynamic shared memory allocator run-time.

3 *SCSTallocator*: Combining the *Sized* and *CSTallocator*

The object size and the call-site have a different point of view. Object sizes are syntactic information describing the characteristics of each object. We can say, on the other hand, call-sites are semantic information showing the future usage pattern of each object. We expect that, by combining the merits of the two heterogeneous prediction clues, larger amount of false sharing misses can be reduced. To evaluate the effectiveness of the combined version, we implement an allocator using the two prediction policies, sized allocation and CSTallocator, simultaneously. Fig. 4 shows an example of allocations when using this combined version. We are sure that the two prediction clues are the only information we can obtain from the allocators without modifying their API.

As we can see in this example, SCSTallocator allocates each object into the different pages if even one of the object size and call-site is different from each other. After all, a shared page contains only the same-sized objects requested at the same location in program codes hoping for that co-location of the different objects that may have different access patterns can be minimized. Because both the object size and the call-site are used to classify each object, the same-sized objects requested at the different call-sites are allocated into the different shared pages, and similarly the same call-site's objects of different size are also allocated into the different shared pages. The case SCSTalocator cannot cover is the following situation: future access patterns of the objects allocated into the same shared pages are unexpectedly different from each other. However, any one who has some experiences of parallel programming can

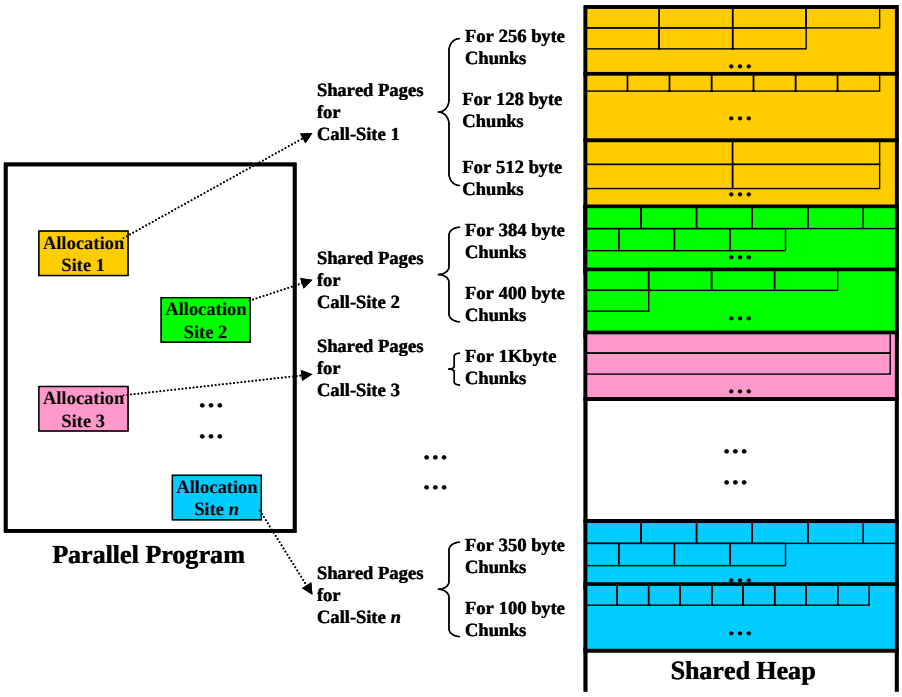

Fig. 4. An example of allocations when using the *SCSTallocator* (Sized allocation + CSTallocator)

easily accept that the above worst case situation may rarely happen because it is impossible to request several objects of different usage plans at the same call-site even though they have the same size.

Fig. 5 shows the implementation snapshot of bucket management in the SCSTallocator. Here, a bucket means a unique allocation control slot to gather only the

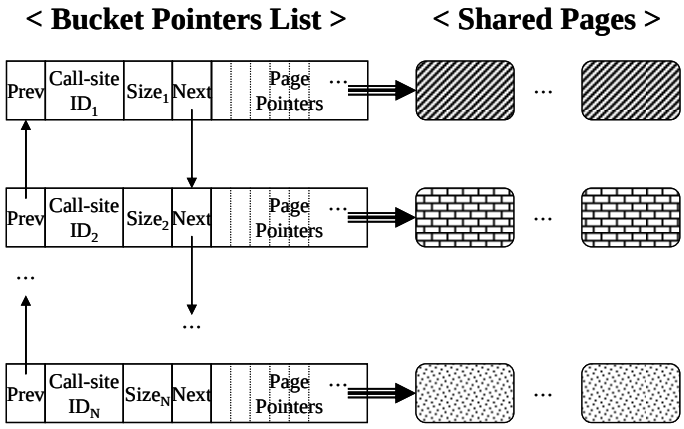

Fig. 5. Bucket management snapshop in SCSTallocator

homogeneous pages.

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 three allocators: CSTallocator, the sized allocator, and the SCSTallocator.

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 three allocation schemes, CSTallocator, sized allocator, and SCSTallocator. Table 1 shows the parallel applications used in our simulation and their characteristics.

Table 1. The parallel applications used in simulation and their characteristics

Application's Name	Memory Reference Length (10^6)	Working Set Size (MB)	Application's Function
Cholesky	39.341	2.88	Cholesky factorize a sparse matrix
Barnes	58.716	1.20	Simulate evolution of galaxies
Mp3d	8.935	2.02	Simulate rarefied hypersonic flow
Volrend	31.186	3.91	3D rendering using ray tracing technique

4.2 Experimental results

Table 2 shows how many false sharing misses are reduced in each parallel application when using each allocator. 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 unique object sizes when using the sized allocator, and the number of unique call-site IDs when using CSTallocator respectively. In case of the SCSTallocator, the number of buckets is equal to the number of unique (Call-Site ID, object size) pairs. These three allocators manage each object as a linked list using the separate pointers for its bucket. The shared pages with the same bucket pointers are assigned to the data objects with the same call-site ID or object size or (Call-Site ID, object size). Thus, the more buckets are found, the more sophisticated classification has been done.

According to our expectation, we can see from the Table 2 that the SCSTallocator outperforms the other two allocators. This means that the combined clue (object size + Call-Site ID) is much more effective in predicting the future memory access patterns of data objects than only the single clue. And we can also find out that the more classification buckets we use the more accurate prediction result we can obtain.

A disadvantage when using our SCSTallocator is the additional space overhead caused by using the more buckets. The next section shows the analysis results of each allocator's space overheads.

4.3 Analysis of space efficiency

For the strict performance evaluation, we need to analyze the space overheads caused by CSTallocator, the sized allocator, and the SCSTallocator respectively. The space overhead is the amount of additionally used pages by each method.

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

Table 2. Results of performance comparison of sized allocation, CSTallocator, and the SCSTallocator (page size = 4KB)

(a) Cholesky

Allocator \ Measure	# of buckets	# of false sharing misses	Reduction rate(%)
Sized	10	44,717	
CSTallocator	17	36,599	18.2
SCSTallocator	21	34,687	22.4

(b) Mp3d

Allocator \ Measure	# of buckets	# of false sharing misses	Reduction rate(%)
Sized	8	6,147,589	
CSTallocator	5	5,754,143	6.4
SCSTallocator	11	5,312,768	13.6

(c) Barnes

Allocator \ Measure	# of buckets	# of false sharing misses	Reduction rate(%)
Sized	27	5,805,705	
CSTallocator	7	5,104,413	12.1
SCSTallocator	29	4,814,970	17.1

(d) Volrend

Allocator \ Measure	# of buckets	# of false sharing misses	Reduction rate(%)
Sized	11	953	
CSTallocator	12	883	7.3
SCSTallocator	17	798	16.3

objects are mixed up into the same shared page regardless of their sizes or call-sites. So the allocation requests stream, *S*, of a general allocator is represented as:

$$S = \{s_1, s_2, ..., s_n\} \quad (1)$$

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:

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

On the other hand, when using CSTallocator or sized allocator, or SCSTallocator, the allocation request stream can be represented as follows if the order of requests is ignored:

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

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

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

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, the number of call-site IDs in CSTallocator, and the number of unique (allocation size, call-site ID) pairs in the SCSTallocator respectively. Thus, the following is valid:

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

The obvious fact we can obtain from the equations (5) is that the shared page overhead is no more than the number of buckets regardless of which bucket classification methods are used. Table 3 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 allocator 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. Another thing we can see in Table 3 is that the space overhead of our SCSTalocator is not quite large when taking into account its performance improvements.

Table 3. Space efficiencies of the three allocators (Page size = 4KB)

Parallel application programs (total # of pages needed in general allocation method)	# of additional pages (space overhead (%))		
	Sized allocation	CSTallocator	**SCSTallocator**
Cholesky (738)	10 (1.36)	17 (2.30)	21 (2.85)
Mp3d (553)	8 (1.45)	5 (0.90)	11 (1.99)
Barnes (308)	27(8.77)	7 (2.27)	29 (9.42)
Volrend (441)	11 (2.49)	12 (2.72)	17 (3.85)

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. Our allocator, called SCSTallocator, can reduce the false sharing misses more effectively than the two existing allocation schemes, sized allocator and CSTallocator. In SCSTallocator, both the object size and the call-site ID are used for predicting the future access patterns of each object. So only the same sized objects requested from the same location in program codes are allocated into the same shared pages. We are sure that the combined clue predicts the programmer's intention more accurately than the single clue, and find out by execution-driven simulation that the SCSTallocator outperforms the existing allocators. The SCSTallocator additionally spends pages only as many as the number of unique (object size, call-site ID) pairs. That is, our method can reduce more false sharing misses at the small sacrifice of space overhead. We expect that our SCSTallocator can contribute to both the reduction of false sharing

misses and the reduction of the cost on keeping the memory coherency in DSM systems.

In the future, 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] Tanenbaum, A.S.: Distributed Operating Systems. PRENTICE HALL (1995) chap. 6, 333-345

[2] Lee, J., Cho, Y.: Page Replication Mechanism using Adjustable DELAY Counter in NUMA Multiprocessors. J. Korean Institute of Telematics and Electronics B **33B(6)** (1996) 23-33

[3] Jeremiassen, T.E., Lam, M.S., Hennessy, J.L.: Shared Data Placement Optimizations to Reduce Multiprocessor Cache Miss Rates. In: ICPP 1990. vol. **II**(Software) (1990) 266-270

[4] Eggers, S.J., Jeremiassen, T.E.: Eliminating False Sharing. In: ICPP 1991. vol. **I**(Architecture) (1991) 377-381

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

[6] Lee, J., Cho, Y.: An Effective Shared Memory Allocator for Reducing False Sharing in NUMA Multiprocessors. In: IEEE 2nd ICA^{3}PP 1996. (1996) 373-382

[7] Adema, R.L., Ellis, C.S.: Memory Allocation Constructs to Complement NUMA Memory Management. In: IEEE 3rd Symposium on Parallel and Distributed Processing (1991)

[8] Lee, J., Kim, M., Han, J., Ji, D., Yoon, J., Kim, J.: Effects of Dynamic Shared Memory Allocation Techniques on False Sharing in DSM Systems. J. Korean Information Science Society(A): Computer Systems and Theory **24(12)** (1997) 1257-1269

[9] Lee, J., Kim, S.D., Lee, J.W., O, J.: CSTallocator: Call-Site Tracing based Shared Memory Allocator for False Sharing Reduction in Page-based DSM Systems, In: 2nd Int. Conf. on High Performance Computing and Communications (2006) 148-159

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

[11] Veenstra, J.E., Fowler, R.J.: MINT: A Front End for Efficient Simulation of Shared-Memory Multiprocessors. In: 2nd Int. Workshop on Modeling, Analysis and Simulation of Computer and Telecommunication Systems (1994) 201-207

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

[13] Woo, S.C., Ohara, M., Torrie, E., Singh, J.P., Gupta, A.: The SPLASH2 Programs: Characterization and Methodological Considerations. In: 22nd Annual Int. Symposium on Computer Architecture (1995) 24-36

[14] Berger, E,D., McKinley, K.S., Blumofe, R.D., Wilson, P.R.: Hoard: A scalable memory allocator for multithreaded applications. In: 9th Int. Conf. on Architectural Support for Programming Languages and Operating Systems (2000) 117-128

[15] Berger, E,D.: Memory Management for High-Performance Applications. PhD thesis, University of Texas at Austin (2002)

[16] Michael, M.M.: Scalable Lock-Free Dynamic Memory Allocation. In: ACM SIGPLAN 2004 Conf. on Programming Language Design and Implementation (2004)